Agostino Cortesi and Flaminia Luccio (Editors)

ACM - IFIP

Informatics Education Europe III

Proceedings

December 4-5 2008-09-22 Venice, Italy

Proceedings of the ACM-IFIP IEEIII 2008
Informatics Education Europe III Conference
Venice, Italy, December 4-5, 2008

Informatics Education Europe III

Editor’s detail

Agostino Cortesi, Flaminia Luccio
Computer Science Department
Università Ca’ Foscari
Venice, Italy

ISBN 978-88-903433-0-8

Program Committee

Proceedings of the ACM-IFIP IEEIII 2008
Informatics Education Europe III Conference
Venice, Italy, December 4-5, 2008

Table of Contents

Preface

Agostino Cortesi and Flaminia Luccio

Dipartimento di Informatica, Università Ca' Foscari di Venezia, cortesi@unive.it, luccio@unive.it

This volume contains the papers of the International Informatics Education Europe Conference being held in Venice, Italy, on 4th and 5th December 2008.

The purpose of IEE is to provide a forum for researchers interested in higher-education of Informatics. Topics covered by IEE include: Accreditation and assessment, Innovative degree programs, Innovative uses of technology in the classroom, Partnerships with industry, International cooperation, double degrees and mobility, Integrating gender and culture issues into informatics curricula, Debugging tools and programming learning, Expanding the audience for informatics, Funding opportunities for curriculum development and studies, and Collaborative learning.

This was the third IEE meeting. Previous meetings were held in Montpellier (2006) and Thessaloniki (2007).

The program committee selected 18 papers out of over 30, on the basis of at least three reviews. The principal criteria were relevance and quality. The program of IEEIII includes in addition three invited talks by Jan van Leeuwen, Rustan Leino and Carlo Ghezzi, and two talks by representatives of the industrial partners Ibm and Intel.

I would like to thank the program committee members and the reviewers without whose dedicated work the conference would not have been possible. A special mention has to be made to Andrew McGattrick of the ACM Education Board, and Gordon Davies chair of IFIP 3.2. group.

Thanks also to Emanuela Boschetto, Andrea Marin, Sonia Barizza, Sandra Scibelli, Loretta Fornaser and Gian Luca Dei Rossi for their organizing support.

The Conference has been fortunate to receive sponsorship from ACM, IFIP, Intel, IBM, Microsoft, AICA, and CRUI.

Why mobility is important for European students in computer science: review of 18 years of a Franco-German university training in with a double degree

Gabriel Michel

Université Metz, Isfates, Ile du Saulcy, Metz, France, Gabriel.Michel.Michel@univ-metz.fr

For students in computer science mobility abroad is still relatively rare compared to other disciplines. Yet mobility is often a genuine social elevator for students who experienced it, especially after having obtained a Franco-German degree in computer science. This is what our empirical study based on several surveys of French and German alumni after the bi-national training of ISFATES (Franco-German Institute of Technology, Economics and Sciences) and after obtaining a dual French and German degree proves. This study also demonstrates the importance of "soft skills" and especially intercultural skills. Indeed we show that those students who had followed in their overall training up to 20% less computer science courses than in national education programs nevertheless generally have far better careers in computer science than the latter.

Keywords
Double degree, CS Curriculum, Engineering, France, Germany, Intercultural, Learning, Professional skills, Surveys, University cooperation.

1. Introduction

In most professions, it is now necessary to master new skills such as autonomy, the ability to communicate (in different languages), flexibility, language and culture, mobility and innovation. To prepare future graduates for these changes, the education system should introduce students to these new skills and knowledge, often grouped under the name "new skills" [11]. But in computer science, since the job market has always been in favourable, the stress has always been focused on technical subjects. And in most departments of computer science "soft skills" were and still are, regarded as secondary.
In this article we want to demonstrate that these soft skills and in particular, knowledge in languages and intercultural skills are very important and offer far better career-advancement opportunities for students in computer science. We rely on our experience of nearly 30 years of double degree (18 years for computer science). There is not or hardly any research that has been done on evaluating the mobility of students abroad despite the fact that it is one of the vital challenges of university education in the coming years (in Europe in particular). The intercultural domain is treated by scientists in regards to acculturation [13] culture shock [2, 3], work (or how to avoid failures during an assignment abroad - Training for Intercultural Competence: Avoiding Failures in International Assignments) [15], the differences between cultures [9], learning styles [12] or skills [1, 5, 8]. But measuring the effects of student mobility and intercultural learning in computer science has rarely been an object of study.

In Europe, Socrates / Erasmus programmes promote student mobility: it concerns semesters of study or internships in another European country. The project Valera "Value of ERASMUS Mobility" of the EU [14] aims to establish the impact of mobility within the sub-ERASMUS programme of SOCRATES on the mobile students' and teachers' careers. With respect to student mobility, professional "success" was measured primarily in terms of general and international competences, transition to work, first and subsequent employment and work, and international aspects of employment and work. According to the most recent survey (four fields of study were selected: Chemistry, Sociology, Mechanical Engineering and Business studies), the impact of ERASMUS is very good for the students if we compare this with those students without international experience (higher status, higher earnings as well as a better chance of reaching a position appropriate to their level of education, better career opportunities, higher competences.

International experience notably seems to reinforce adaptability, initiative, the ability to plan and assertiveness, higher socio-communicative skills as well as better ways of problem-solving and leadership). More than half of the formerly mobile students assess their knowledge and understanding of international differences in cultures and societies, and almost half their knowledge of other countries is as important for their job tasks. These proportions are mostly somewhat higher than in previous years But the impact of ERASMUS is smaller than before according to surveys of previous generations for graduates in obtaining a first job, getting a higher income and taking over job tasks for which visible international competences are needed. This is most likely caused by a growing internationalisation in general that leads to a gradual decline of the uniqueness of the ERASMUS experience. But now the value of experience abroad as such is declining in the wake of the general internationalisation of the environment.

But the number of mobile students is still small compared to the general student population and depends both on the country, on scientific fields and also… on the parents income. In France students who are mobile are still mostly students who major in business and management, who graduated from reputable business and engineer schools and who take advantage of this mobility that has been imposed on them. Parents and / or banks are the financial support of these students whose future was in any case very favourable from the outset.

At the other end of the hierarchy of higher education are the general universities, whose students come mainly from popular classes. For them mobility is very difficult because neither is it organized (no information, little or no cooperation agreements, no mobility grants, …) nor recognized (no validation of units acquired abroad), without mentioning the cost of mobility. Such mobility is particularly low in the computer science fields (for example in 2005, only 0.3% of French students in mathematics and computer science experienced Erasmus). Yet for these students, this mobility is possible and would be extremely beneficial particularly in the context of a more competitive Europe. That's what we try to show on the basis of our experience. To do this we will first present this institute, its history and its functioning, then the results of several surveys that we conducted with former students. Then we try to analyze what made these students successful, how we can develop their intercultural skills and how we have integrated them into our curriculum.

2. Presentation of the Institut

2.1 History

After the Second World War, the construction of the European union was a means of avoiding the conflicts of the past. It was necessary to build a political, a cultural and an industrial Europe as well as an academic Europe. Isfates was created in this will in order to pose the first stepping stones of a European education. The institute was created on September 15th, 1978 during a meeting at the Franco-German summit of Aix-la-Chapelle: its statutes come from a convention signed by the German Chancellor Helmut Schmidt and President Giscard D' Estaing. This operation was the first of its kind on the European university level. This Franco-German university institute was created to develop economic ties between France and Germany by training a personnel qualified for the light industries that are rising in this epoch (the mechanics of precision, electronics...) and by training technical and commercial specialists that are really bilingual.

The fact that the institute was created in the towns of Metz (in France) and Saarbrücken (in Germany) is not by chance as the two cities are only 70 kilometres away from each other which makes it possible for the students to move easily from one city to the other throughout their training. The choice of these two cities was symbolic. Robert Schuman originated from Metz and the two cities also share a common story: in a little more than a century they continued to pass from the hands of one country to the other.

Since its creation this institute has not ceased to develop and adapt. Today, nearly 2,200 French and German students follow the bi-national training of ISFATES and obtain a both a French and German university diploma. The institute is at the very least unique and original since it had allowed, until 1999, students having succeeded their first 2 years of higher education in their country of origin, to then continue their studies for the next two years in alternation between Saarland (one of the German “lands”) and the University of Metz.

ISFATES has made it possible, from the beginning, to obtain a French “licence” and a German engineer diploma in three fields: electrical engineering, mechanical engineering and company management and economics. The computer science department was created in 1990, and other departments such as the civil engineering and logistics department were soon to follow. Since 1993, students can obtain a masters degree at the University of Metz in the field of their choice (as well as the German engineer diploma). And since 1999 students have been able to enter ISFATES directly after the baccalaureat (highschool diploma), its duration having been extended to four years. This new structure thus offers students a course comparable with the national university programmes.

The ISFATES-DFHI has been used as a model for many trainings that have been set up since 1988 under the aegis of the Franco-German College for higher education, then, since 1997 under the aegis of the Franco-German University / Deutsch-französische Hochschule (UFA/DFH), created at the time of the Franco-German summit of Weimar. The purpose of this institution is to bring together all of the cooperations integrated between higher French and German educational establishments. Currently gathering 150 establishments, 5000 students were registered with the UFA for the academic year of 2007/2008. The ISFATES-DFHI, with its 400 registered students, remains today the most significant Franco-German training delivering at the same time French and German diplomas.

2.2 Operation of the Institut

Currently Isfates has 6 branches : Computer Science, Logistics, Management Sciences, Civil Engineering and Infrastructures, Mechanical Engineering and Industrialized Manufacturing and Engineering Systems. For each branch, a group of 15 to 30 French and German students " will travel " together each year from one university to the other. During these 4 years alternatively divided between the University of Metz (France) and the HTW Saarland (Germany), the French and German students attend all of the courses jointly (of the national host branch of reception), which allows a permanent and enriching contact with the other culture. The students from the six branches of a given year also have some “inter-branch” courses in order to maintain a certain cohesion. The course of study is as follows:

- 1st and 2nd semester in Metz,
- 3rd and 4th semester in Saarbrücken,

- 5th and 6th semester in Metz,
- 7th semester in Saarbrücken,
- 8th semester period training course in a company outside the linguistic area of origin.

This training leads to a Bac + 4 in each of the branches leading to the following qualifications:

- An engineer diploma (Abschluss DER HTW Saarbrücken).
- A master's degree (maitrise) at the University of Metz.

The teaching staff is in itself bicultural and meets regularly and alternatively in the 2 countries.

3. Alumni surveys

We present here the two surveys which were carried out with all students from all sectors: the results of the computer science sector were not significantly different from those of the other sectors. These results are thus also valid for the computer science specialists.

3.1 First Survey (1995)

This survey was sent to 800 alumni of the old structure: 317 questionnaires were returned. It concerned alumni who graduated from the institute between 1980 and 1994. They had completed the Isfates curriculum in Bac+3 and Bac+4 and had obtained the French "licence" and German Engineer Diploma.

Out of the 500 questionnaires that had apparently reached their destination, 317 returned completed within one month after sending them (which constitutes a very strong and significant return rate). Of the 317 questionnaires at our disposal, only 299 were usable. Out of the 299 responses, 127 came from former German students, 172 from former French students. 196 engineers responded (1/3 German, 2/3 French) and 103 managers (47% German, 53% French). The average age of the subjects concerned in this survey was 29 years old and they had had 5 years of professional experience on average.

We will describe only a part of the results of this survey: certain aspects, such as their motivation for coming to Isfates, pedagogy, marks, contact with teachers, the quality of teaching offered at the two sites, the autonomy granted to students, how their first job was found, and the size of their company will not be revealed in this article. But these data were used to cope with the various reforms of recent years.

We will here describe the responses to the following questions: who they were before joining the institute, in which countries they found a job, what the nationality of their first employer was when they were expatriates, how much they earned, what were the essential skills offered by this training for their professional life and would they repeat the same training.

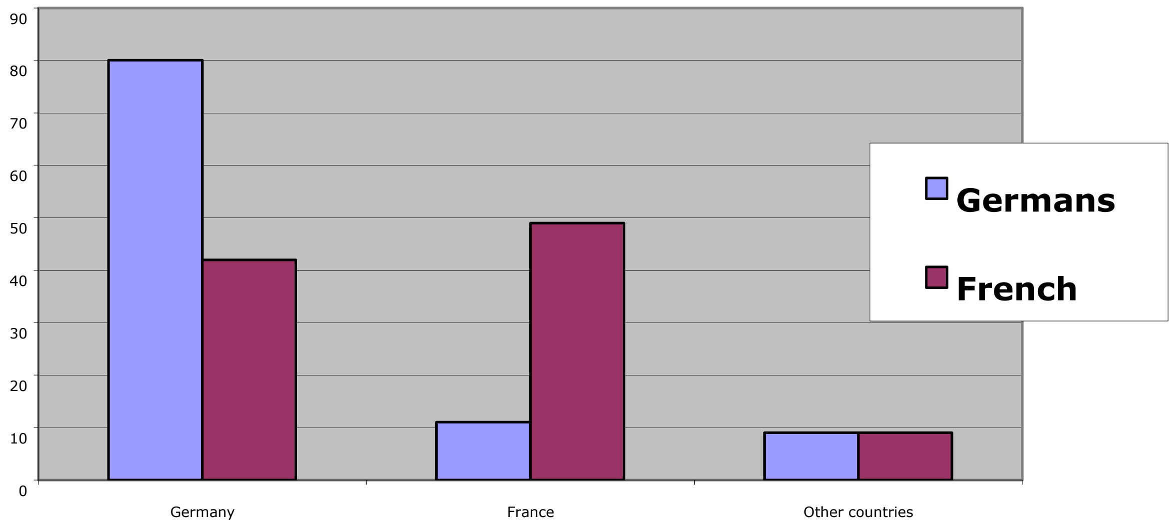
• Profession of father: 70% of French and 52% of German alumni belonged to the most disadvantaged social-professional classes : workers (miners often for the French), employees, farmers, civil servants of a basic category.

• In which country did you find your first job (in %)?

The survey concludes that there is a strong mobility for the first employment, especially for the French (42% in Germany, 9% in another country against 49% in France). Thus a little over half of French students expatriated after their degree (especially to Germany) in contrast to 21% of German students (12% + 9%).

For the French, attractive higher salaries abroad (in Germany in particular), is certainly the essential explanation of this mobility.

Figure 1 Mobility country of your first employment.



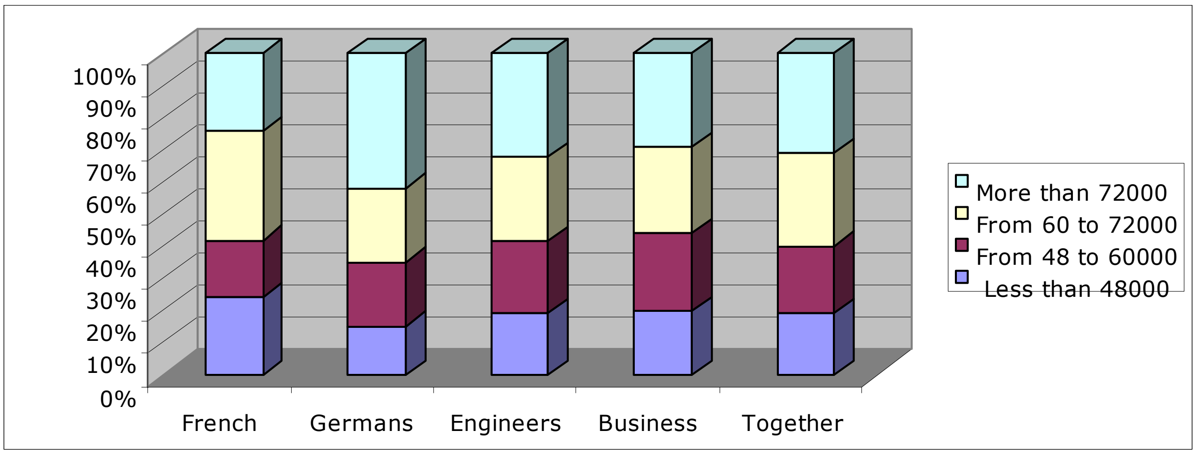
- Nationality of the company for expatriates at the time of their first employment. The aim was to assess the number of French and German alumni who left their country of origin for their first job.

Most expatriates worked in companies in partner countries. (for example the French worked primarily in Germany in a German company, only 10% of French expatriates worked in a French company).
Nearly 30% of the French alumni worked in a company in a third country. This phenomenon was even more significant for German expatriates. This can be explained by the attractiveness of an international curriculum, even if this curriculum does not directly integrate knowledge about this third country. This enforces the assumption that already in 1995, intercultural skills were acknowledged and recognized by international companies.
The results show that relatively few expatriates abroad worked in companies from their country (keeping in mind that in 1995 the number of the French companies operating abroad was much lower than today).

- What is your gross annual salary in DM?

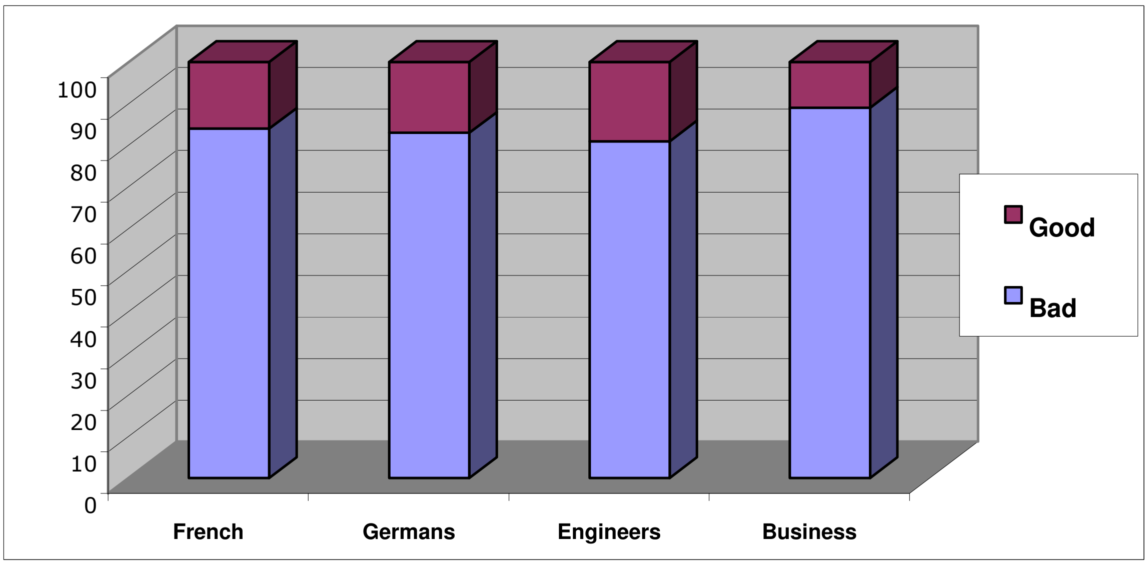
The result of this question showed an annual average gross salary of 65000 DM for the average 29 years old age group and 5 years of professional experience. In 1995, this corresponded to the wages of engineers from reputable schools in France and to that of engineers of good technological universities in Germany. Note that these wages were obtained by students after four years of training (instead of five in the well-known schools), in university establishments that are not the most reputable (Metz and Saarbrücken) and coming from, in the majority of cases, from the underprivileged social classes. Whereas in the renowned schools around 7% of the pupils came from the " disadvantaged" classes. At Isfates nearly 70% at that time were from the “disadvantaged” classes.

Figure 2 Annual Wages in DM



- How you see your professional future (in %)

Figure 3 Your professional future

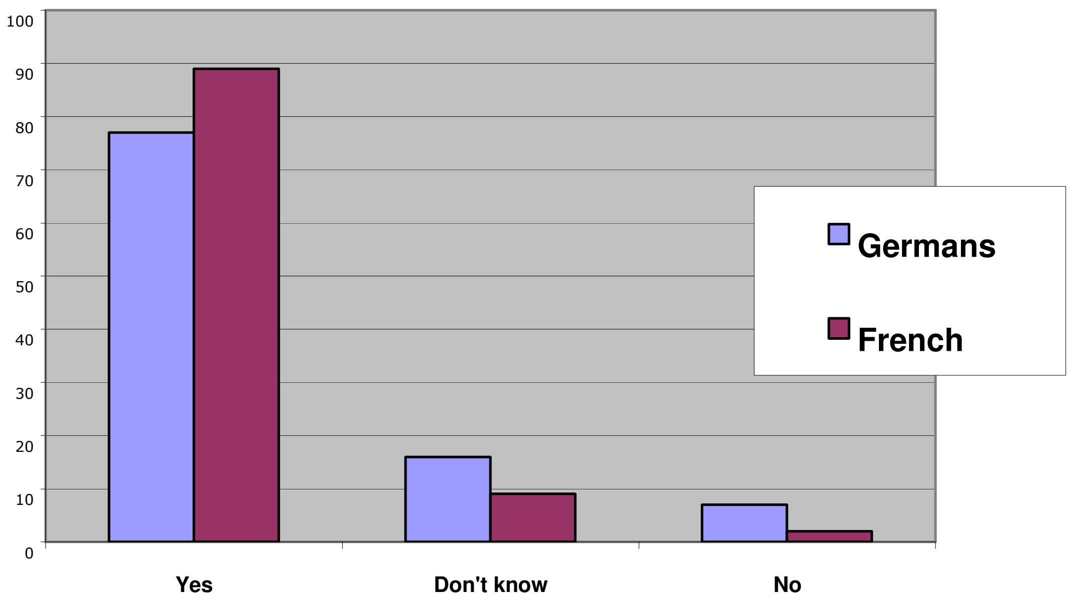


Here again, we can see a certain optimism whatever the sectors or nationalities : 84% of French felt good about these prospects! Considering that the French culture is known for its pessimism in regard to the future, this is an excellent result.

- If you had the choice, would you reattend Isfates?

Overall 89% of French alumni (9% did not know and 2% would not recommence this training) and nearly 80% of German alumni would agree to repeat this training (a little more the technicians than the managers). This is an excellent result, and very significant, because among the 9% of the French alumni who did not know, or who would not repeat this training, it is necessary to also include those who were aware of having chosen the wrong field. The Germans were probably a little less enthusiastic because the social elevator was a little less significant for them (the social-professional levels of their parents being a little higher) and the net worth wages which they earned thanks to the diploma was a little less significant for them than for the French. If one asked the question " If I had to start again? " to a 100 managers of an average age of 29 years and working for approximately 5 years on average, how many would say they do not regret their choice of training?

Figure 4 And if were to be redone?



3.2 Second alumni survey (2002)

This survey was carried out in July and August 2002 in the perspective of a renewal of Isfates. At that time the essential problem was to make the courses more attractive, in particular in the scientific field affected by both the decline in the vocations in the field of engineering (still of more an advantage in Germany than in France), and the drop in the number of high-school pupils having learned German in France and French in Germany. Whereas the first survey raised the question: " What have you become and what do you think of us? ", this second survey was primarily focused in responding to recent concerns such as:

- taking into account the current situation, " what type of courses should be offered in order to make the institute more attractive? "

-" is the masters degree necessary and which LMD should be put into place? "

Meanwhile we had also taken the questions of the first survey again in order to determine whether we would still have the same results: career, mobility, wages, vision of their future professional life and satisfaction with their training are the questions which we repeated. We will reveal in this part only this last data.

Overall the sample was comparable to that of the survey of 1995: the same proportion of managers (39%) and engineers (51%), a type of professional situation slightly more favourable than the father. The average age of the subjects covered by this survey was 30 years old and they had had 5 years of professional experience on average. 70% of them had obtained their diploma from Isfates after 1995, and thus had not been consulted at the time of the first survey: they were primarily recent graduates.

The answers to these various questions confirmed the 1995 survey: there was still a majority of students that came from more disadvantaged social-professional levels, a mobility as significant for the first job, annual wages still high in taking into account their level of studies (comparable with those of students leaving good engineering and business schools). Although the survey was conducted during a period of economic recession in France as well as in Germany during the summer of 2002, the old optimism remained. Their professional future was seen as " good or very good " for 69% of the alumni, " not good " for 11%, while 20% did not know. Finally the very significant indicator of satisfaction with this training when asked " If I had to start again, would I? " : 83% answered yes, 7% no and 10% did not know.

3.3 Some other indicators

Since the creation of the computer science department of Isfates in 1990, one could note very visible differences between the students who followed this curriculum compared to the students of the national education French and German sectors. Most of these differences were also observed in other sectors, but we will focus on computer science students. In the computer science subjects these students obtained similar results to those obtained by students from the national education sectors. There was no significant difference in the average mark of a group of computer science students from Isfates with the average mark of students from the national group (even if the students from Isfates had completed, in overall, less computer science courses throughout their training compared to other students). On the other hand the difference was extremely favourable for Isfates when it was a question of finding a training course or an employment, in the average duration of research, the type of missions given or salaries earned. These Franco-German students found their first training course or their first job much more quickly, had a very interesting position, and had higher paid salaries. And this variation was even more visible during the years of the computer science crisis. Indeed the students from Isfates were hardly affected in these years of crisis. Let us notice that in a period of full employment in computer science, the number of candidates in the computer science department of Isfates fell because mobility was not necessary to find a job. And of course as soon as the job market in computer science weakened, one could note an increase in the number of candidates again.
Therefore the students of Isfates have always interested employers, as much as the well-known engineer schools and universities. Thus, those wishing to continue their studies after their fourth year had their student files very easily accepted (much easier than those from the national education) even in the most renowned establishments in France. We have thus been able to set up an agreement with the Polytechnique School of Montreal, known to be a very selective establishment for European computer science specialists and which only accepts students from the most reputable schools in France and those of the best German technological universities. Today, Polytechnique, noting the adaptability and excellent results of Isfates, is opening its doors more widely to the students of Isfates and is continuing to accept an increasing number of our students (whereas the number of candidates for this school has not decreased).
A last indicator is the number of prizes obtained for the best training courses carried out by the students of the two countries. In Isfates, and in the computer science sector in particular, students regularly receive prizes of having carried out the best training course, whereas students from the national education (having a much more significant manpower: from 4 to 7 times more depending on the year and locations) only obtain prizes very exceptionally.

4. Discussion

4.1 The cultural skills that pay

The results of the studies we have undertaken show that this type of training offers students interesting career-advancement opportunities. The high level of satisfaction of former students who would be willing to repeat the same career choice (89% and 83% according to the survey) can be explained in the following way: even if the direction they had chosen at the age of 18 or 20 years was not necessarily the best compared to their tastes today, the position they currently occupy, and also the professional opportunities encountered enabled them to have a job that pleases them. The awareness of the effect of a social elevator (remembering that over 50% of students had grants) is certainly another reason not to regret their choice.
If we compare our surveys with the results of the Valera study on Erasmus, one finds that the benefits of mobility is much more favourable for students from Isfates. This can be explained by the following reasons:

- The duration of mobility: the students from Isfates carried out either two semesters (in the old system), or four semesters (in the intermediate system), or five semesters (in the current system) abroad. Erasmus does only one or two semesters abroad.
- Integration: even when the semester takes place in the country of origin, the student is in a bicultural group. Moreover, the programmes of the different semesters are synchronized. This is not the case for Erasmus.
- Diploma: Isfates students receive the diploma from both countries which is not the case for Erasmus. The importance of a double degree no longer needs to be proven: for example, most German companies in computer science technology do not know the French diplomas. For a French person, having the German diploma allows him easy access to the German labour market.
- Economic ties are very developed between the two countries. The Erasmus survey covered four specialities of which two are general (sociology and general chemistry) and where mobility seems, a priori, to be not so great. For Isfates, its six specialties are in great demand in the international market.

We also asked the alumni in these questionnaires, and in many interviews, to classify three skills (specific knowledge in the field, knowledge of the language and intercultural skills) that they considered to have contributed to the success of their careers. Intercultural skills were in majority considered to be the first, followed by language skills and finally academic knowledge. Obviously a good academic level is indispensable, but intercultural skills are increasingly needed in the job market, and appear to make a difference. It is the reason why we have taken advantage of the various reforms of the institute by integrating, each time a little more, these skills into our educational programmes.

4.2 Taking into account the results of the survey: intercultural skill development in the curriculum

In the computer science department, as well as in all other departments, each group is made up of 50% Germans and 50% French. So even when French students spend their academic year in France, they study each day with German students and a have a certain number of German teachers. All projects in computer science must be bicultural meaning a group working on the project must necessarily have both French and German students. We gradually introduced specific courses aimed at developing language and intercultural skills. These courses consist of presentations on the cultures of each country, conferences, projects to be realized by the students, company visits, etc.… Moreover, the importance we grant these courses is illustrated by the specific number of dedicated hours and their weight in the marking system. Thus intercultural and language lessons account for 30% of the number of students' hours. To illustrate these choices, here is the content of the courses taken by first year students in the computer science department.

First Semester: (4 Units: Each Unit consists of subjects each representing 48 hours or 24 hours per semester)

- M1 Languages and intercultural education
  - German / French 48 h.
  - English 48 h.
  - Intercultural 24 h.
  - Knowledge of companies 24 h.
- M2 Core Subjects
  - Mathematics 48 h.
  - Computer Science 48 h.
- M3 Programming
  - Algorithmic 48 h.
- M4 Management
  - General Economy 48 h.
  - Commercial law 48 h.

2nd Semester (4 Units)

- M1 languages and intercultural education
  - German / French 48 h.
  - English 24 h.
  - Intercultural 2   24 h.
- M2 Core Subjects
  - Mathematics 48 h.
  - Computer Science 48 h.
- M3 Programming
  - Databases 48 h.
  - Algorithmic 48 h.
- M4 Networks
  - Internet 48 h.

The "languages and intercultural learning" units and the "management" course units are taught in all of the departments : it allows computer science students to not remain only among themselves and to have ongoing contacts with students from other departments. In the first semester, rather than starting immediately on specific computer science subjects we have decided to offer many non-IT units. Here are the reasons:

- To allow German students, who often have an average knowledge of the French language on their arrival in the first year in France, to improve their level of French so as to not fall behind in the technical courses (which really begin in the 2nd semester)
- To reduce the dropout rate (frequent in the 1st year at universities) by giving a chance to students who come from high schools to gradually adapt to the pace of university life and to adapt to the country.

For the following three years the programme consists of close to 30% of language and intercultural education. We recognize the difficulty confronted in setting up these type of lessons as it is a new field, is little recognized and is often poorly accepted by the students. Explaining the choice of "sacrificing teaching hours to something other than computer science" has not been easy with both students and among teachers. Indeed in the computer science department (as in other departments) the same resistance and remarks were made: computer skills are essential, and any other subject (sometimes even languages) were not a priority, the academic level of the students at the end of the training would be too low, other subjects could be self-taught, it was a waste of time and would undermine the value of the diploma…

The results of our surveys, the facility with which each year our graduates found an employment, and an experience of nearly 30 years of Franco-German exchange helped to convince the different teaching teams to devote a significant part of the programmes to "soft skills" and in particular to language and intercultural skills. Even today when a new computer science colleague joins the teaching staff of the computer science department, his first thoughts on the training are still the same and he has again to be persuaded.

5. Conclusion

Studying abroad and obtaining a double diploma is, for a computer scientist, a passport for an even more successful future career than if he had not been mobile. And this is the case even if in the beginning this computer scientist is not particularly brilliant or good at languages, nor coming from an elevated social class. Our study proves this. The surveys carried out with the Isfates alumni confirm that such a training allows a greater mobility, a starting salary significantly higher than those holding an equivalent national diploma and a very optimistic vision of the alumni's future professional life. The study also shows that most alumni would be ready to start over again and that this training is a very effective social elevator.

This study proves that this type of training is a very important asset, even for a future computer science engineer who does not have to worry much about his early career. The recipe for Isfates is as follows: a good basic training in computer science plus current practice of foreign languages, a permanent cultural bath, and double diplomas.

This study also demonstrates the importance of "soft skills" and especially intercultural skills. In fact, those students who had, in their overall training, up to 20% less computer science courses than in the national education programme have a far better career than the latter.

Currently Isfates has the system of Bachelors and Masters Degree:

- For the Bachelors Degree: three semesters take place in Metz and three semesters in Saarbrücken in the end obtaining a joint Franco-German Bachelors Degree. It is also possible to spend one semester with Erasmus in another country.
- For the Masters Degree: three semesters of study followed by a semester of internship in a third country. For the three semesters of study, the first takes place in Saarbrücken, the second in a third country or Saarbrücken and the third in Metz. In the end a joint Franco-German Masters Degree is obtained.

This year the computer science department of Isfates has joined the network ECS (European Computer Science) [6] which allows students to do the 3rd year of their bachelors degree in one of universities belonging to the network (Burgos in Spain, Turku in Finland, Coimbra, Portugal, Huddersfield in England) and in addition to obtain the diploma from this country. That makes a total of three diploma's in three different countries at the end of the Bachelors Degree. This will entail even more intercultural skills and certainly better careers prospects.

These experiences of setting up joint trainings between different countries should be taken into account in order to arrive at a coherent European university training in computer science involving an optimal mobility of students. This is already what a project such as Euro-Inf [7] has; its goal is to create a coherence among all trainings in computer science from the European standpoint.

References

1. Beamer, L. Learning Intercultural Communication Competence, The Journal of Business Communication, 1992 ; vol. 29, n°.3, p. 285-295.

2. Berry, J. W., Segall, M. H., & Kagitçibasi, C. (Eds.). (1997). Handbook of cross-cultural psychology. Social Behavior and Applications (2 ed.). (Vol. 3). Boston, MA: Allyn & Bacon.

3. Camilleri, C. & Cohen-Emerique, M. (Ed.) (1989). Chocs de cultures: concepts et enjeux pratiques de l'interculturel, Paris : L'Harmattan, p. 363-398.

4. Deutsch-Französische Hochschule - Université franco-allemande http://www.dfh-ufa.org/

5. Dinges, N., K. Baldwin. Intercultural Competence. A Research Perspective, In D. Landis et R. Bhagat (dir.), 1996 ; Handbook of Intercultural Training, London, Sage, p.106-123.

6. ECS European Computer Science http://www2.turkuamk.fi/ECS/index.php?option=com_content&task=view&id=46&Itemid=44

7. Euro-inf The Euro-Inf Project - European Accreditation of Informatics Programmes http://www.euro-inf.eu/

8. Hampden-Turner, C. et A. Trompenaars. Building Cross-Cultural Competence. 2005 ; London, Yale University Press.

9. Hofstede, G. Culture's Consequences. International Differences in Work-Related Values. 2001; London, Sage.

10. ISFATES-DFHI Institut Supérieur Franco Allemand des Techniques, d'Economie et des Sciences - Deutsch-Französisches Hochschulinstitut http://www.isfates-dfhi.eu

11. Morin, E., Les sept Savoirs nécessaires à l'éducation du futur, 2000 ; Paris, Seuil.

12. Oxford, L. R., N. J. Anderson. A crosscultural view of learning styles. » Language Teaching, 1995 ; Cambridge University Press, n° 10, p. 201-215.

13. Redfield, R., Linton, R., Herskovits, M.J., Memorandum on the study of acculturation, in American Anthropology, 1936 ; n°38,

14. Valera "Value of ERASMUS Mobility" of the EU http://ec.europa.eu/education/programmes/socrates/erasmus/evalcareer.pdf 2005

15. Vulpe T. Training for Intercultural Competence: Avoiding Failures in International Assignments. CERC news. Issue No. 132 http://www.dfait-maeci.gc.ca/cfsi-icse/cil-cai/pdf/cerc_article-en.pdf 2004

Accreditation practice for degree programs in Computer Science: Experience gained at a classical research university in Germany.

Thomas Ottmann, Christoph Hermann[1], Christoph Heumann[2]

[1]Georges-Köhler-Allee Geb. 51, 79110 Freiburg, Germany, ottmann@informatik.uni-freiburg.de; hermann@informatik.uni-freiburg.de

[2] ASIIN e.V., Robert-Stolz-Straße 5, 40470 Düsseldorf, Deutschland heumann@asiin.de

In this paper we outline how large German research universities handle the Bologna reform. We in particular describe how they fulfill the requirement to accredit their new undergraduate and graduate programs. A description of the general framework is complemented by specific examples and experiences gained in the accreditation process of the computer science programs at the University of Freiburg. We in particular illustrate the positive effects of peer review in the accreditation process for the design and implementation of the new degree programs and disclose hurdles solved for the accreditation. It turns out that the external program accreditation of all degree programs is complex and expensive. Therefore, large traditional universities go for the so called systems accreditation. The aim is to make external program accreditation superfluous once an efficient internal quality management system is implemented and operational. We outline some advantages and disadvantages of system accreditation (*Systemakkreditierung*) versus program accreditation and conclude with lessons learned during our accreditation experience.

Keywords

Program versus System Accreditation, Quality Control, Changing Computer Science Programs, Bologna Reform, German Research Universities

1. General framework for accreditation

In the German federal system, the responsibility for higher education mainly lies with the 16 federal states (*Bundesländer*) and their respective ministries for higher education. Parallel to the introduction of the two-cycle structure with Bachelor's and Master's programs by Germany's universities as a result of the Bologna process, quality control has been transferred from the state ministries of higher education to accrediting bodies founded by the higher education institutions (HEIs) themselves, sometimes in co-operation with other stakeholders in higher education. This change goes along with a transfer of responsibility from the state to the HEIs, leading to greater autonomy on the part of universities, but also to new administrative responsibilities and to requirements regarding the documentation of the quality of their processes, also with regard to the development and implementation of the new Bachelor's and Master's degree programs. In the past, the state ministries of higher education had a much more direct influence on the state-funded universities and their degree programs. Nowadays, most state higher education laws just contain the requirement that degree programs must be accredited.

The accreditation system in Germany is regulated by the Accreditation Council (*Akkreditierungsrat*) which was established by the 16 federal states as a foundation under public law. The Accreditation Council sets the standards and guidelines for the accreditation agencies and awards the right to accredit degree programs. Currently, six accreditation agencies have been authorized by the Accreditation Council to award its Seal to accredited Bachelor's and Master's degree programs: ACQUIN, AHPGS, AQAS, ASIIN, FIBAA, and ZEVA[1].

Three of these cover the entire breadth of academic education, the other half focuses its activities on certain subject areas. The accreditation agency ASIIN e. V., for instance, is specialized in the accreditation of degree programs in engineering and informatics/computer science, as well as mathematics, biology, physics, chemistry, geosciences and pharmacy. Most computer science programs offered at German universities and universities of applied sciences (*Fachhochschulen*, UAS) are accredited by ASIIN. ASIIN e. V. is organized as a non-profit association carried not only by networks of HEIs, but also by federations and trade associations (including the federal association of state engineering chambers), scientific societies and umbrella organizations from industry and trade unions – all of which are active in the fields of technology and natural sciences. It is supported by the national bodies representing the faculties of engineering and natural sciences at German universities and universities of applied sciences.

The Bologna reform has not only changed the administrative framework for quality control but also shifted the focus from learning inputs to educational objectives and learning outcomes. The objectives of a degree program should reflect the needs of the different stakeholders in higher education: academia, industry, state governments, and students. The accreditation process must ensure that degree programs seeking accreditation meet their objectives. The assessment method is the same for all German accreditation agencies: First, any degree program must specify its objectives; this requirement is then further elaborated by a number of key questions to be answered (what subject-specific and subject-independent competences/qualifications are being imparted, how is the employability of graduates achieved, what are typical occupational fields for graduates, what is the specific profile of the degree program, and further similar questions). Second, the concept of the degree program and its implementation must assure that the objectives are met and that the HEI has the necessary means to carry out the program. Some accreditation agencies, like ASIIN e. V., complement the general requirements and procedural principles by subject-specific criteria. These are conceived as parameters for orientation and comparison against this background, allowing for reasonable deviations and serving as an orientation for application and auditing of degree programs in the accreditation procedure. ASIIN e. V. has 13 technical committees (TC) for the various disciplines represented within ASIIN. They formulate learning outcomes and objectives, develop the subject-specific criteria and guidelines, nominate audit teams for accreditation procedures, and review and comment reports of audits to the Accreditation Commission. This is not done in isolation but in discussion with the scientific organizations, future employers of graduates, and students. In addition, ASIIN e. V. is part of international networks aiming at establishing comparable outcome descriptors for higher education degree programs in specific disciplines on the European level, amongst them the Euro-Inf Project (European Accreditation of Informatics Programs).[2] With this project, a framework has been established that supports the coordinated further development of accreditation criteria for informatics degree programs in Europe between supra-national organizations such as the Council of European Professional Informatics Societies (CEPIS) and Informatics Europe as well as national bodies such as the British Computer Society or GRIN in Italy.

[1] http://www.akkreditierungsrat.de/index.php?id=5

[2] http://www.euro-inf.eu

Beyond the fitness of purpose of the Bachelor's or Master's degree program and the correspondence of its objectives to the needs of the stakeholders in the higher education process, the Standing Conference of the Ministers of Education and Cultural Affairs (*Kultusministerkonferenz*, KMK) of the 16 federal states has set common structural guidelines for Bachelor's and Master's programs, compliance with which is also checked in the accreditation process. These guidelines specify modularization, the award of credit points, and the duration of degree programs; they regulate entry requirements and transition, distinguish between different profiles and types of Master's programs, and clarify the equivalence of the new degrees with the traditional one-cycle diploma degrees.

In what follows we briefly characterize the different versions of how the learning outcomes of undergraduate and graduate programs in informatics/computer science are specified in the subject-specific criteria of the TC 04 of ASIIN, which is the technical committee responsible for informatics/computer science. The specification is by far less detailed than the one developed by the ACM/IEEE joint task force contained in the Computing Curricula 2001 Report [4]. The specification does not distinguish between different computing related fields but tries to roughly characterize the body of knowledge and competences to be expected from every graduate of a computer science program in about 10 pages. The subject specific competences comprise formal, algorithmic, and mathematical competences, analysis, design and implementation abilities, as well as technological and application-oriented competences. No detailed curricular recommendations and no sample programs or course descriptions are given. Thus, the requirements are much more generic than the ACM/IEEE recommendations or the guidelines compiled by the German Computer Science Society GI [5]. A typical example is the specification of the formal, algorithmic, and mathematical competences to be expected from every computer science graduate:

> *Graduates of any informatics program are able to analyze, structure, and describe real world problems by formal means. They can transfer formal requirements into correctly implemented and efficient solutions using currently available hardware- and software systems. They are able to identify the algorithmic core of a problem and have a good command of the respective algorithms, data structures, and patterns for solving problems. They are able to assess the correctness and efficiency of a solution using mathematical means. These and other abilities are based on a solid mathematical training. It includes discrete mathematics, formal logics, and an introduction to calculus and real analysis such that students are able to distinguish between ideal mathematical objects and their incarnations on current computer systems. They master not only formal methods to infer conclusions from facts but also the statistical methods to detect patterns in large data sets. Finally, graduates not only have a large repertoire of means and methods at their disposal; they are also aware of the limits of algorithmic and formal methods.*

Design and implementation abilities are described in a similar way. They include programming skills and the mastering of the software development techniques and tools. The technological competences specify the body of knowledge in computer hardware, architecture, operating systems, computer security etc. Beyond the subject specific competences, more general, social skills to be acquired by computer science graduates are described in a similar generic way: Graduates have learned to develop solutions in teams; they can communicate their solutions both in writing and orally; graduates of a Master's program have obtained an introduction into the scientific methodology of the discipline and are able to acquire new knowledge from scientific literature. They are able to organize and monitor projects.

It should be clear that such a description of learning outcomes does not necessarily lead to a unique curricular structure but may result in a large variety of different programs. The accreditation practice, however, has shown that the diversity is much less distinct than one might expect. Despite all local characteristics, undergraduate programs in informatics/computer science at German HEIs often look quite similar: About 30 ECTS credits are spent for mathematical training including discrete mathematics, calculus, formal logics and stochastic. There is an introduction into computer hardware and architecture, algorithms- and data structures, and computer systems including operating systems, networks, and database management. The strongest emphasis is laid on the introduction into programming and software development including project work, where students not only learn the basics of software technology but also gain practical experience in their application in small teams of 6 to 10 students.

There are several reasons why undergraduate programs often look quite similar: often, informatics departments would base the design of their new Bachelor's and Master's programs on the established one-cycle degree programs instead of starting from scratch; also, they would consult the recommendations of scientific and professional organizations like ACM/IEEE and GI for designing Bachelor's and Master's programs; and accreditation is based on a peer review of programs which means that experts from other HEIs and from industry discuss the concept and its realization with the members of the informatics department. Diversity is much more visible in the Master's programs. Here, the traditional German view, comprised in the traditional one-cycle informatics diploma, of granting a unique universal degree enabling their degree holders for a large variety of different jobs, has been replaced by a variety of highly specialized Master's programs ranging from media informatics, bioinformatics, technical informatics, security systems, to software technology, and many more.

Informatics/computer science is a subject offered at all three major categories of HEI: universities, universities of applied sciences (UAS), and universities of cooperative studies (BA). All HEIs grant Bachelor's degrees, at least in some states like Baden-Württemberg. Universities and UAS also grant Master's degrees. Formally, all degrees are considered to grant equal rights. This is new in the German system of HE. Before, only the Diploma degrees granted by universities qualified their degree holders to enter a PhD program. Now it is possible to switch between institutions with a Bachelor's degree in order to enter a Master's program and to enter a PhD program with a Master's degree regardless of where it has been acquired. It should be obvious that this has been a massive gain in prestige for the UAS. This change was politically intended and is one reason why UAS were much faster than universities in adapting the new two-cycle Bachelor's and Master's system and in seeking accreditation of their degree programs. The only difference remaining between universities and UAS is that the latter ones cannot grant PhD degrees. Nevertheless, the accreditation process has the effect that the educational standards of CS education at universities and UAS are becoming similar, though both still try to maintain their specific profiles: Universities concentrate more on the methodological and theoretical foundations of the discipline while UAS see their strength in the more practical aspects of the discipline and their close relationship with industry.

2. Challenges to applying the Bologna reforms at the University of Freiburg

The University of Freiburg (officially called Albert-Ludwigs-Universität Freiburg named after its founder, Duke Albert Ludwig) is a classical German research university already established 550 years ago in 1457. Today it belongs to the top tier of German universities with proven excellence in research and teaching. It is structured into 11 faculties: Theology, Law, Economics, Medicine, Philosophy, Philology, Mathematics and Physics, Chemistry, Biology, Forestry, and Applied Sciences. It offers 151 different subjects ranging from large subjects like German, to exotic and small subjects like European Ethnology. The structure and organization of this university poses several challenges to the implementation of reforms in higher education.

Heterogeneous program structure

More than half of the degree programs and almost half of the 20,000 students of the university are studying programs not affected by the Bologna reform. These are the state controlled educational programs in medicine, law, and all subjects qualifying students for becoming high-school teachers. Diploma degrees have traditionally been granted in all science and engineering disciplines, but they have to be replaced by the new programs. Thus, there is a discrepancy between the two educational systems, the state-controlled programs and the ones under the sole responsibility of the university. A large number of subjects (like mathematics, for example) may be studied in both modes leading to different degrees. It should be clear that the university tries to utilize synergies between the two programs wherever possible by mutually using identical courses in both programs. However, it turns out to be quite difficult to synchronize the different modes; transition and examination regulations vary, and the objectives of the degree programs differ.

Because of these structural differences, the change of degree programs to the two-cycle system of Bachelor's and Master's degrees at a full university with a broad spectrum of subjects is by far more complex than the corresponding change at a UAS or even at a more homogeneous technical university. Therefore, it is not surprising that most faculty members and study deans were quite hesitant implementing the change.

Heterogeneous information systems

What also has not been considered was the very heterogeneous software environment at the university. The switch to the bachelor/master systems created major headaches for the technical and administrative departments; In particular, they were not flexible enough to handle the large number of course-related exams in the Bachelor's and Master's programs instead of the few punctual exams in the diploma programs. Furthermore, the new examination regulations contain a lot more differentiations and regulations. Those are, for example, the specification of compulsory and elective subjects, keeping track of the numbers of required exams, the number of attempts as well as the introduction of a bonus and malus system. Additionally to the requirement to enter all this information in software systems there is also the need to be able to view the study progress for every student at any time. Therefore, new web-services for supporting the examination authority and the students alike had to be introduced. They should allow every student to retrieve his study progress at any time. First calculations in 2005 showed that the additional effort for manually administrating all the new exams for the students at the University of Freiburg amounted to additional work for six people employed full time for one year (counting 5 minutes of additional work per exam) [9]. Therefore, the software environment had to be prepared to the new system allowing a decentralized maintenance of all the students' data. The University of Freiburg addressed this problem with an infrastructure of a centralized service allowing decentralized administration within the university. Since the beginning of the transition to the new system, the software systems have been improved step by step, but they are still far away from a perfect solution. For example, most of the basic information in the course manual needed for the accreditation (see section three) as well as the data about the staff members and their research experience and current research fields are all stored in one of the universities software systems. From a technical point of view, it should be no problem to collect all this required data and to bundle it for the accreditation. Unfortunately, due to the heterogeneous nature of those software systems, they are not all interconnected which makes it quite hard to get the required data.

The "unknown" first cycle graduates

The common structural framework set in place by the state ministers for higher education stipulates that Bachelor's programs must lead to a professionally relevant qualification. Access to Master's programs must be based on further criteria, i.e. only those students having passed the Bachelor's exam with a result above average are allowed to enter a Master's program. However, where CS is concerned we know that students holding a Bachelor's degree have no difficulties to find a job, at least in the current economical situation. Furthermore, the assumption that the good students stay on and continue their studies in the Master's program is also not universally true. Most of our own graduates entered the industry workforce or changed to another university. Some of them returned after a short period in the industry to continue their studies in a Master's program.

"Soft skills" as explicit element of higher education

In order to qualify the students enrolled in a Bachelor's program for a job, not only a thorough education in the subject is necessary but also the acquisition of so-called "soft skills". These include the ability to work in teams, scientific reading and writing, presentation competences and the mastering of the appropriate computer-based tools, etc. This is also quite new for a classical university. Freiburg has solved this problem as follows: It has established a central unit organizing and teaching a broad spectrum of courses in four different categories: management, communication, media and computer usage, and foreign language education. Each bachelor program must contain between 8 and 12 ECTS credit points taken from this list in order to assure that the students obtain at least a basic training in these soft skills. Beyond this "external" training, informatics students are usually required to complete seminars and project work within their subject also providing such "soft skills" as a by-product.

Every informatics/computer science graduate should be able to solve real-world problems outside his own discipline. For that purpose, he should have learned to communicate with experts from other disciplines and apply his knowledge appropriately. In Freiburg, as in most other universities, this has been solved in such a way that informatics students are requested to enroll in a number of courses from some other discipline, the so called application area. A traditional university has much to offer in this respect: In principle, students may choose any subject of their interest ranging from medicine to business administration and micro system technology. In practice, it has, however, turned out that this was very difficult to organize: it requires special agreements with other faculties, a coordination of the class schedules and examination rules, a compensation for the teaching loads, and much more. Because the Informatics/Computer Science Department was among the very first departments who changed their degree programs to the new two-cycle system, it turned out to be almost impossible to import a limited number of modules from other programs into the informatics program.

General reservation against accreditation

For faculty members bearing in mind the Humboldtian ideal of a university teacher, combining excellence in research with excellence in teaching, the transformation of traditional research-oriented *Diploma* degree programs into a two-cycle structure and the requirement of subjecting the new programs to regular reviews “from the outside”, is a radical change to which they find it difficult to adapt. At the University of Freiburg there are a total of 151 programs (in which a total of 20.714 students[3] is currently enrolled). Only 10 of these have already been accredited: four programs in the Computer Science and Microsystems department (see next chapter), one in medicine (Master Online Periodontics), two in economics (MBA International Taxation, MBA Estate Planning), a global studies program (MA Social Sciences) and the programs Environmental Governance and Forest, Ecology and Management. For all of these programs, the higher education ministry of the state of Baden-Wuerttemberg (BW) required the accreditation either before the programs started or after a certain time. Also, all of them receive direct funding from the state ministry for higher education, giving the ministry more leverage than in cases where the funding is provided indirectly via the university as an institution.

As we will see in the next sections the accreditation process is quite time-consuming and sometimes difficult, but certainly worth it for some of the benefits gained by a peer review of the programs.

3. Design and Accreditation of Computer Science programs

Degree programs in informatics/computer science are offered by the Faculty of Applied Sciences, which consists of two departments, the Department of Computer Science and the Department of Microsystems Technology. The Faculty of Applied Sciences in Freiburg already decided in 2004 to suspend the informatics one-cycle Diploma degree program and to introduce new undergraduate and graduate programs. At the time, there was almost no experience at the university with the new two-cycle educational system and accreditation of degree programs. There was, as yet, no accepted framework for new Bachelor’s and Master’s programs. The aim of the department was to guarantee that the consecutive combination of the Bachelor’s and the Master’s program should lead to a qualification at least comparable to the old diploma studies. This is in concordance with the definition of so called consecutive Master’s programs: The general rules of the Standing Conference of the Ministers of Education and Cultural Affairs of the 16 states in the Federal Republic of Germany for the introduction and accreditation of Bachelor’s and Master’s programs [7] distinguish between consecutive, non-consecutive, and continuing educational master programs. A consecutive Master’s program extends, deepens, and continues its preceding Bachelor’s program. Thus, both the undergraduate and the graduate program are considered as a combined unit of coordinated modules leading to a clearly specified common profile. Therefore the idea, when introducing the new programs at the department, was to replace the diploma by a consecutive undergraduate and graduate program in informatics.

The design of the new programs and the accreditation procedure was quite a new experience for the faculty. In particular, the concentration on learning outcomes for the design and implementation of the new programs was completely unfamiliar to the staff. Two

[3] In winter semester 2007/2008 20.714 students were matriculated.

main questions had to be answered: How do we achieve the professional qualification of our bachelor students and what are typical jobs for a bachelor graduate? Hence, what is the desired learning outcome? Of course the general requirements, procedural principles and subject-specific criteria mentioned in section one of the accreditation agency had to be fulfilled for a successful accreditation of the new programs as well. This implied the collection and preparation of a lot of information. The output orientation forced explicit formulation of the objectives of the educational programs and not only to derive the curriculum from the objectives but also to show the contribution for each module to the successful achievement of the desired learning outcome.

Another challenge was the requirement that the Bachelor’s degree must enable its holders to successfully apply for a job outside the HEI. This meant that the traditional concentration of engineering and science studies at universities on a thorough mathematical training in the first two years could not be maintained anymore. Instead, in the undergraduate program the practical parts, inclusive the procurement of “soft skills”, were considerably strengthened, and the mathematical components and advanced topics were only marginally and exemplarily included. Parts of them had to be shifted to the Master’s program. The latter fact was also the source of a conflict which seems to be typical for universities transforming their diploma studies into new undergraduate and graduate programs: Traditionally, in the diploma degree programs at universities each core subject (in Germany usually represented by a chair) was represented by at least one advanced course for third and fourth year students; usually there is no specified order in which these advanced core courses have to be taken. The successful completion of these courses is the prerequisite for any further specialization in specific related subjects. The question now was which of the core and special courses should be included into the three years of the undergraduate program and which into the graduate program. In Freiburg (as in many other university departments) this conflict could not be solved amicably. It required a discussion of the staff with the peers during the accreditation audit in order to find a solution and to mostly eliminate this problem, leaving a marginal rest: Courses on software technology and on database systems were turned from electives to compulsory; only very few modules became eligible both in the undergraduate and graduate programs.

When designing the new graduate program, the faculty also wanted to attract students from other universities already holding a Bachelor’s degree. Students from abroad already enrolled in an international Master’s program (applied computer science, ACS, funded by the German Academic Exchange Service, DAAD, and offered by the faculty for several years) were of particular interest. Originally, it was the aim to merge this international ACS program and the new Master’s program. During the accreditation procedure it became clear that this merger was not compatible with the requirements for consecutive degree programs. Therefore, the faculty decided to split the Master’s programs into two variants: The consecutive “standard” program and a new non-consecutive ACS program. The latter has special conditions for enrolling students holding Bachelor’s degrees from institutions offering degree programs having some overlap with the Bachelor’s program in Freiburg but which are not almost identical. In order to allow for the previously gained knowledge of these “external” students and to adapt the level of qualification, the new ACS program was endowed with two new bridge modules providing students with the necessary knowledge enabling them to enter the advanced master courses. That is, both the standard consecutive graduate program and the new non-consecutive ACS-program share large parts of their curriculum but are not identical.

All the modules in the curriculum had to be described within a course manual which consists of a detailed description of each course: The ECTS credits awarded for this course, a detailed list of the workload (time to spend on the practical parts, theoretical parts, lab sessions, homework etc.) the language and type (major, minor, elective course, etc.), the role of the module in the curriculum (i.e. one of four elective modules in a certain area), the requirements, goals and learning contents as well as the kind of exam of the module and of course literature accompanying the course. The most difficult part here was that staff members were not used to distinguish between learning outcomes and the description of the learning input, that is, of the content of a module. Quite often, a professor specified the competences to be acquired by a module simply by "the goal of the course is to obtain an introduction to ... (and then the content description was repeated)". For all modules the learning outcome had to be specified in terms of knowledge (the ability to recall or remember facts without necessarily understanding them), comprehension (the ability to understand and interpret information), application (the ability to put ideas and concepts to work in solving problems), analysis (the ability to break information in its components to see interrelationships), synthesis (the ability to use creativity to compose and design something original) and evaluation (the ability to judge the value of information based on established criteria) [6]. Moreover, it was necessary to eliminate overlaps and to close gaps in the contents of modules. This, of course, first means that a professor offering a course is himself conscious about the learning outcome and the appropriate method to reach it.

The workload specified in the course manual has to match the total amount of time a student spends on the different modules while he is enrolled in a course. To get a Master's degree in Baden-Württemberg, students have to take courses for a total of 300 ECTS credit points (including the credits gained during their first degree – i.e. bachelor). This corresponds to a total number of 9,000 hours of student work load (1 ECTS credit point awarded corresponds to 30h work). This workload and its related burden for the student (which is often underestimated by students) have to be monitored continuously. In this way the student is not overworked, but the workload should also not be too low during a semester. The required amount of work and other factors like the quality of teaching should be evaluated at least every semester to make sure that the quality fits the expectations one would have from a top tier university. This includes student's critics of courses. In the Master's program "Intelligent Embedded Microsystems" (IEMS) we evaluate courses twice a semester (once in the middle to be able to intervene if something goes wrong, and once after the exams to get accurate/detailed feedback from the students about the real workload they had with the modules including the time they spent on preparing for the exam).
The major problem here is that every faculty or department used to have its own style of using evaluations to implement some kind of quality monitoring. Today the University of Freiburg is establishing a general evaluation framework for the whole university in order to assure a continuous and standardized evaluation of all courses in every faculty. Quality control is also a requirement for the accreditation of the degree programs and therefore there is no need that every faculty starts from scratch. But the standardization of all different faculties needs for evaluation is almost impossible. What fits the Computer Science Department needs does not fit the needs of the Psychology Department and vice versa. A cautious legal department caused additional problems by rejecting the drafts for an evaluation regulation because they allegedly were conflicting with legal provisions.

Since Master's programs should have a clearly visible research orientation, curriculum design cannot be independent from the research activities of the staff members responsible for the program. Therefore, a meaningful detailed description of the professional experience, the areas of research, and role of all the people involved in the program had also been compiled for accreditation. It is not surprising that this was one of the easier tasks, for professors of a research university which are accustomed to present themselves and their research achievements.

Of course, also the basic formalities for any degree program, like admission regulations, regulations for exams, form of the diploma degree, and the diploma supplement had to be formulated and approved by the university committees. Creating these regulations without the existence of a framework for Master's/Bachelor's degree programs at the university was arduous but not impossible. Today the University of Freiburg has a general framework for all Bachelor's and masters' programs; it is based on the experience gained in the establishment of the informatics/computer science programs and considerably facilitates the introduction of new subject specific programs.

Currently, the Faculty of Applied Science with its two departments, the Department of Computer Science and the Department of Microsystems Technology, offers a bachelor program in informatics and two master programs (consecutive and non-consecutive) in informatics as well as bachelor and master programs in microsystems technology. Both departments have a special research focus on intelligent systems and in the field of embedded microsystems. They have all the technologies and know-how available necessary to design and develop modern high-tech embedded systems. Therefore the faculty decided to also offer a new graduate program in this field as a program for continuing education, the Master's program "Intelligent Embedded Microsystems" (IEMS). The idea for this new program was not only to combine the expertise of the two departments but also to utilize the long experience of the faculty in the usage of networked computers and multimedia for establishing online versions of their study courses. As already mentioned, the Bachelor's degree from German universities of applied sciences (UAS) and universities of cooperative studies (BAs) now gives students the possibility to enter a Master's program at a university. In order to broaden the reservoir of potential graduates for the new Master's program, and, in particular, in order to attract students who have already worked for some time in industry after their graduation, the curriculum of the master IEMS has been designed to accept all kinds of different students from these institutions [1]. The establishment of this new Master's program IEMS has been supported by a generous fund from the State Foundation BW. This funding, however, was combined with the requirement to accredit the degree program before its introduction.

Fortunately, the accreditation of the new Master's program IEMS was a lot easier. The complete documentation describing the resources available for implementing the program, like staff, buildings, equipment etc. could be reused. So the biggest hurdles were mainly the concept of the degree program, the mode of its realization, the detailed description of all modules, and the regulations for admission and transfer from other programs.

After delivery of the whole documentation to the accreditation agency, a group of four peers, all specialists in the related fields, carefully studied the material provided. They then visited the department in order to discuss the new program, its goals and implementation as well as the studying conditions for the students with the professors involved in the program.

During the first audit of the computer science bachelor program it was criticized that the given ECTS credits did not reflect the correct workload of the students and that the qualification profile of the bachelor students was too blurry. The audit of the master IEMS was similar: The peers detected a number of inconsistencies, overlaps, and gaps in the program and helped to sharpen its profile. Moreover, there were some singularities involved in this IEMS program requiring special attention: First, the degree program is designed as a Master's program for continuing education. This means, that its modules and the mode of instruction should relate to the acquired professional experience of students. Second, it should be possible to study the program part-time and enable the students to stay in their job with a possibly reduced workload. Third, the study mode is blended learning, which is a mixture of online delivery and present studies. Many modules are based on newly compiled lecture recordings and enriched with other study material like self-assessments and a study guide. The special delivery mode for the content required us to show the lecture recording rooms where the recordings are taken and the laboratories where students can do practical lab sessions. The peers were quite impressed by the professionalism on how lecture recordings [3] are produced at the faculty and how they are delivered via a learning management system. Even some labs are organized as online labs [2], something quite unusual for such degree programs. As a result of the peer review and the audit, accreditation was made pending on only very few improvements: Overlapping contents in two courses had to be removed, in some others the contents had to be clarified, two new modules had to be introduced (control theory, actuators), and some of the detailed module descriptions had to be revised. For all modules, a responsible contact person had to be nominated, and detailed module descriptions had to be created for each project management module instead of a general description of project management modules.

The result of the whole accreditation processes was a considerable improvement and a much better coherence of the whole program. Thus, the faculty has many reasons to appreciate the careful and thorough work done by the accreditation agency and the peers.

4. System accreditation versus program accreditation

The introduction of system accreditation as an alternative to program accreditation is a recent development. The German Accreditation Council has developed system accreditation as a new instrument of external quality assurance for higher education degree programs in 2007 in order to address several problems that had been identified by HEIs and several state ministries for higher education.

On the one hand, it was found that program accreditation as practiced since 2000 was a costly process, both in terms of financial and personal resources: These encompass external costs for fees charged by accreditation agencies as well as internal costs associated with preparing the required documentation. The fee charged for an accreditation process for a single degree program or a consecutive Bachelor-Master-combination is about EUR 11,000 to 12,000, when more related programs are reviewed within one joint procedure, the average cost per program is about EUR 3,500 to 4,000. Additional costs of the accreditation process were caused by the compilation of the self evaluation and additional documentation: since almost no German HEI had pre-defined processes or dedicated administrative staff for quality assurance, and heterogeneous information systems did not provide coherent data on student success, it was most often left to academic staff to compile the required documentation by their own hand. Not surprisingly, this was not seen as an efficient use of scarce resources. All the more, because on the other hand program accreditation was not always perceived to have a visible and lasting impact on the quality of the degree programs. All too often, it seemed, the peer review was limited to discussing formalities rather than discussing ways to further improve the quality of the degree programs; when recommendations for quality development were made, HEIs lacked proper instruments for ensuring that they were being followed up upon. From this perspective, and given limitations to the institutions' capacities, program accreditation was criticized as a futile exercise without lasting impact.

To address these problems, system accreditation has been introduced – initially as an alternative to program accreditation but with the perspective of replacing it altogether within five to ten years' time. System accreditation requires HEIs to install a comprehensive quality management system that – at least – covers their core process teaching and studying. In order to receive a system accreditation, HEIs must demonstrate that not only have they designed and installed a comprehensive quality management system in this area, but also its effectiveness, i. e. the QM system must be able to effectively control and guarantee the quality of all degree programs on an ongoing basis. This includes the effective control of teaching quality based on student surveys, the periodic review of program outcomes, the transparent documentation of the programs and of all modules including their contribution to achieving the desired learning outcomes, the efficient organization of the course of studies including the design, administration and registration of exams, research on alumni and their career development, to name a few. Quality management systems suitable for system accreditation must meet the Standards and Guidelines for Quality Assurance in the European Higher Education Area [8], and should be as effective as the external quality control achieved by the program accreditation – in effect, this implies that HEIs assume the functions of program accreditation rather than leaving this task to an external agency. HEIs that have been awarded a system accreditation are freed from the requirement of having all degree programs accredited individually.

Given the organizational capacities and current state of quality management instruments and processes at most German HEIs (as described above using the – not untypical – example of the University of Freiburg), the road to system accreditation will be a long and winding one for many of them: Universities like the University of Freiburg have just started to establish such a quality control system. They are already routinely collecting quite a large number of benchmark data relevant for quality control. But there are still serious deficiencies and fundamental problems resulting from the fact that many of the involved services and their computer-based support is not organized in an interoperable way. Here, universities will have to invest a lot of energy and money within the near future.

Nevertheless, at least the large research universities in Germany seem to aim for system accreditation instead of program accreditation in the medium term. They also consider this a further indication of autonomy and independence of external control by the state or state-governed institutions. Furthermore, organizing their own quality control system facilitates the combination of teaching and learning assessments with research assessments, also to be periodically carried out every five to seven years.

5. Lessons learned

The biggest advantage of the accreditation process of the new undergraduate and graduate programs is that faculty members are obliged to seriously discuss the teaching and learning objectives in their departments; the competence profile expected from their alumni by their future employers and not their own (research) interests dictates the design and implementation of a degree program. All our experiences show that external peer reviews are the most efficient means to initiate the necessary change from the input to output orientation and to guarantee the quality of degree programs. The requirement to document the objectives, the learning outcomes, and the contents of a curriculum in the accreditation procedure implies that accredited degree programs are usually much better documented than non-accredited ones. This is of special benefit for students, future employers, and staff members alike. The additional expenditure for the accreditation appears to be worth the effort; it is assuaged if the university offers well-developed computer-based services for staff and students supporting all phases of studying at a HEI. Currently, we see the greatest deficits in this respect. Though universities have a large number of web-based services supporting students and staff, these services are seldom interoperable, well integrated, reliable, secure, and user friendly enough in order to fulfill current and future needs. If quality control by accrediting a degree program is not considered an isolated event to be carried out every five or seven years but understood as a continuous process, universities must be able to provide benchmark data a mouse click away. Information about success and failure rates of degree programs, acceptance of their alumni by the job market, workload of modules and results of course-related exams must be easily available.

The emphasis on learning outcomes and competences to be acquired in the new degree programs designed after the Bologna reform also raises new challenging questions for faculty and scientists alike: How can we measure the competences? What are the best means to achieve the desired learning outcomes? Are our courses, labs, seminars the appropriate, the only means to achieve a defined competence level? Are there other ways to reach a competence level comparable to what universities can offer? How can the university contribute to the life-long learning process of continuously refreshing the competence of their alumni? The modularization of the new degree programs should facilitate this quite a bit: Universities may allow interested students to enroll in single courses which are of current interest for industry irrespective of whether the applicant for participation has an appropriate degree. Once he can show that he has the necessary competence to follow the course, he could be allowed to enroll. If he succeeds, the university may issue a certificate but not (another) degree to him.

Finally, though most large (research) universities currently go for systems accreditation instead of program accreditation, we see that most of them have still to go a long way until a reliable and effective internal quality control system is operational. We do not expect that its successful implementation will make procedures like the current program accreditation obsolete. It is expected that they may become easier to handle and less cumbersome; but we anticipate that their essentials remain valid in order to guarantee the quality of current and future degree programs.

References

1 Hermann C, Welte M. Continuing Education at Universities: New Perspectives at German Universities, Informatics Education Europe II, Thessaloniki, Greece, Nov. 2007.

2 Becker M., Hermann C., Welte M. and Manoli Y. "intelligent embedded microsystems" distance learning in microsystem engineering and applied computer science. In EWME 2008 - 7th European Workshop on Microelectronics Education, to appear. Springer.

3 Hermann C., Hürst W. and Welte M. The electure-portal: an advanced archive for lecture recordings. Informatics Education Europe, Montpellier, France, Oct. 2006.

4 Computing Curricula 2001 Computer Science, ACM/IEEE 2001, http://www.sigcse.org/cc2001/cc2001.pdf

5 Bachelor- und Masterprogramm im Studienfach Informatik an Hochschulen, Dezember 2005, GI Empfehlungen 48, 2005 http://www.gi-ev.de/fileadmin/redaktion/empfehlungen/GI-Empfehlung_BaMa2005.pdf

6 Bloom, B.S. Taxonomy of educational objectives: The classification of educational goals: Handbook I, cognitive domain. New York, 1969

7 Ländergemeinsame Strukturvorgaben gemäß § 9 Abs. 2 HRG für die Akkreditierung von Bachelor- und Masterstudiengängen; Beschluss der Kultusministerkonferenz; 10.10.2003 as amended on 22.09.2005 http://www.kmk.org/hschule/strukvorgaben.pdf

8 Standards and Guidelines for Quality Assurance in the European Higher Education Area; European Association for Quality Assurance in Higher Education; Helsinki, Finland; 2005 http://www.bologna-bergen2005.no/Docs/00-Main_doc/050221_ENQA_report.pdf

9 Kraus M., Trahasch S., Vögele E. and Hermann C. eGovernment-Dienste als Voraussetzung für den Bologna-Prozess. In Proceedings of Multikonferenz Wirtschaftsinformatik, Passau, February 2006. Springer.

Comparison of the Hungarian and Euro-Inf accreditation systems

László Aszalós

Faculty of Informatics, University of Debrecen, Hungary, Laszlo.Aszalos@inf.unideb.hu

The programme Software Engineering First Cycle Degree at the University of Debrecen was founded in 2004 according to Hungarian Higher Education Law and Bologna Process and accredited by the Hungarian Accreditation Committee. In 2008 it was also accredited by the Euro-Inf organization. This paper will present the details of the Hungarian accreditation system and how it has evolved. We will describe the Euro-Inf accreditation system which was established to provide comparison between higher education qualifications in the field of informatics. This paper will present the similarities and differences between the two accreditations system, which have different origins yet have the same goals.

Keywords

Accreditation, Bergen Conference, Bologna Process, Euro-Inf, Hungarian Accreditation Committee

1. Introduction

Before 1989 the Hungarian higher education and academic system followed the Russian model. Many researchers received their degrees in the Soviet Union. After 1990 Hungary began to adopt the standards of the European Union. The foundation of the Hungarian Accreditation Committee [3] was part of this process through the first higher educational law. It was created as a professional advisory board to the ministry, responsible for higher education. The first aim of this organization was the founding of the uniform PhD education, and its quality control. Among its duties are evaluating the establishment of new higher educational programmes and the reviewing the applications of universities/colleges to start these accredited programmes. The organization evaluates the intent for founding new institutes, universities, and accredits the universities colleges and PhD schools. The Hungarian Accreditation Committee follows the Bergen Document [6], and its standards. The recommendations of the Hungarian Accreditation Committee are in no way mandatory for the ministry, but yes they are followed through.

The European Union having been formed from very different countries, there is a big variety in higher educational programmes, even in informatics. For an employer it is almost impossible to compare two different programmes. One of the main aims of the Bologna Process [5] is to make these programmes comparable. The Euro-Inf organization [2] undertakes providing a common framework for the European accreditation system for the engineering and informatics sector. All this could facilitate the mobility of graduated students because the employer can trust in the “European label” given by Euro-Inf.

2. Hungarian accreditation system

Before 1989 the ratio of graduates was significantly smaller than in the European Union. Therefore there was a political motivation to increase the number of graduates and to reject the school fee. The Hungarian state paid a predetermined amount for each regular student to the institutes. To attract more and more students the institutes started new programmes. Presently there are 25 universities and 47 colleges in Hungary with a population of 10 million. The new higher education law introduced an authorized minimum headcount to stop this tendency, and fixed the list of first cycle degree programmes [4]. This list was created in accordance with the universities and colleges after 3 years of preparation and accreditation process. At the moment there is no opportunity to establish a new BA/BSc programme that is not on the list.

2.1 Establishment of a programme

There is no limit in the higher educational law about establishing second cycle degree programmes. The sole limit is economic; the Hungarian government limits the number of regular students who can enrol in MA/MSc programmes, which is 30 percent of the number of BA/BSc students. If an institute wants to establish a second cycle degree programme, it needs to present an application to the Hungarian Accreditation Committee. This application contains:

- description of the subject of the programme
- the number of required ECTS credits to fulfil the programme
- classification of credits: how many are from required subjects, how many are from required elective subjects and how many are from free-elective subjects
- scientific fields
- the programme outcome, general and specific competencies
- system of exams

Based on the application the Hungarian Accreditation Committee can recommend accepting or rejecting the establishment of the programme. After a positive decision of the ministry the institute has the right to apply for establishing program.

2.2 Launching a programme

The Higher Education Act 2005 [1] and the Government Decree 79/2006 regulate the launching of a new programme in Hungary. According to these, any institute can launch any established programme, if it can prove it has a suitable staff, infrastructure, and a demand for graduates.

Staff requirements

The institution needs to name a senior instructor with scientific, i.e. PhD, degree who is employed full time at the institution. This person needs to be responsible for only one degree program at a time. For any subject a person with the scientific degree is needed, who is responsible for that subject. The research area of the person responsible for the subject must cover this subject. One person can be responsible up to 25 ECTS credits in any programme of any institute.

Content requirement

The curriculum of the programme should satisfy the “National Qualification and Outcome Requirements” [4]. These requirements are published regularly on the homepage of the Ministry of Education and Culture. The first cycle degree programme needs to be suitable for entry into a second cycle degree programme; and the second cycle degree programme needs to be suitable for conducting research and development and doctoral studies.

Research

The institution needs be involved in one research or development project in which it employs a nationally recognized research team. The teaching staffs needs to publish on a regular basis and present their research results in appropriate forums of science/engineering.

Infrastructure requirements

The institution provides the basic infrastructure necessary for their programmes on an ongoing basis

- premises corresponding to the actual number of students for students, teaching and support staff corresponding to the needs of the programme
- training tools for the entire cycle
- organizational and administrative structure supporting education

All the programmes need a library where the major periodicals of the given discipline are available or electronically accessible. The library's textbook holdings have to include books listed as suggestive reading in the subject syllabus. The programme needs an IT network offering state-of-the-art services accessible to students on a regular basis and in an organized manner as well as research, design, measurement and other facilities and equipment needed for preparing graduation thesis and projects; infrastructure and training site for practical education.

Special requirements

- Bachelor programmes in the technical and natural sciences should always include courses in computer sciences, general economics and management, quality assurance, environmental studies and European studies
- Programs in the technical sciences require a minimum of 40 credits of basic education in the natural sciences and minimum 40 credits of distinct professional core material in the area of study.

Requirements for distance learning

There is very long list of requirements for distance learning. Here some of them:

- One tutor may teach no more than 50 students in more than 3 subjects at a time (in one semester).
- Conditions for the ongoing updating of the course material.

Content of the application

The application needs to present the previous teaching and research activities of the institute. The aspirant institute need to give a forecast of the demand of graduates in the region and in the country proving the launching of the programme is really necessary. The application contains the curriculum, and for all subjects the syllabi and the list of recommended literature. The institution ensures the personnel and material condition for the number of students in line with the needs of the discipline. The institute needs to present the methods of learning and the development of skills and competencies described in the outcome requirements. Finally it must support the successful completion of a foreign language exam, which is obligatory for issuance of a diploma. To be able to launch the distance learning, all the teaching materials need to be included in the application.

The application must contain a staff handbook which includes the teachers' CV. The following data will be included: name; qualification; skill; working place; scope of activities; degree; title of thesis; scientific scholarships; teaching activities; professional practise; publications over the last five years and the most important ones; international relations in research/teaching.

The application must contain the following appendix: the recommendation of the senate of the university, the outcome requirements and the opinion of the potential employers.

2.3 Institute accreditation

An institute accreditation is valid for maximum eight years. Purpose of these accreditations is to establish if an institute satisfies the minimal requirements and the management ensures the teaching and research activities required.

The institutes are converting to the Bologna Process, thus the accreditation contains the monitoring of this conversion. An institute cannot be accredited without a working quality assurance system. The goal of the quality assurance is:

- perfect satisfaction demand of “costumers”, rising complacence
- rising level of education
- ensuring effective performance

The management needs to determine the functional tasks, long-term goals and plans, the monitoring, evaluating, and modifying of short-term goals, rules, regular evaluation; quality assurance organizations, the parameters for changes and their outcomes. The aim of the self-assessment report of the institute is to sum up the performance and to explore problems. The main points of the self assessment report are

- quality policy, strategy and quality assurance procedures;
- launching and follow-up programmes, regular internal evaluation;
- student’s evaluation, complacency of students;
- ensuring quality of teachers (minimum level of publication, taking part in applications, organization of conferences, developing content of subjects);
- students services, tools;
- internal informational system (most important economical, scientific results of the institute, result of doctoral schools and gifted students);
- publicity

3. Euro-Inf accreditation system

This part is based on documents at [2]. The ultimate goal of the Euro-Inf Project is to facilitate European-wide professional recognition by the competent national authorities of informatics degrees. These recognitions are awarded by study programmes accredited on the basis of the programme outcomes and accreditation criteria defined in the Euro-Inf Framework Standards. The partners of Euro-Inf Project are ASIIN (Accreditation Agency for Degree Programmes in Engineering, Informatics, the Natural Sciences and Mathematics, Germany), Council of European Professional Informatics Societies (37 informatics societies from 32 countries), University of Applied Sciences Hamburg and University of Paderborn. The Euro-Inf, based on standards and accreditation systems, created a set of framework standards which were tested and refined through trial accreditations. The Faculty of Informatics of the University of Debrecen, who applied for the trial accreditation for its programme of Software Engineering, was a partner in a trial accreditation in March of 2008.

3.1 The aims of Euro-Inf

Euro-Inf aims to create a framework for setting up a European system of Standards for assessing informatics education at the First Cycle and Second Cycle level (as defined within the Bologna process). Based on the establishment and approval of this set of standards, the main objectives of the Euro-Inf Project are:

- to provide an appropriate “European label” for accredited educational in informatics
- to provide a basis for comparing educational qualifications in informatics in the European Higher Education Area (EHEA)
- to facilitate mutual transnational recognition by programme validation and certification
- to facilitate recognition of accredited degrees in informatics higher education in accordance with the EU Directives and other agreements
- together with other field-specific standards and criteria, to contribute to the harmonisation of the European Higher Education Area
- to support the mobility of informatics graduates
- to contribute to international transparency - as one of the objectives of the Bologna Declaration
- to support improvements to the quality of informatics programmes in general

3.2 Content of the application

The Euro-Inf standard asks the applicants to prepare self-assessment report; entry standard and requirements; curriculum outline; module handbook (learning outcomes, syllabus content, etc.); titles of final year projects for the previous three years; list of facilities including buildings, laboratories and equipment; staff handbook (CV’s of academic, technical and support staff); provide for the accreditation visit the set of representative examination papers.

The structure of the self-assessment report is the following

- the programme *needs, objectives and outcomes*: needs of the stakeholders (students, potential employers, informatics societies), educational objectives (consistency with the mission of the institute, publicity, standards), programme outcomes (the practice is consistent with the requirements)
- relevant and effective *educational processes*: planning (quality of curriculum), delivery (analysis of students’ and tutors’ evaluation, workload statistics), learning assessment (transparency and publicity of rules,)
- appropriate *resources and partnerships*: academic and support staff (competence and qualification of teaching staff, research and consulting work, statistics about support staff), facilities (lecture, computer facilities, library), financial resources (budget for staff, for running and upgrading facilities, for training), partnership (local, regional, national, international partnership and cooperation agreements)
- adequate assessment of the *educational process*: students (entrance requirements, success rates, etc.), graduates (numbers, match between employment and education, graduates’ and employers’ opinion)
- an effective *management system*: organization and decision-making processes, quality assurance system (policy, procedures, system of evaluation)

3.2 Meetings at accreditation visit

The accreditation visit takes at least two days with the auditing team having a preliminary meeting where they discuss the documentation and the necessary information to be obtained. The team meets with the head of department/university who introduces the institute the academic staff members and support staff. The staffs need to prove the facts in the self-assessment report are valid and to answer any questions the auditing team might have. After this, the team meets with students and with former students, with relevant employers/industry/professional informatics organisations representatives. The aims of these meetings also serve in checking the validity of the report, and learning more about institute. The accreditation visit includes an evaluation of relevant facilities (libraries, laboratories, etc.) and a review of project work, final papers and other assessed work (with regards to the standard and modes of assessment as well as to the learning achievements of the students). At the end of the visit the auditing team gives feedback to the institute including the preliminary result.

3. Discussion

Although the two different accreditation systems have the same origin in that they are based on the same standard, they have very different roles and aims.

The requirements of the Hungarian Accreditation Committee are sometimes too general and it is hard to audit them. For example the competences of the programme in Software Engineering is a 12 item long list, where a sample item is *planning, analyzing and developing algorithms considering the most important paradigms of programming.*

The Euro-Inf system contains a 34 item long list of competences, where one of the corresponding items is *ability to select relevant analytic and modelling methods,* which is more concrete, and easier to check.

The knowledge of foreign language in Hungary is under the average level of the European Union. Hence foreign language exams are necessary to get a diploma and a remarkable portion of students graduate several years after their final exam, because they didn’t pass the language exam. This is the reason why the support of foreign language learning is so important in the Hungarian accreditation system. In some cases the applicants need to give data about command of a language of the teaching staff. The Euro-Inf accreditation doesn’t require similar data.

The Euro-Inf was developed according to European Qualification Framework (EQF). The core of the EQF is its *eight reference levels* describing what a learner knows, understands and is able to do – their 'learning outcomes' – regardless of whether a particular qualification was acquired. In the module handbook the applicants need to label these levels for all the subjects. While the EQF is a hot topic on the conferences about education in Hungary, the accreditation system has not been adopted yet.

In the Hungarian accreditation system does not emphasize the direct connections with the industry. The staff handbook of Euro-Inf accreditation directly asks about the cooperation with industry over the past five year, and the patents and protected rights. Since the Faculty of Informatics, Debrecen originates from Institute Mathematics, most of the teaching staff is a researcher, and the members of the faculty only a few patents and very light connections with the industry.

In the Hungarian accreditation process the programme accreditation is based exclusively on the written proposal, not on the on-site visit. This is common during the accreditation of institutes or universities/colleges.

4. Conclusions

The regular accreditation is determined by the higher education law in Hungary. It helps for the students and potential employers, to compare universities and colleges, but annoying for our colleagues. It also helps for the heads of institutes to see that everything is going well. Carefully analyzing the Hungarian accreditation system shows that it is an almost state-of-the-art system. The Euro-Inf accreditation is a good opportunity to measure our result in international manner.

References

1 Higher Educational Act (in Hungarian) http://www.om.hu/letolt/felsoo/ftv_20051129.pdf
2 Homepage of the Euro-Inf organization http://www.euro-inf.eu/
3 Homepage of the Hungarian Accreditation Committee http://www.mab.hu/english/index.html
4 National Qualification and Outcome Requirements (in Hungarian) http://www.okm.gov.hu/download.php?ctag=download&docID=650
5 The Bologna Process: Towards the European Higher Education Area http://ec.europa.eu/education/policies/educ/bologna/bologna_en.html
6 The European Higher Education Area - Achieving the Goals http://www.bologna-bergen2005.no/Docs/00-Main_doc/050520_Bergen_Communique.pdf
7 The European Qualifications Framework (EQF) for Lifelong Learning http://eacea.ec.europa.eu/llp/eqf/index_en.htm 2007

GRASP: Grading and Rating ASsistant Professor

Maura Cerioli[1], *Pierpaolo Cinelli*[2]

[1]*DISI – Dipartimento di Informatica e Scienze dell'Informazione, Via Dodecaneso 35, Genova, Italy, cerioli@disi.unige.it*

[2]*DISI – Dipartimento di Informatica e Scienze dell'Informazione, Via Dodecaneso 35, Genova, Italy, cpier83@gmail.com*

GRASP is a simple tool for assessing and grading libraries in the .NET® framework. It has been developed to support the management of the project for a third year course and has been used to grade the student projects for two years. The main functionality offered by GRASP is the execution of an arbitrary number of test sets, grading the result on the basis of a configurable mapping between test results and scores. The code of the tests depends only on the interfaces given as assignment, and the connection between interfaces and student implementation is managed by GRASP. The tool is built as an open architecture, where additional filters can easily be plugged in.

Keywords
Automatic grading of code, CS Curricula, Programming in the .NET® framework, Project assessment

1. Introduction

Since the early times of teaching programming techniques, many researchers have actively pursued the automatic assessment of code (see e.g., [1], [2], [3], [4], and [5] for further references). But, recently the needs for such tools have become more pressing.

Indeed, the number of projects to be assessed in the average class is dramatically increasing, following the number of students in computer science and related areas, which, in turn, is gaining as the world population is continuously expanding and the educated percentage of it is improving. Moreover, due to economic shortages in many educational institutions, the number of teachers is not keeping up the pace with that of student, sometimes it is even decreasing. So there are proportionally less teachers for each student. Moreover, the amount of code to be assessed is increasing also because of the diminished skills and motivations of the average student. Indeed, nowadays several young people join computer science and engineering programs only in order to be able to get a job at the end of their studies, disregarding their personal interests and, in most cases, their previous educational career. Thus, they are missing the basis and the intrinsic motivations to learn programming. Therefore, such students on one hand are more prone to submit insufficient projects, implicitly increasing the number of assignments to be assessed, and on the other hand need more feedback to be able to learn programming skill, also in terms of corrections of preliminary exercises.

Another reason in favour of automated assessments is that e-learning and blended teaching strategies are spreading worldwide and need to empower their end-users by efficient self-evaluation tools, providing quick feedbacks at any time. Thus, it is imperative for such distance-learning environment to include automatic evaluations. In most generic environments for distance learning, the automatic evaluation is limited to very simple assignments, like for instance quizzes and multiple-choice tests. But, in order to support programming courses, it is necessary to provide better feedback on the code elaborated by the students, including automatic testing.

Though several automated assessment tools have recently been developed to address the automatic testing of code, most of them are limited to the case of *filter* programs[1]. This is for instance the case of [6], [7], [8] and its successor [9], and [10]. But, currently a large part of programming teaching is devoted to object-oriented languages and techniques, which are becoming the *de facto* standard programming paradigm. Most recent tools slightly extend their applicative domain to object-oriented code, but still testing only input-output behaviour on elementary types.

In order to assess functional correctness of object-oriented programs, we need more flexible testing, featuring assertions and verifications on multiple objects and their status. The tool we know that is closest to satisfy this requirement is JUnit [11]. However, its didactic use has a few limitations. First of all, JUnit is a tool for unit testing, not for education. Thus, on one hand it is coarse grained, in the sense that, as usual in a standard testing activity the only interesting output of a test is whether the code passes or fails the test, possibly collecting some information about failures to help the following fixing phase. For grading purposes, instead, the output of the tests needs to be a level of compliance to the specifications in a sufficiently subtle scale. We may need, for instance, to assign a smaller penalty to the wrong choice of exception in a test expecting an exception to be raised than to some code terminating without signalling anything abnormal, possibly distinguishing different paths of executions and so on. The second problem of using JUnit to assess didactic projects is that it supports the evaluation part, but fails to have grading features. Thus, the educational user needs another (inexistent as far as we know) tool to translate the testing report in a mark. Finally, automated testing tools are designed to test projects where both classes and implementations are available during the test development, so that the test can refer to them. Instead, in order to totally separate the student implementation from the project assignment and tests, produced by the teacher, we need a specific feature allowing writing tests without any knowledge of the implementations, not even at the syntax level.

Our interest for automated assessment tools for object-oriented libraries has arisen teaching a programming course for the third year of the degree in computer science at the University of Genova, called TAP (Tecniche Avanzate di Programmazione, i.e. Advanced Programming Techniques). Such course has around 60 students per year and gives the basis of component based programming and design to the students. It has a major project aimed at having the students familiarize themselves with the .NET® technologies in general, and C# programming in particular, and at improving their programming abilities, by individually realizing a library implementing a set of fixed interfaces.

The particular setting of the TAP project prevented us from using already existing automated assessment tools. Indeed, the development platform is .NET® and the language is C# and we did not find any tool at all supporting them[2]. Thus, we decided to develop a small tool, called GRASP (Grading and Rating ASsistant Professor), to automate the project grading.

The basic idea is to grade the projects only on the basis of (mainly functional) correctness, by testing them. To make this possible, the project specification must be extremely precise, so that the behaviour of the implementations is totally predictable. Moreover, it is mandatory that the assignment cannot be tampered with, that is, that the implementation of the students

[1] A filter program is typically composed by routines, taking in input basic values, and producing basic values as output.

[2] Microsoft Visual Studio Team System edition provides an integrated testing environment, but it has all the limitation expressed for the JUnit testing tool.

is totally disjoint from the official text of the project. Therefore, GRASP is designed to rate projects implementing a set of interfaces, constituting its specification (at a syntactic level).

The tool runs test sets on all the project implementations included in a given folder and produces for each of them a short report stating the assigned mark and a more detailed one with the results of each test.

Though the tool is written in C#, it can be used to grade projects realized in any language supported by the .NET® framework, because of the support for mixing up libraries written in different languages provided by the Microsoft platform. Actually it is even possible to write the tests in a different language from that used by the students to implement the project.

In Section 2 we describe GRASP features, while Section 3 is devoted to a brief sketch of its architecture. Possible improvements of GRASP are discussed in the last Section.

2. GRASP Main Functionalities and Usage

Roughly speaking, the requirement of GRASP is to support the automatic evaluation of any number of projects realizing a component, by way of the execution of test sets.

A library of classes, each one representing an individual test case, constitutes a test set. Each such class has a special method, with signature `void ExecuteTest()`, and executing the test corresponds to create a new object of the class and invoke that method on it. In the method body it is possible to set the result of the test, choosing a value among `Unacceptable`, `Error_Serious`, `Error_Medium`, `Error_Light`, `Error_Paltry`, `OK`, `Correct_Good`, `Correct_Optimum`, and `Correct_Excellent` in the enumeration `TestResult`. Each test is aware of the execution environment and capable to access part of the context information. Moreover, it is possible to create test cases that cooperate with each other. More details on the definition of tests are discussed in Subsection 2.4.

2.1 Supported Projects

GRASP has been developed in order to support the projects of the TAP course, which is focused on component based development. Thus, those projects consists of the implementation of a component, realized by a .NET® assembly.

Since there are no immediate linguistic means to represent components[3] in the .NET® framework, we use the object factory pattern (see e.g., [12]) to simulate the (dynamic) connection between the specification (i.e. the component contract) and the implementation (the binary code[4]) of a component.

Thus, the assignment consists of

- a collection of interfaces in an object-oriented language (usually C#[5]), describing the syntax of the component interface;
- for each interface and each method in it, the specification of its semantics, presented by the Microsoft Xml Documentation providing at the same time also the standard documentation of the interface;

[3] The `IComponent` interface and its standard implementation `Component` in the .NET® framework represent a very special model of components, providing features to be managed both at design and at run time. A generic component in the sense of component based development is usually mapped on an assembly. However, the full separation between interface and implementation is not enforced by the framework and has to be imposed by methodological means.

[4] We are loosely using “binary code” to refer to MSIL code as well.

[5] It could be any language supported by the .NET® Framework; but, for TAP we use C#.

- a collection of *factories*, i.e. classes with virtual methods creating objects of the given interfaces, playing the role of constructors for the classes implementing the interfaces.

For instance, let us consider as a running example the specification of teams, as part of a project for the development of a component to be used in systems supporting the management of tournaments. Then, the interface to be implemented could be the following, where `IPlayer` is an interface for the definition of players, which we omit, being irrelevant for understanding the example.

```
/// <summary>
/// Represents a generic team.
/// </summary>
public interface ITeam
{
     /// <summary>
     /// Returns the name of this team.
     /// </summary>
     /// <remarks>This field is constant.
     /// </remarks>
     string Name
     {
         get;
     }

     /// <summary>
     /// Adds a new player to the team.
     /// </summary>
     /// <param name="player">The player to add.</param>
     /// <exception cref="ArgumentNullException">Exception thrown when
     /// <paramref name="player"/> is null.</exception>
     /// <exception cref="InvalidOperationException">Exception thrown
     /// when <paramref name="player"/> is already in team.</exception>
     void AddPlayer(IPlayer player);

     /// <summary>
     /// Removes a player from the team.
     /// </summary>
     /// <param name="player">The player to be removed.</param>
     /// <exception cref="ArgumentNullException">Exception thrown when
     /// <paramref name="player"/> is null.</exception>
     /// <exception cref="InvalidOperationException">Exception thrown
     /// when <paramref name="player"/> is not in team.</exception>
     void RemovePlayer(IPlayer player);

     /// <summary>
     /// Returns a numeric value indicating how many players compose the
     /// team.
     /// </summary>
     int PlayersCount
     {
         get;
     }
```

```
        /// <summary>
        /// Returns the list of all the players composing the team.
        /// </summary>
        /// <remarks>The list must be a safe readonly wrapper.</remarks>
        IList<IPlayer> GetPlayers();
    }
```

Then, the factory for such component could be the following, where we detail only the constructor for the empty team and drop the other constructors, both for this interface and the others.

```
    /// <summary>
    /// Provides the API for creating and using the tournament via a
    /// remoting context such as the GRASP one.
    /// </summary>
    [Grasp.Definitions.FactoryDefinition]
    public class TeamFactory : MarshalByRefObject
    {
        /// <summary>
        /// Creates a new team with no players.
        /// </summary>
        /// <param name="name">The name of the team.</param>
        /// <exception cref="ArgumentOutOfRangeException">Exception thrown
        /// if <b>name</b> does not belong to ANB[3,25].</exception>
        public virtual ITeam CreateTeam(string name)
        {
            return null[6];
        }
       ...
    }
```

In this constructor code, a requirement is imposed on the format of the acceptable team names: team names must contain only letters, digits or blanks and have length between 3 and 25. This kind of requirement is quite common and cannot be efficiently tested by a small number of test cases.

Note that the class `TeamFactory` is recognized by GRASP as a factory, because of the attribute `Grasp.Definitions.FactoryDefinition`. The tool will explore the student projects[7] looking for classes inheriting from it and having the attribute `Grasp.Definitions.FactoryImplementation` in order to use them during the test to produce the objects of the interfaces to be tested. In this way, the test can be written without a reference to the student implementations, which are hooked at run time by this mechanism.

[6] The factory class produced by the teacher is actually an abstract class, never to be instantiated. Thus its factory methods will never be invoked, do not carry any information and can safely return null. The student will be required to inherit from this class and override all the virtual methods, inserting the actual code building the objects.

[7] Actually, GRASP will take into account only those assemblies that the student has notified the system to contain factory classes, by adding to the evaluation directory files by the same name of these assemblies and with an extra `factory` extension. This is required both for individual multi-assembly projects, and for cross testing, in case a student produces a component depending on the implementations of other students, for instance if a large assignment is distributed among the members of a group. Thus, the tool will correctly manage the merging.

The assembly (or assemblies) containing interfaces and factories constitutes the assignment from the point of view of GRASP and has to be distributed to the students, to be referenced in their projects. In the sequel we will refer to such assembly as `ProjectAssignment.dll`.

2.2 Evaluating One Project

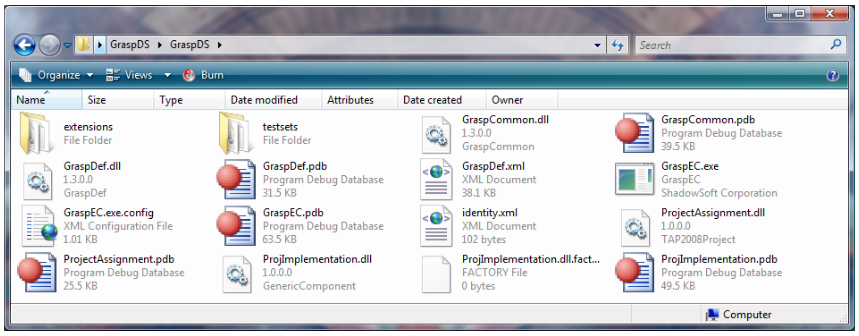
In any configuration of the system, the evaluation of student projects is ultimately delegated to the `GraspEC` application, for Grasp Execution Context. It executes all tests included in the `testsets` folder on the student project, producing as output an xml file, with a node for each test. Such a node contains the following attributes.

- The full *name* of the test, including the test set (hierarchical) name, if any.
- The test *scale*, i.e. its difficulty. It may be any integer value in the range [0,100], is declared in the definition of the test (see Subsection 2.4), and is used as multiplicative factor when computing the final mark as weighted sum (see next subsection).
- The test *result*, that is the numeric code associated to the value from the enumeration `TestResult` chosen as result in that execution.
- The *stage* of the test code where the result has been established (see Subsection 2.4 for further details). This information is mostly useful for debugging; both for using GRASP to debug the project under evaluation, for instance by the students during development, and for debugging the tests themselves, clarifying the execution path.
- The *exception*, if any is raised, disregarding the result of the test. Indeed, notice that raising an exception may be the correct behaviour of the project.

In the next Subsection we will see how the xml result file is used to grade the project.

In order to evaluate a project, the easiest way is to locally execute `GraspEC`. In that case, the application should be executed in a directory with a structure similar to that in Figure 1[8].

Besides `GraspEC.exe` and its configuration file, the folder should contain the assignment and its dependencies, (`ProjectAssignment.***`, `GraspCommon.***`, `GraspDef.***`), the project implementation to be tested (`ProjImplementation.***`), any file with the same name as the assembly containing the factory implementation but with extension `factory` (`ProjImplementation.dll.factory`), used to direct GRASP in the search for the factories pointing out the correct assemblies. Moreover, two special folders are needed: `extensions`, including the libraries extending the functionalities of GRASP, and `testsets`, containing the libraries of tests to be executed.



extensions File Folder	testsets File Folder	GraspCommon.dll 1.3.0.0 GraspCommon	GraspCommon.pdb Program Debug Database 39.5 KB
GraspDef.dll 1.3.0.0 GraspDef	GraspDef.pdb Program Debug Database 31.5 KB	GraspDef.xml XML Document 38.1 KB	GraspEC.exe GraspEC ShadowSoft Corporation
GraspEC.exe.config XML Configuration File 1.01 KB	GraspEC.pdb Program Debug Database 63.5 KB	identity.xml XML Document 102 bytes	ProjectAssignment.dll 1.0.0.0 TAP2008Project
ProjectAssignment.pdb Program Debug Database 25.5 KB	ProjImplementation.dll 1.0.0.0 GenericComponent	ProjImplementation.dll.fact... FACTORY File 0 bytes	ProjImplementation.pdb Program Debug Database 49.5 KB

Figure 1 The Structure of GRASP Folder.

In order to be evaluated, student projects must comply with two rules.

First of all, they must implement the assignment, `ProjectAssignment.dll`, that is, they need to have a reference to that library[9] and provide implementations for all the required

[8] The GRASP suite is fully customizable. Here, we are showing the default configuration.

interfaces described there. The student projects may also include a number of extra classes and methods, of any visibility level.

The second aspect is that the factory classes have to be marked by the `Grasp.Definitions.FactoryImplementation` and must give a meaningful implementation of the factory methods, using in their bodies the constructors of the appropriate classes.

For instance, a student project for our running example should include in the same directory an assembly `ProjectImplementation.dll` containing the following definition (where `MyTeam` is the implementation of `ITeam`) and a file `ProjectImplementation.dll.factory`.

```
[Grasp.Definitions.FactoryImplementation]
public class MyTeamFactory: TeamFactory
{
    public override ITeam CreateTeam(string name);
    {
        return new MyTeam(name);
    }
   …
}
```

Since the student projects need to use the `Grasp.Definitions.FactoryImplementation` attribute, they also have to reference `GraspDef`, where the attribute is defined.

All the configuration definitions for `ProjectImplementation`, if any, should be added to GRASPEC.config in order to be available during test execution.

Using `GraspEC` locally and on a single project, however, is not a solution to the problem of evaluating a large number of student elaborates. In the next Subsection we will see how it is possible to grade several projects at the same time.

2.3 Grading Projects

In order to perform the evaluation of several projects at the same time, a service, called `GraspAgent`, is provided. It is totally automatic and it processes *packages*, taking them from the `source` directory and placing the resulting evaluation and grading reports in a subdirectory of `result` created with a unique name. At the end of a successful evaluation, the input package is deleted from `source`. Instead, if the evaluation cannot be performed, the package is marked as faulty and left in the `source` directory for further consideration. In that case, the corresponding evaluation is meaningless and has to be ignored, but for debugging purposes.

There are several reasons for which the evaluation may abort.

The most obvious is the attempt at evaluating a package where some important element is missing, or corrupt. Indeed, each package must contain all the files needed for the evaluation, i.e. the project implementation assemblies and possibly the configuration file, any needed resource, like for instance databases or other files, and the file `identity.xml`, whose root has the attributes `name`, `surname` and `id`; the latter is the student id, used to disambiguate in case of homonymy[10].

[9] GRASP automatically resolves versioning conflicts, if any, by forcing the student project to use those assemblies specified both in the default environment and in the teacher's package. This measure helps to prevent malicious students from changing the specification. See Section 3.1 for further details.

[10] This file is used to identify the student who produced the package. The teacher may decide to specify validation rules for each field and in that case, a non-conforming identity file will cause the package rejection without any further action.

Another common cause of failure is the mismatch between the version of the project assignment used to compile the submitted project implementation and that given to `GraspAgent` to build the evaluation environment.

Indeed, in order to prevent student tampering with the testing environment, the evaluation service build a brand new directory like that described in Subsection 2.2 using its own copy of the *standard* parts (that is, GRASP, the assignment and the tests) and taking from the student package only the project implementation assemblies, local resources and configuration, forcing the redirection of references and resources to those provided by teacher, even if in a different version with respect to those used by the student.

Other likely troubles that could prevent `GraspAgent` from completing the evaluation are timing problems. For instance, extremely inefficient project may take so long in completing some test to fail. Indeed, GRASP uses a configurable time limit for the execution of each test case so that looping programs can be interrupted and the revealing test marked as failed.

The evaluation of a package produces two reports; the first one is the detailed evaluation described in the previous Subsection, while the second one is the actual grading of the project, which is an xml file containing the student data from the `identity.xml` file and the computed mark, besides information about active extensions on the grading algorithm.

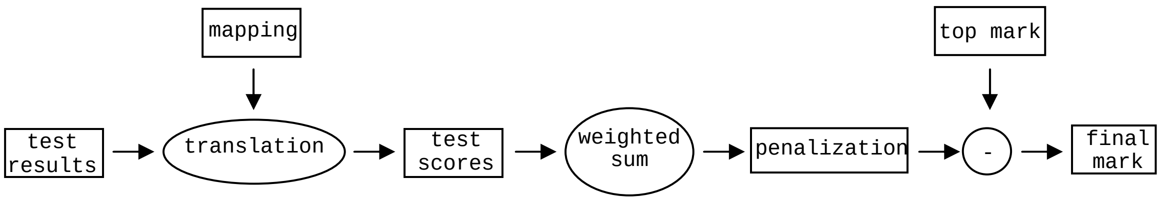


Figure 2 The Basic Grading Algorithm.

The basic grading algorithm is depicted in Figure 2. First the logical results of the individual tests are translated into numerical values, following a map defined in the resource `mapping`. Using logical values to define the test results to be mapped on scores only at the moment of the final grading not only allows the teacher to fix the numeric details having already seen the student results, making guesses unnecessary to get a reasonable mark distribution, but it also helps students to focus on the relevance of the individual test without being carried out by the numeric score, especially during the debugging phase.

The second step is to multiply the numeric score of a test for its scale and algebraically sum all of the resulting values to get the weighted sum, representing the penalty earned by the project, to be subtracted from the top score. Since a higher final mark is determined by a smaller penalty, each error will contribute a positive part to the penalty, while each excellent performance will be rewarded with a negative value. Notice that a particularly brilliant student could score a grade higher than the top mark, in case the penalty is negative.

The algorithm described so far only applies if all the results are acceptable. If there is at least one `unacceptable` result, the system always computes a non-positive result to signal that the project is unacceptable. However, there may be cases where the teacher wants to consider all the projects as passed (possibly with a very small score). For instance, if the students are given a subset of the tests to be used as a threshold, any package passing all of them should be considered sufficient, even if in some extra test it scores an `unacceptable` result. In order to take care of these situations, an extension of the basic algorithm allows to change all the scores for instance from `unacceptable` to `error_serious` so that the algorithm computes a correct mark in any case. It suffices to add the following lines to the configuration file. The first line activates the extension and the second one provides that extension with (a bit of) the actual mapping.

```
<add key="extension_ChangeResult" value ="none, GraspFactory,
Grasp.Factory.Extensions.ChangeResultChannelPlumb"/>
```

```
<add key ="service_changeresult_unacceptable" value ="error_serious"/>
```

It is also possible to change more result values, for instance, in order to preserve fairness of evaluation distinguishing the real `error_serious` from those obtained from former `unacceptable`. To add further changes it suffices to add the corresponding lines, like for instance

```
<add key ="service_changeresult_ error_serious " value ="Error_Medium"/>
```

This extension is but an example of the flexibility of GRASP. Other tailoring tasks can be similarly implemented as a plug-in.

2.4 Writing Tests

The most time consuming activity of project evaluation with the support of GRASP is the definition and implementation of tests.

A test set is simply any assembly of the .NET® framework using the namespace `Grasp.Definitions` provided by the `GraspDef` assembly and containing classes inheriting from the class `TestCase`. In order to be executed by GRASP, the individual test cases must be public classes and marked by the `TestCase` attribute. They need to have a public constructor with an integer parameter representing the difficulty scale of the test, used by the grading algorithm. The most important method of a test case class is `Execute`, which is automatically invoked during the evaluation phase. In that method it is possible to write any kind of (correct) code, using, in particular, the `ExecutionContext` property with its methods and properties, of the `TestCase` class. The `ExecutionContext` is the core of the testing system. It provides the following methods

- `GetFactory(Type)` produces a new object of the class implementing the factory type in the student project. Thus, it is the entry point in the code to be tested. Since its implementation do not require a reference to the code to be tested, not even the name of the student factory class, using this method we achieve total independence between testing and student implementations.
- `SetTestResult(TestResult)` set the result of the test and terminates the execution of the test. It may also have an extra parameter of type `Exception`, in which case the exception is added to the result report.
- `EnterStage(int)`, which mark the beginning of a new logical stage of execution; if the result of the test is established at stage `n`, then `n` is written in the report, helping the user to locate the execution flow.
- `MapException(Type, TestResult)` maps the raising of an exception of the given type to the assignment of the result to the test; it is a convenient way to avoid disseminating the test of try-catch blocks, making the test code more readable.
- `UnMapException(Type)` vice versa ends the redirection of the exception.

For instance, an elementary test case for the example of the team creation will be the following

```
/// <summary>
/// Creates a new team and checks that there are no participants.
/// </summary>
[Grasp.Definitions.TestCase]
public class CreateTeam : TestCase
{
    public CreateTeam()
```

```
            : base(10)
        { }

        public override void ExecuteTest()
        {
         TeamFactory f =
         ((TeamFactory)ExecutionContext.GetFactory(typeof(TeamFactory)));
            ITeam t = f.CreateTeam("Team 1");
             ExecutionContext.EnterStage(10);
            if (t.PlayersCount != 0)
            {
                ExecutionContext.EnterStage(20);
                ExecutionContext.SetTestResult(TestResult.Unacceptable);
                return;
            }
            if (t.GetPlayers() == null)
            {
                ExecutionContext.SetTestResult(TestResult.Error_Serious);
                return;
            }
            else
            {
                if (t.GetPlayers().Count != 0)
                {
                    ExecutionContext.EnterStage(30);
                    ExecutionContext.SetTestResult(TestResult.Unacceptable);
                    return;
                }
                else
                {
                    ExecutionContext.SetTestResult(TestResult.OK);
                    return;
                }
            }
        }
    }
```

Each test is executed independently[11], so that a priori there is no interaction among objects created in different tests. Since it is convenient to reuse the object created (and checked to be correct) in previous test cases, the `ExecutionContext` also provide the property `GlobalContext`, which is a simple dictionary where such objects can be permanently stored in order to be reused. For instance, if in the previous example we add the line

```
    ExecutionContext.GlobalStore.Add("team1", t);
```

just before setting the test result in the correct case, then we can reuse it in further tests, for instance those checking the functionalities to add and remove players.

[11] In the standard configuration, GRASP will execute all the tests in a test set in the same process sequentially. But, it can be configured to reset the environment for each test set.

2.5 Using GRASP for Debugging

From a technical point of view, it is very easy to use GRASP and its test sets for debugging the project implementation during its development. Indeed, it suffices to distribute the skeleton of the GRASP directory discussed in subsection 2.2, complete with the assemblies of the tests to be published before project submission, and set `GraspEC.exe` as external program to be started during debugging. In this way it is possible to execute the tests and even put breakpoints both in the implementation and in the tests, following the execution step by step.

Moreover, giving the students at least part of the test sets to evaluate their projects helps them to understand how systematic testing is performed, improves their understanding of the specifications and save them the time of choosing and implementing the tests themselves.

However, letting the students have the test sets (or part of them) also has some drawbacks. Indeed, some students are driven to think that passing those tests is the unique required achievement of their code. They spend their time tampering with their code so that it passes a specific test, without taking into account the damages that unplanned modifications are doing to their code integrity, nor understanding the general troubles identified by the specific failure. Thus, totally absorbed by solving a few specific cases where their code fails, they are actually wasting time in decreasing the global code quality. For instance, if they have a method accepting as parameter only alphanumeric strings (like the constructor in our running example) and are given tests where the actual parameter includes three forbidden chars, the less brilliant students check exactly for those with a cascade of `if` statements, instead of reading the original specification and modify the code to check that all the elements of the string are letters or digits. In this way, they waste time and get in the end unreadable code, which will fail any new test using some different unacceptable char.

This negative didactic effect may be avoided, or at least reduced, by discussing with the students how the testing plan should be derived from the specifications, the huge number of exhaustive testing (if ever possible to achieve) and how it is reasonable to cut down the size by choosing representative examples.

3. GRASP Lifecycle and Architecture

Though it may be used locally, GRASP has been designed to run on a remote machine and interact with the users in a limited and simple way.

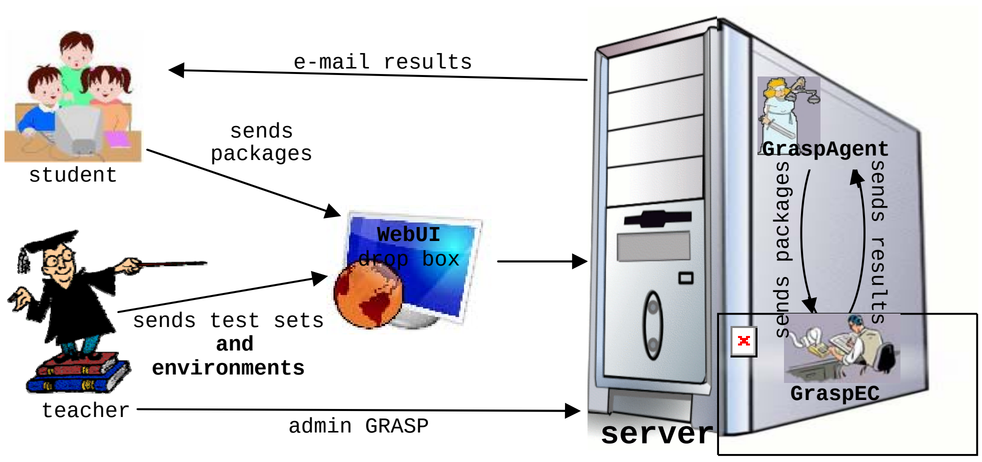


Figure 3 GRASP Standard Deployment.

In Figure 3 we see the standard distributed deployment for GRASP. The teacher (or an administrator) set up GRASP on the server. Afterwards, the interactions of the students with the running service are totally encapsulated by a minimal web interface letting the students drop their packages on the server to be processed by `GraspAgent` that will send the results by email to the address given at the submission moment. The user provided to the students has no other right on the server than writing in the drop-box. Thus, even a malicious user can only consume its disk quota, without damaging the system.

The teacher can use the web interface to add new test sets and environments to the service.

The set up of the service consists simply of copying the `GraspAgent` directory on the server, copying the data for the current project in the correct positions inside such directory and start the service.

The needed data for assessing a project are the following:

- The *project assignment*, i.e., the assemblies containing the definition of interfaces and factories, which must be packed in a zip file[12] and put into the environments directory.
- The *test sets*, i.e. the assemblies containing the chosen `TestCase` classes, must be copied into the `testsets` directory.
- The *mapping* file used to translate the test results into scores[13], which must be copied into the `mappings` directory.

3.1 GRASP Lifecycle

When starting up, the GRASP service first of all reads its own configuration and checks that all the needed extensions and resources are available. Then, it loads all the required extensions and simulates the execution of an internal evaluation task, to verify its feasibility. If no errors are encountered in these preliminary phases, then GRASP starts the scheduling of the tasks and executes them till it is stopped. When stopped, GRASP releases all the resources and terminates its execution. Though ideally the service should always be stopped at the end of the scheduled tasks, it is possible to end its execution at any moment and the system closes down gracefully. Indeed, if GRASP is stopped during the execution of a task, the task is immediately cancelled, its data are saved on the disk and it will be performed from scratch (not resumed) at the next start up of the service, so that no data can be lost.

The standard task executed by GRASP is the evaluation of a package. The main concern in this phase is to guarantee the security of the system as well as the correctness of the execution of a package. Thus, it is mandatory to be sure that all the tests to be executed coincide with those prepared by the teacher and that the student code is executed in a sandbox, with privileges as low as possible.

At this aim, the service builds an execution directory, playing the role of sandbox, starting from the data submitted by the student and injects in it all the standard files for the evaluation, in order to prevent any code tampering. Thus, the project assignment is taken from the default environment[14], the GRASP libraries from the root directory and the test sets from the `testsets` directory of the service. Before starting the execution of `GraspEC` in the built sandbox, the service may switch to a less privileged user, called *limited user*, and in any case strips down its own privileges so that only the part of file system inside the sandbox can be accessed. The choice of using the limited user and the indication of its login and password is done by specific keywords in the service configuration. The actual definition of

[12] The default value of this file is `default.zip`

[13] The default value of this file is `default.mapping`

[14] The default environment is the template for the minimal and common files that are needed for running a package.

such user in the system is left to the administrator, who should create it with the minimum amount of privileges to execute the code. However, due to the generality of the applications of our tool, the correct configuration may greatly vary from case to case. Hence, it is impossible to give indications in the general case.

Analogously to the overall service, also `GraspEC` starts its lifecycle by checking that its execution environment contains all the needed resources. In this case, the activity corresponds to verify that all the required factories and internal resources are available, because the overall service has already checked the correctness of the extension configuration.

If the verification does not fail, the test cases are executed in alphabetical order and their results are recorded in the final xml report. While the test cases are executed, the results are communicated via a secure channel to the GraspAgent service.

At the end of the execution of all the tests, the service grades the results and saves them in the `results` directory, possibly sends them by email to the submitting student, and finally destroys the sandbox.

If the execution of a package stops unexpectedly, like for instance if it encounters unexpected errors, e.g. some missing resource or an execution timing exceeding the allowed time limit, the task is immediately cancelled, no data is saved on the disk and the package will be marked as faulty. Faulty packages are scheduled for a fresh new execution.

3.2 GRASP Architecture

The development of GRASP has followed a component-based approach, so that its architecture is highly modularized and planned for further extensions. Most parts have been developed specifically for this project, though some are third part components, released under the GPL license.

The main subsystems and their relationships are depicted in Figure 4. As it is easy to see, the core of the system is the `Agent` component. Not only `Agent` coordinates the other components of the system, but it is also responsible for the management of the system lifecycle. Indeed, it takes care of configuring and initializing the different parts of the system and of the interactions with the execution runtime, with the collaboration of the specific components.

Roughly speaking, the other components can be distinguished in those used to configure and manage the overall service and those responsible for the execution of an individual task.

In the first group we have the component managing the configuration of the overall service (`AgentConfigurationManager`), those registering the common parts to be used for the evaluation, managing respectively the mapping files (`MappingRegistrationManager`) and the environments (`EnvironmentsRegistrationManager`), those managing the extensions both of the overall service and of the environment for the evaluation (`ExtensionLoader` and `ExtensionManager`) and finally the component managing the internal events (`EventManager`), and the scheduler (`ServiceScheduler`), taking responsibility for the timed execution of individual tasks. It is worth to notice that this latter component manages all the loaded active extensions of GRASP, i.e., all the extensions implementing the interface `ISchedulableAgentExtension`. Thus, in order to enrich GRASP by other evaluation means, it suffices to define suitable active extensions and to add their activation keyword to the configuration file. This architectural choice makes GRASP an open system, in the sense that it is quite easy and convenient to improve it with other functionalities.

In the second group we have the component building the correct configuration for `GraspEC` (`RuntimeConfigurator`), the component responsible for the creation and management of the sandbox (`ConfigurationBuilder`) and finally the component taking care of the management of the resources needed by test execution, from their creation to their final release (`ExecutionTracer`).

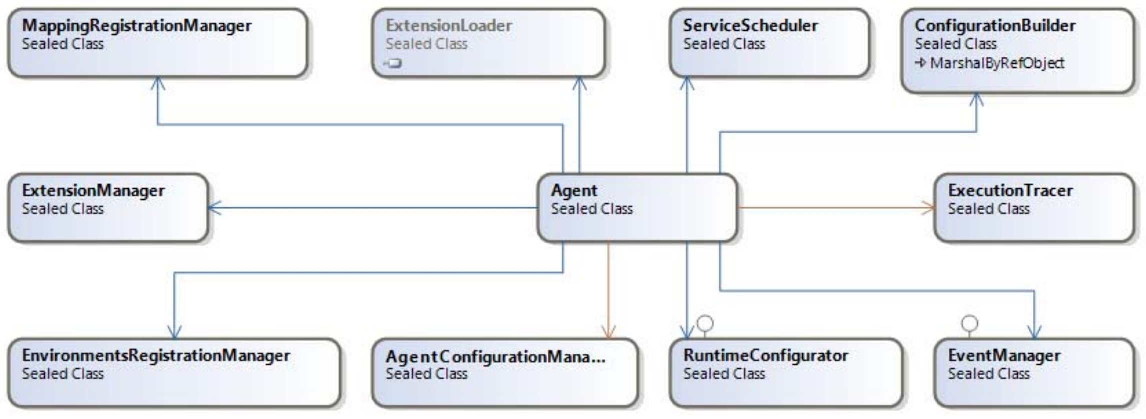


Figure 4 `GraspAgent` **Architecture.**

4. Conclusions and Further Work

We have presented an automated assessment and grading tool for student projects on the .NET® platform, for which there are no available alternatives, called GRASP. It supports a larger class of tests then simple input-output statements and allows grading the test results on an eight values scale, supporting fine-grained evaluations.

GRASP has been successfully applied for marking over a hundred student projects, implementing 8 different assignment. The constraints on teaching resources for TAP require an automatic evaluation solution and we are quite satisfied with GRASP performances and applicability. However, it would be interesting to investigate the relationship between automated (by testing) and human (by code inspection) evaluation. To this aim, we plan to reassess the grades of (at least part of) the projects graded by GRASP, to get an insight on the limits of our tool and possible improvements on the test design to make automated and human evaluations closer. Another interesting experiment we plan to conduct is to investigate the impact of the use of GRASP during the development of the student projects. Indeed, having some tests and an environment to perform them could influence the quality of the produced implementations. To verify if this hypothesis is true and, in that case, see if the influence is positive or negative, we plan to assign as next term project the very same library already used as project assignment a few years ago, before using GRASP and compare the student products in the two cases by the same tests. Since the students have the same background, the results should be significantly comparable.

In the future, we plan to improve GRASP from three points of view.

First of all, we will generalize it in order to support additional test systems, like for instance dependent or parallel testing, and will further extend the capability of customization of the suite. Moreover, we will provide an additional layer to encapsulate a database system, standardizing both the database schema, if any, and the database calls. Thus, it will be possible to evaluate conformance to requirements on the data layer as well.

Then, we will integrate other evaluation criteria, possibly by interoperating with static and dynamic analysis tools like fxcop[15]. This extension will enable the tool to check the conformity of the code to required patterns and to verify that good programming practices are respected.

Finally, we will address the management aspects, on one hand by tracing the progress of the development made by every student, allowing in this way to collect statistical information about how and when the students work and help detecting plagiarisms. On the other hand, we will integrate the GRASP suite with an e-learning platform like Moodle in order to improve the experience of the students, providing one uniform front-end for all their e-activity.

References

1 Hollingsworth J. Automatic graders for programming classes, Communications of the ACM 1960; 3(10): 528-529.

2 Naur P. Automatic grading of students' ALGOL programming. BIT 1964; 4: 177-188.

3 Forsythe, G. E., Wirth, N. Automatic grading programs. Communications of the ACM 1965; 8(5): 275-529.

4 Hext J. B. and Winings J. W. An automatic grading scheme for simple programming exercises. Communications of the ACM 1969; 12(5): 272-275.

5 Douce C., Livingstone D., and Orwell J. (2005). Automatic test-based assessment of programming: A review. J. Educ. Resour. Comput. September 2005; 5(3).

6 Blumenstein M.M., Green S., Nguyen A. T., Muthukkumarasamy V. GAME: a generic automated marking environment for programming assessment. In: Pradip K. S. et. al. editors. Proceedings ITCC 2004 International Conference on Information Technology: Coding and Computing; 2004; IEEE Computer Society.

7 Matloobi R., Blumenstein M.M., Green S. An enhanced generic automated marking environment: GAME-2. In: Auer M.E., Al-Zoubi A. editors. Proceedings of the International conference on interactive mobile and computer-aided learning. 2007.

8 Ceilidh on the WWW. URL: www.cs.nott.ac.uk/~ceilidh/.

9 Higgins C., Hegazy T., Symeonidis P., Tsintsifas A. The CourseMarker CBA system: improvements over Ceilidh. Education and information technologies September 2003; 8(3): 287 – 304.

10 Tremblay G., GuÈrin F., Pons A., Salah A. Oto a generic and extensible tool for marking programming assignments. Source Software—Practice & experience archive March 2008; 38(3): 307-333

11 JUnit resources for test driven development. URL: http://www.junit.org/.

12 Gamma E., Helm R., Johnson R., Vlissides J.M. Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional computing series. November 10, 1994.

[15] This tool is shipped with Microsoft Visual Studio Professional Edition and above.

Teacher Studies in Austria

Bridging the Gap Between Mathematics and Informatics Education

Karl Josef Fuchs

University of Salzburg
Department of Mathematics and Informatics Education
karl.fuchs@sbg.ac.at

The paper documents the necessary changings of an established basic lecture for teacher students at the University of Innsbruck caused by the intention to cover the requirements of Mathematics and Informatics teacher students with one conjoint course. The article gives a short overview of the goals of the lecture first. Afterwards the course schemes of the two teacher studies Mathematics and Informatics are confronted with each other. The main part of the paper is built by reflected and reasoned considerations derived from the comparison of the two curricula. Therewith the reader's interest should be aroused by the experience that basic principles and concepts of Mathematics and Informatics education hold similarily on one hand but that each discipline has its specific approach on the other hand.

1 Relation, Intention and Goal

In 2000 I was invited to hold the basic lecture in Mathematics Education at the University of Innsbruck the first time. Since then I have been acting as visiting professor in Mathematics Education continuously. In Winter 2007 the administration of the university decided to add Informatics to the offered teacher studies in natural sciences in Innsbruck. As the Bachelor and Master studies in Applied Informatics have already been running at the technical faculty for several years one can expect that some of these students would ask for courses in Informatics education by and by to gain an additional vocational pillar. In summer 2008 after due considerations I gave my agreement to rename the established basic lecture in 'Mathematics education' to an 'Introduction in Mathematics and Informatics Education' to satisfy the expected requests. The time quota for this lesson - two hours a week - was left unchanged. So far the lecture was customarily visited by twenty - five students approximately. Henceforward we may count on nearly thirty students as the number of teacher - students in Informatics will be at best one-fifth of those in Mathematics.
Now it is necessary to present the goal of the basic course. The lecture should feed the students into the concepts and models of scientific Mathematics and Informatics Education. The course ought to be visited in the sixth semester or later. Afterwards the students have to visit an educational seminar. The head of such a course expects the students to be able to work on selected publications in Mathematics or Informatics Education independently. Usually the students have to prepare a term paper which is presented to the head and other participants in one of the seminars' hours.

Practice in schools is covered by separate courses.
Diploma theses can be written as well in Mathematics as in Informatics Education.

2 Counterparts Teacher Studies Informatics and Mathematics

The teacher studies we are focussing in entitle graduates to teach in grammar schools (10 to 18 years old students) or in schools providing vocational education (15 to 19 years old students) [31]. A full teacher study is composed of two subjects where each subject needn't be within one faculty. For example: Combinations like Informatics / Physics or Mathematics / Informatics are as possible as Mathematics (Informatics) / History. Actually the minimum period for these studies is 9 semesters. Each subject in natural sciences covers approximately ninety hours a week educational parts and school practice included.

2.1 Course Scheme of the Combination Informatics / Mathematics

A roughly done review provides the reader with the information that both studies rest on three pillars:

- *Special branches of science*,
- *Educational concepts of each subject*,
- *Pedagogy and School Practice*.

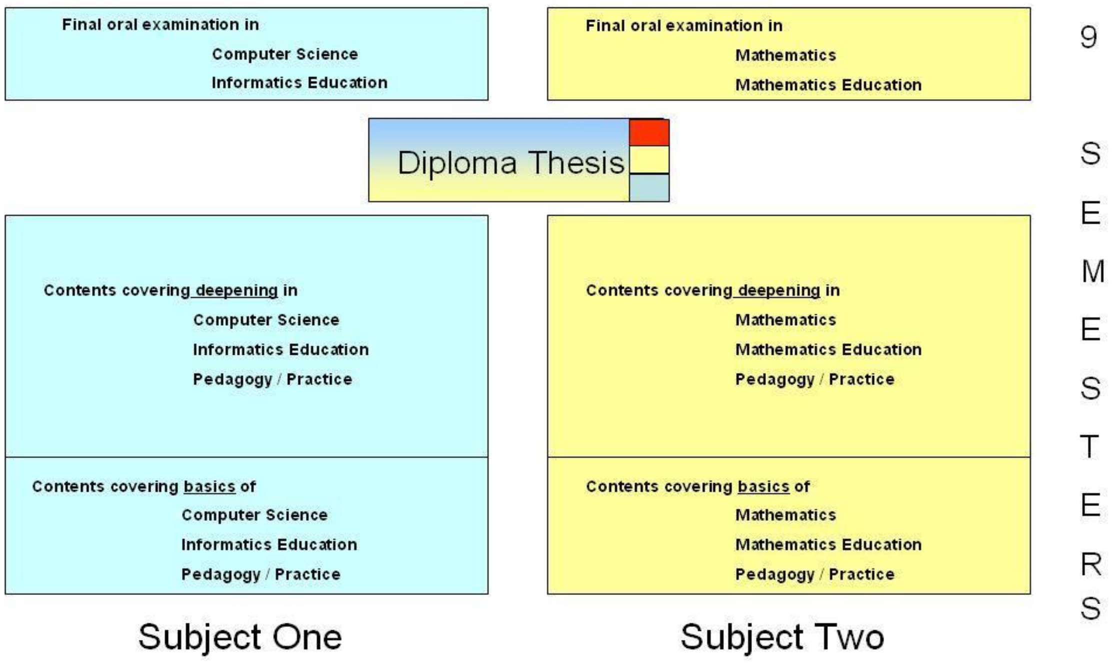


Figure 1: Teacher Studies Informatics / Mathematics

2.2 Contents and Methods

Special branches of science are covered with

- Introduction to Informatics / Algorithms and Data Structures / Computer Architecture/ Data Bases / WWW & Multimedia / Software Applications / Operating Systems and Computer Networks / Programming Methodology / Legal and social Aspects in INFORMATICS,
- Introduction to Mathematics / Analysis / Linear Algebra / Algebra / Discrete Mathematics / Geometry / Stochastics / Differential Equations / Numerical Mathematics in MATHEMATICS.

The *Educational concepts of each subject* are discussed in courses

- Introduction to Informatics Education / Teaching Methods / Programming in School / Applied Software in School / Operating Systems and Computer Networks in School in INFORMATICS,
- Introduction to Mathematics Education / Teaching Methods / Analysis, Algebra, Geometry and Stochastics in School / History of Mathematics and social aspects in MATHEMATICS.

Pedagogy and School Practice are integrated in the whole vocational training

- with courses in teachers' personality training (Basic Competences, Fundamentals of teaching and learning),
- with orientation- and self reflection - phases as a result of confrontation with *School Practice* ,
- with the so called semester of practice in school.

Now having shown the structure and the whole field of contents in which the course 'Introduction in Mathematics and Informatics' will be embedded we will try to gain reflected and reasoned considerations out of it.

3 The New Face

Tieing in with the model of Informatics Education presented at ISSEP 2005 in Klagenfurt [7] I decided to arrange the new face of the course along the diverse functions of a scientific educator.
Hence headmost a Mathematics and Informatics Educator is working as a

3.1 Mathematician and Computer Scientist

Her or his job is

- to *analyze* problems,

- to *review* and *judge* actions and
- to *summarize* the results.

Let's illustrate the different aspects of the basic activities by prototypical examples. The example in Mathematics is taken out of Fuchs [11].

Table 1: Illustrating Examples

Subject	Mathematics	Informatics
Analyzing	Studying the consumption data of a car in dependence of velocity we may come to the decision that a quadratic function will describe the continuous process optimally.	Simulating the limited growth of a population defined by a recursion may lead to the decision to calculate the values and visualize the data with the help of a spreadsheet.
Reviewing	Fitting the graph of a quadratic will show that the model is adequate for a particular but totally improper for another relevant range. We will have to rethink our decision. We will have to resume the analysis of the problem.	Wading through the wide variety of applied software we may find special programs for the simulation of dynamical systems. Inputs can be made with the help of a GUI. Variable outputs in the form of value tables or specific diagrams can be created easily.
Summarizing	The example tells us that *Modeling* is a cyclic process essentially.	Recapitulatory we must see that *Implementation Competence* which addresses the general view on implementation tools is one of the main competences of Computer Scientists.

Additionally the mathematics and informatics educator is

3.2 Educator, Psychologist and Philosopher

The scientist is engaged in educational science, social sciences, psychology and philosophy of culture from this point of view. Her or his position is standing midst of *Theory* and *Practice*.

Theory refers to educational topics such as internal differentiation, general properties of learners, interindividual or intraindividual differencies in the behavior of learners [34] or even educational ideals from a historical - philosophical point of view [30], [13].

PRACTICE
THEORY
M / INF
Educator

Figure 2: Teethed Disciplines

Practice refers to the public obligations of each subject against learners which means the development of attitudes, virtues, abilities and skills in detail. Actually social relevance addresses the discussion about educational standards [3], [20], [25] and the compilation of *Mathematics and Informatics Competences* [8].
Let's stick into essential topics of *Practice* in detail.

3.2.1 Models for Sequencing Courses

Deductive Organization

This style of sequencing courses may be characterized by systematic and smart representations, appropriate definitions, preparation of all required sources. In general you may say that it is a soften universities' lectures' style. In no case I want to argue against courses which go after essential Mathematical / Informatical Concepts to great lengths on one hand but I refer to the following common practice in deductive teaching just as strongly. Very often teachers of Mathematics or Computer - Science acquire the habit of a magician who celebrates the grip into the bag of tricks instead of pointing out recurrent strategies. Additionally this style of teaching is accompanied by sentences like 'The facts shown are for direct reading' or 'The proof of the statement is trivial' characteristically. Even naturally the deductive representation is advisable for teaching perhaps the following contents:

Table 2: Representative Topics

Mathematics	Informatics
Differentiation Rules	Logic: Reasoning Methods
Topology: Continuity	Elementary Sorting Methods
Axiomatic Stochastic Calculus	Number representation and Number conversion

Tasks' Oriented Representation

This method can be defined by a concentration on classes of tasks such as quadratic equations (Elementary Algebra) or forming straight lines' equations (Analytical Geometry) [26] in Mathematics as well as generating value tables (Calculating with spreadsheets) [15] or implementing text - processing functions in Informatics [9].
The tasks' oriented style plays a major role in school. Most of the prosperous approbated school books in Mathematics and Informatics are designed in this way. A lot of teachers give economy of time as reasons for their choice of Tasks' orientation.
Especially the sharp and strict partition of topics - each chapter introduced by so called 'Specimen Tasks' - may raise the impressions of isolated subject - matters by the students which comes in opposition to the aim of networked thinking [32].
Additionally the disconnectedness of 'innermathematical / innerinformatical' topics and applied Mathematics / Informatics makes an impact on students' concepts' development as a whole.

3.2.2 Genetic Method

The strategy that ties up to preconceptions of the students and incorporates problems into bigger holistic contexts outside or inside of Mathematics and Informatics may be named the *Genetic Approach*. Deeper considerations are introduced by intuitive or heuristic entries. Gradually one's horizon is broadened towards abstraction. Education must be seen as a process. The *Genetic Approach* comes along with following accents involving the names of famous scientist:

1. Education in Natural Sciences ought to be psychological, vital and application - oriented. Additionally it should assimilate historical facts into its approach [17].

2. Concepts in Natural Sciences have been created all times whereupon creating covers recreation as well as newly - creation. Moreover this activity mustn't be a privilege of the scientist but also students must have the opportunity to arrange their own insight - process [5].

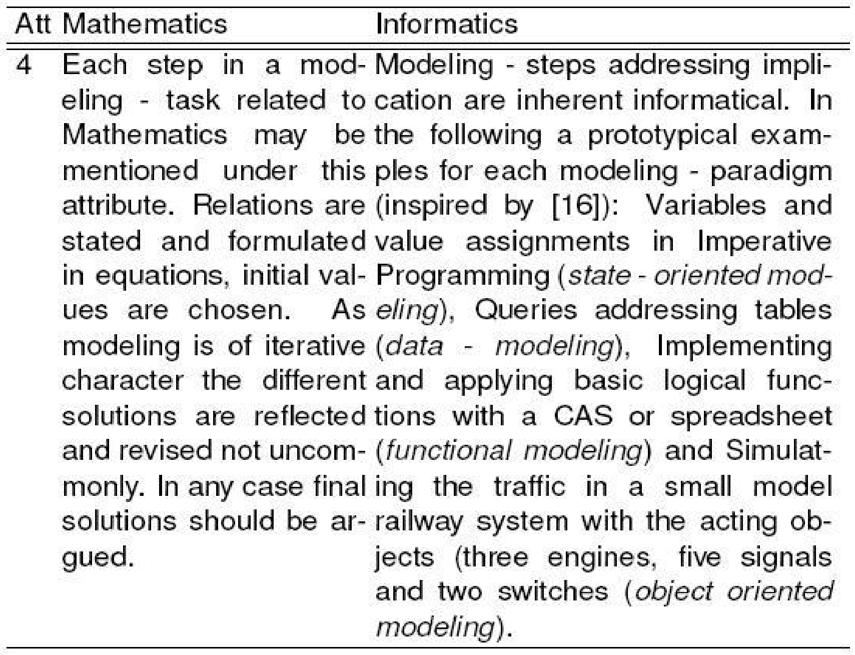
3. The *Genetic Approach* features the turning to the learner. Hence it ought to be exemplarily and socratic which addresses the fact that the development of concepts takes place effectively under discussion [33].

4. Discussing the Genetic point of view *Problem Based Learning* comes along with. Thereby learning is seen as an iterative design - process embracing externalisation (including some kind of presentable product, critical reflection (addressing the discussion of different possible solutions) and argumentation (characterizing well - grounded decisions in favor of a design product) [10].

Let's illustrate the single attributes with some short examples?

Table 3: Attributes, Subjects and Examples

Att	Mathematics	Informatics
1,2	Outline objects of your field of experience applying different kinds of projection (inclined, normal and central projection). Abstract constitutional properties of each representation. Include milestones of the history of projection into your abstract [11].	Addressing psychology, vitality, application and recreation *flexibility* and *participation* may come into the teachers' minds. *Flexibility* caters to tasks dealing with program conception, *participation* to any problems demanding the special engagements of the students [21].
3	Discuss the outcomes of your studies in 1, 2 with your classmates and your lecturer.	Both, *flexibility* but especially *participation*, contain a communicative dimension. In detail: *Flexible* designed Content Management Systems contain tool - areas with some kind of communication support like email or chat - rooms. Cultural or citizen *participation* via software for collaboration like Super - Brain from the department of Computer and System Sciences at Stockholm University opens up CMC (Worldwide) Computer Mediated Communication [23].



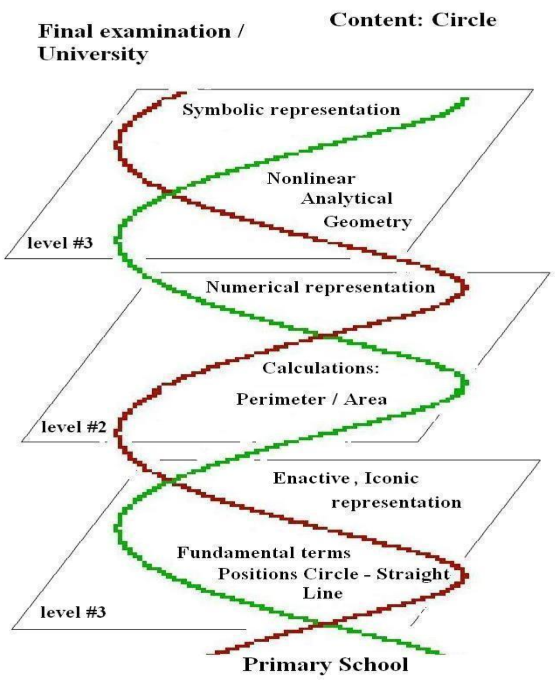
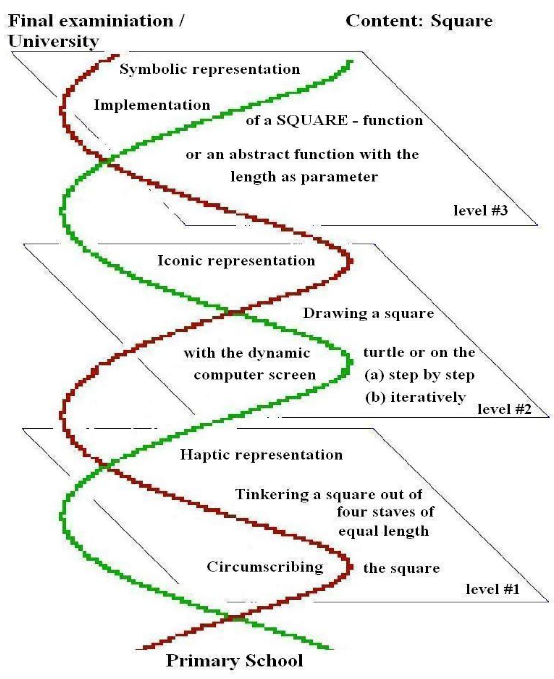
Att	Mathematics	Informatics
4	Each step in a modeling - task related to Mathematics may be mentioned under this attribute. Relations are stated and formulated in equations, initial values are chosen. As modeling is of iterative character the different solutions are reflected and revised not uncommonly. In any case final solutions should be argued.	Modeling - steps addressing implication are inherent informatical. In the following a prototypical examples for each modeling - paradigm (inspired by [16]): Variables and value assignments in Imperative Programming (*state - oriented modeling*), Queries addressing tables (*data - modeling*), Implementing and applying basic logical functions with a CAS or spreadsheet (*functional modeling*) and Simulating the traffic in a small model railway system with the acting objects (three engines, five signals and two switches (*object oriented modeling*).

3.2.3 Analyzing and Structuring the Process of Learning Mathematical and Informatical Concepts

Learning Following the Spiral - Principle

If you regard learning of Mathematics and Computer Science as process it will be inadvisable to postpone topics until a definite, extensive and closed explanation will be possible. In fact the discussion of serval aspects ought to be introduced even in earlier stages in an adaequate form of representation. Jerome Bruner [2] gets to the point with his hypothesis '... that every subject can be taught to any child at any stage of development 'intellectual - honestly' ...'.
The deduction for Mathematics and Computer Science is as follows: Instructions should start as early as possible in an age - appropriate form of thinking. In higher stages the instructions revive onto a final concept. This process can be interpreted as helix, the so called 'Spiral of Learning'. The following examples will illustrate the theoretical construct.

Table 4: 'Spiral of Learning Examples

Mathematics	Informatics
 Final examination / University Content: Circle Symbolic representation Nonlinear Analytical Geometry level #3 Numerical representation Calculations: Perimeter / Area level #2 Enactive , Iconic representation Fundamental terms Positions Circle - Straight Line level #3 Primary School	 Final examiniation / University Content: Square Symbolic representation Implementation of a SQUARE - function or an abstract function with the length as parameter level #3 Iconic representation Drawing a square with the dynamic turtle or on the computer screen (a) step by step (b) iteratively level #2 Haptic representation Tinkering a square out of four staves of equal length Circumscribing the square level #1 Primary School

3.3 Different Forms of Representations

Teaching 'intellectual - honestly' - as Bruner phrased it - focusses our concentration on the different forms of representations.

3.3.1 The Enactive Form

Following this concept informations are extracted from operations which are directly related to reality. It addresses actions wherewith the learning individuum can affect his environment immediately.
Mathematics: Record the track when walking around a fixed point taking into account a constant distance to this point as constraint (concept circle) [18].
Informatics: Try to find out as much as possible names of your parents, grandparents, great - grandparents and so forth. Then sketch a tree for the relationship 'Child_of' with the learner being the root and the ancestors as further knots (concept structuring information - file folders) [4].
Additionally the results of Piaget's research ought to be noted in this context [24].

3.3.2 The Iconic Form

This form of representation addresses the relevance of any graphical, stylized or diagrammatic description to the acquirement of mathematical and informatical knowledge. The iconic form is of notable significance for designing algorithms. As far as possible standardized are the following forms of graphical representations.

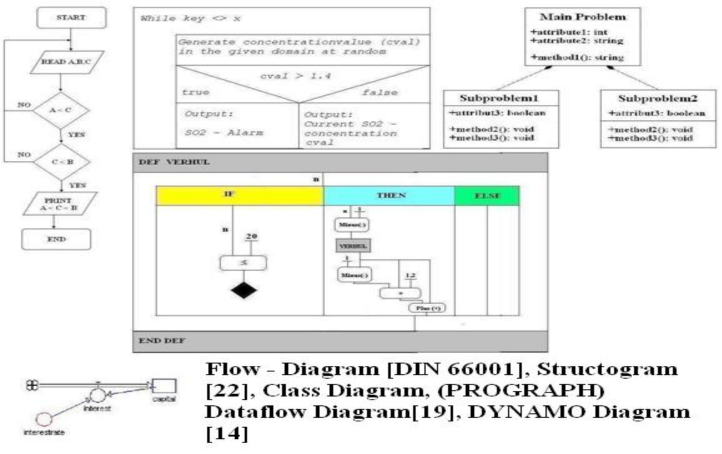
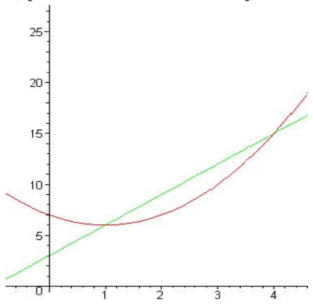


Figure 3: A Potpourri of Graphical Representations

With the proceeding use of computers in Mathematics especially in connection with Computer Algebra Systems (CAS) and Dynamical Geometry Software (DGS) the *Tile - Windows - Principle* was introduced. This concept points out the benefits of using different representations side by side for developing concepts. Imposingly the principle can be demonstrated in the context of equations' solving.

Table 5: *Tile - Windows - Principle* Example

Algebraic	>solve({y=x^2-2*x+7,y=3*x+3},{x,y});
	{x=1,y=6},{x=4,y=15}

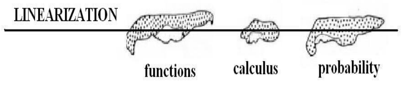
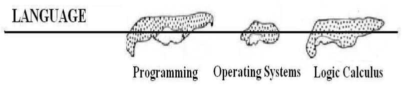
and Graphical interpretation for Solving a system of equations with CAS MAPLE	>plot({x^2-2*x+7,3*x+3},x=-1..6); 

As Mathematics and Computer Science are intrinsic *symbolically* I will only make mention of this form of *representation* in this paper for the kind of completeness.

3.4 Fundamental Ideas

Jerome Bruner made the proposal to arrange the several topics of each subject according to general principles. In 1982 the mathematician Fritz Schweiger picked up the concept and enunciated a remedy definition of fundamental ideas [27].
Accordingly the ideas should hold at least the following properties:
The idea
- should make a contribution to speak about a subject,
- should be instructive which means that it should make concepts more plain,
- should be good as 'vertical fiber' for the curriculum which means that it should clear and bundle topics and
- finally it should have been important within the historical development of a subject.
Shortly after in 1983 Fritz Schweiger puts the idea of Enhanced Redefining up for discussion [28]. Together with Klaus Aspetsberger and Karl Fuchs he published the ideas *Prototypes* and *Linearization* as fundamental ideas when using Computer Algebra Systems [1].
In 1993 Andreas Schwill renders the concept in his preface to 'Fundamental Ideas of Informatics' [29] more precisely.
Hence basic ideas have to comply with the following criteria:
- The *Horizontal Criterion* can be illustrated by an axis which pierces a multiplicity of topics.

Table 6: Section of the Multiplicity of Topics

Mathematics	
Informatics	 LANGUAGE Programming Operating Systems Logic Calculus

- The *Vertical Criterion* addresses the different levels in consideration of itemization and formalization.

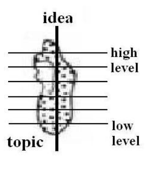


Figure 4: Illustration of Different Levels

- The *Criterion of Meaning* depicts the idea's commonsensical relevance.
- The *Criterion of Time* circumscribes the historical aspects of ideas, the monitoring of the historical development of ideas, concepts and structures. On the other hand this criterion indicates that fundamental ideas must be significant for longer periods.

Andreas Schwill presents the idea of Algorithmization (running from design - paradigms to evaluation) and *Structural Decomposition* (including *Modularization*) in his collection of ideas. Approximately at the same time in 1994 Karl Josef Fuchs published an article about the 'Logic of Fundamental Ideas' in Informatics [6]. Therein he put the ideas *Structure* (*Data-* and *Relations' Structures*) and *Modeling* up for discussion.

4 Bringing to a Close: Expectations and Perspectives

4.1 Objectives for School Practice

In recent years schools in Austria have tended to greater teachers' autonomy. Minutious instructions dealing with the contents have been reduced in support of substantial methodological and educational principles. More than ever the responsibility to form a modern teaching process with justice to the students will become the teachers' challenge. This development will make huge demands in teachers' vocational training courses. The adding of new computer science contents to traditional school curricula have caused new tasks for modern Mathematics and Informatics Education more than so far. Substantial knowledge of basic concepts will be important in almost the same manner as great pedagogical and educational skills for state-of-the-art schools in the future. The illustrated course will try out to come up to these new standards.

4.2 Stimuli for Research in Science Education

Focussing on these developments from a scientific point of view makes a case for an increasing research in science education. On one hand courses like the introduced should provide a basis and awaken interest in Mathematics and Informatics Education. On the other hand we must think of young scientists who are already motivated and competent in research beyond school practice. In a first step I have in my mind the offers for diploma theses dealing with educational topics in further steps the promotion of doctorate and postdoctorate programs for teacher students at the universities.

Fortunately a couple of journals and international conferences like IEE III open the opportunity to present and discuss educational concepts and programs.

References

1 Aspetsberger, Klaus; Fuchs, Karl, Schweiger, Fritz (1995) Fundamental ideas and symbolic algebra. In: The State of Computer Algebra in Mathematics Education Berry, J.; Monaghan, J.; Kronfellner. M.; Kutzler, B. (eds.), Bromley: Chartwell-Bratt, pp. 45-51.

2 Bruner, Jerome (1980) Der Prozess der Erziehung (engl. Process of Education). 104pp, Cornelsen Verlag.

3 Dorninger, Christian (2007) IT - Bildungsstandards (engl. IT - Educational Standards). In: PC - News Nr. 106, Nov. 2007, pp. 8-11.

4 Frey, Elke et al (2004) Informatik 1, Klasse 6 und 7 - Objekte, Strukturen, Algorithmen (engl. Informatics 1, 6th and 7th class - Objects, Structures, Algorithms). 97pp, Klett Verlag.

5 Freudenthal, Hans (1973) Mathematik als pädagogische Aufgabe (engl. Mathematics as an Educational Task). 680pp, Dordrecht: Kluver Academic Publisher.

6 Fuchs, Karl Josef (1994) Didaktik der Informatik: Die Logik Fundamentaler Ideen (engl. Informatics Education: The Logic of Fundamental Ideas). In: Medien und Schulpraxis, 4+5, pp. 42-45.

7 Fuchs, Karl Josef (2005) How Strict May, Should, Must the Borders be Drawn? In: Micheuz, Antonitsch, Mittermeir (eds.) Innovative Concepts for Teaching Informatics, pp. 7 - 14, Wien: Carl Ueberreuter.

8 Fuchs, Karl; Landerer, Claudio (2005) Das mühsame Ringen um ein Komeptenzmodell (engl. The Exhausting Struggle with a Competencies' Model). In: CD - Austra, Special Edition, pp. 6-9.

9 Fuchs, Karl Josef; Vasarhelyi, Eva (2007) Informatics with Casio CP 300+. Bilingual learning material, pp. 38, Norderstedt: Casio Europe GmbH.

10 Fuchs, Karl Josef; Landerer, Claudio (2007) Problembasiertes Lernen im Informatikunterricht (engl. Problem Based Learning in Teaching Informatics). In: Problembasiertes Lernen (engl. Problem Based Learning) Zumbach, J.; Weber, A.; Olsowski, G. (eds.) Bern:h.e.p. Verlag, pp. 158-175.

11 Fuchs, Karl Josef (2007) Fachdidaktische Studien (engl. Educational Studies). In: Fuchs, Karl Josef (ed.): Beiträge zur Didaktik der Mathematik und Informatik an der Universität Salzburg (engl. Contributions to Mathematics- and Informatics Education at the University of Salzburg), 386pp, Aachen: Shaker Verlag.

12 Herber, Hans-Jörg (1979) Motivationstheorie und pädagogische Praxis (engl. Theory of Motivation and Pedagogical Practice). 187pp, Stuttgart: Verlag Kohlhammer.

13 Heymann, Hans Werner (1996) Allgemeinbildung und Mathematik. Studien zur Schulpädagogik und Didaktik, Band 13 (engl. General Education and Mathematics. Educational Studies concerning Academic Pedagogy (Volume 13), 320pp, Weinheim / Basel: Beltz.

14 Hubfeld, Walter (1995) Dynasys. Ein Werkzeug zur graphischen Modellierung und Simulation dynamischer Systeme (engl. Dynasys - A Tool for Graphical Modelling and Simulation of Dynamical Systems). In: http://www.bics.be.schule.de/cif/Physik/software/dynasys.htm#Anfang (23.11.2007).

15 Hubwieser, Peter et al (2007) Informatik 2, Klasse 9 - Tabellenkalkulationssysteme, Datenbanksysteme (engl. Informatics 2, 9th class - Spreadsheets, Data Bases). 179pp., Stuttgart / Leipzig: Ernst Klett.

16 Hubwieser, Peter (2007) Didaktik der Informatik - Grundlagen, Konzepte und Beispiele (engl. Informatics Education - Basics, Concepts and Examples). 285pp, Berlin: Springer Verlag.

17 Klein, Felix (1928 / 2004) Elementarmathematik vom höheren Standpunkte aus (engl. Elementary Mathematics from a Higher Point of View). 256pp, Berlin: Springer Verlag.

18 Krainer, Konrad (1990) Lebendige Geometrie (engl. Spirited Geometry). In Journal für Mathematikdidaktik 11(2), pp. 165-166.

19 Matkin, Stan; Pietrzykowski, Tomasz (1985) The Programming Language PROGRAPH: A Preliminary Report. In: Computer Languages, 10:2, pp. 91-125.

20 Micheuz, Peter (2006) Informatics Education at Austria's Lower Secondary Schools Between Autonomy and Standards. In: Informatics Education - The Bridge between Using and Understanding Computers Mittermeir, Roland T. (ed.), Berlin / Heidelberg: Springer-Verlag, pp. 189-198.

21 Micheuz, Peter et al (2007) Mission Possible - Computers in "Anyschool". In: Informatics, Mathematics, and ICT: a 'golden triangle', College of Computer and Information Science Northeastern University, Boston, Massachusetts, USA.

22 Nassi, Ike; Shneiderman, Ben (1973) Flowchart Techniques for Structured Programming. In: SIGPLAN Notices, pp. 12-26.

23 Palme, Jacob (2005) Multi-lingual content management. Stockholm University.

24 Piaget, Jean (2003) Meine Theorien der geistigen Entwicklung (engl. Theories of Mental Development). 120pp, Beltz Verlag.

25 Puhlmann, Hermann et al (2007) Grundsätze und Standards für die Informatik in der Schule. Bildungsstandards Informatik, Entwurfsfassung der Empfehlungen der Gesellschaft fr Informatik. (engl. Principles and Standards of Informatics in Schools. Educational Standards of Informatics, a Conceptual Design of Commentations of the Infomatics Society). In: LOG-IN, Vol. 146/147, Enclosure to the Jounal.

26 Reichel, Hans-Christian; Müller, Robert (2002) Lehrbuch der Mathematik 5 (engl. Textbook in Mathematics for the 5th class (9th level of education). 318pp, Wien: bv - hpt.

27 Schweiger, Fritz (1982) Fundamentale Ideen der Analysis und handlungsorientierter Unterricht (engl. Fundamental Ideas of Calculus and Action-Oriented Instruction). In: Beiträge zum Mathematikunterricht, pp. 103-111.

28 Schweiger, Fritz (1983) Erweitertes Umdefinieren (engl. Enhanced Redefining). In: Mathematik im Unterricht, pp. 9-15.

29 Schwill Andreas (1993) Fundamentale Ideen der Informatik (engl. Fundamental Ideas in Computer Science). In: Zentralblatt für Didaktik der Mathematik, 25(1), pp. 20-31.

30 Spranger, Eduard et al (1969) Gesammelte Schriften, Band 5: Kulturphilosophie und Kulturkritik (engl. Volume 5: Philosophy of Culture and High-Culture Critique. 483p, Halle: Niemeyer Verlag.
31 Studienplan für das Lehramtsstudium an der Universität Innsbruck in den Unterrichtsfächern Biologie und Umweltkunde, Chemie, Geographie und Wirtschaftskunde, Informatik und Informatikmanagement, Mathematik sowie Physik (engl. Curricula for Teacher Studies in Biology, Chemistry, Geography, Informatics, Mathematics and Physics at the University of Innsbruck) (2007), Universität Innsbruck.
32 Vester, Frederic (2002) Unsere Welt - ein vernetztes System (engl. Our World - a Networked System). 177pp, dTV.
33 Wagenschein, Martin (1999) Verstehen lehren (engl. Teach to Understand). 180pp, Beltz Verlag.
34 Zimbardo, Philip; Gerrig, Richard (2004) Psychologie. Eine Einführung (engl. Psychology. An Introduction). 976pp, Pearson Studium.

A study on basic mathematics knowledge for the enhancement of programming learning skills

Ana Pacheco[1], Anabela Gomes[2], Joana Henriques[3], Ana Maria Almeida[4] , António José Mendes[5]

[1]CISUC - University of Coimbra and Department of Mathematics of the University of Coimbra, Polo II, Pinhal de Marrocos, 3030 Coimbra, Portugal, aa0506@mat.uc.pt

[2]CISUC - University of Coimbra and Department of Informatics Engineering and Systems, Polytechnic Institute of Coimbra, Rua Pedro Nunes, Quinta da Nora, 3030-199 COIMBRA, anabela@isec.pt

[3]CISUC - University of Coimbra, Polo II, Pinhal de Marrocos, 3030 Coimbra, Portugal, joanahenriques33@hotmail.com

[4]CISUC - University of Coimbra and Department of Mathematics of the University of Coimbra, Polo II, Pinhal de Marrocos, 3030 Coimbra, Portugal, amca@mat.uc.pt

[5]CISUC - University of Coimbra, Polo II, Pinhal de Marrocos, 3030 Coimbra, Portugal, toze@dei.uc.pt

Novice students often find it difficult to learn to program. This translates in high failure and dropout rates that are often felt by teachers and discussed in the literature. According to our own experience as teachers we believe that student basic mathematical knowledge may have a positive effect on their ability to learn programming. The paper presents a study that involved a small number of Mathematics students enrolled in an introductory programming course who showed deep difficulties in programming learning. The study tried to evaluate if the lack of mathematical knowledge was a main cause for the students' difficulties or if the problems were mainly caused by programming specific issues.

Keywords
CS1 programming, Learning difficulties, Mathematics background, Problem solving, Programming learning.

1. Introduction

The high failure levels in programming courses suggest that computer programming learning is a difficult process for many students. This is a problem that has been noted not only in Portugal but also, as literature suggests, in several Universities around the world [1,2,3,4]. Some very interesting international studies related with this problem were carried out by two different teams [1,2].
Many researchers proposed educational tools to support computer programming teaching and learning. Several of those tools use animation and simulation techniques, some stressing on algorithms others on complete programs. Some focus on low level details (e.g. showing data structures and their evolution during a program), while others use a higher

detail level, showing program behaviours, component relations and methodologies. Some systems just animate pre-defined programs and/or data structures, while others accept student's programs, allowing them to see how they work and, eventually, making the necessary corrections. Jeliot 2000 [5], Trackla2 [6], BlueJ [7] and Jhavé [8] are recent and known examples.

Although some of these tools are very interesting and useful at some of the learning stages, the problem remains and we continue to find reports mentioning a high failure and dropout rates in initial programming courses, even when computer science students are involved.

In our view several factors complicate the difficult task of learning programming. They are related with teaching methods, study methods and the specific nature of programming.

Although a student centred education focusing in individual needs would be important to develop programming competences, we believe that most institutions don't have conditions to support this approach. Time constraints and course sizes are common obstacles, as classes continue to have too many students to allow a personalized education, with suitable feedback and adequate supervision to each students needs. Additionally, many teachers seem to think that in higher education pedagogical care is not very important and that the students should adapt themselves to each teacher's style and options. For example, programming involves dynamic concepts that, frequently, are taught using static materials (projected presentations, verbal explanations, diagrams, drawings, texts, and so on) not helping many of the students to get a full understanding of the involved dynamics. Usually, teachers also forget to diversify their teaching strategies to contemplate the diversity of thoughts, rhythms and learning styles that coexist in the classroom.

Our own experience as teachers tells us that many students don't work enough and use study methods that aren't suitable for learning programming. This is a consequence of habits acquired in secondary education and also in some higher education courses. Often students focus their study on memorization of formulas or procedures, using readings and a certain mechanization of procedures as the basis of their work. This approach is clearly inadequate for programming learning, as studying a text book is not enough. Programming learning needs a very different type of study, demanding intensive practical problem solving work, reflection, and a true understanding of the concepts involved (not only memorization). Many studies suggest that a good mathematical background and good generic problem solving abilities are also important for programming learning [1,9]. Many students often lack the determination demanded by programming problems for which they don't find simple and straightforward solutions. In this situation students often give up or immediately look for external help, from colleagues or the teacher. This is an unsuitable attitude as, from an individual learning point of view, solving a problem after going through difficulties and correcting the errors made can be more valuable than easily solving a problem. The process of trying to solve a difficult problem, making errors, understanding and correcting them is a good learning activity that allows the students to gain experience and develop important cognitive structures. To go through this process it is necessary to demonstrate a good amount of persistence that many of our students often fail to show.

Another problem is that the specific nature of programming is substantially different from many other subjects. In this sense, programming learning requires skills that are not easy to develop like abstraction, generalization, transfer and critical thinking, among others [2]. Also the curricular structure of programming courses is often too centered on the syntactical details of a particular programming language, without promoting a real understanding of the purpose and utility of programming, and especially the mastering of important programming concepts. We think that the purpose of an introductory programming course should be to improve the students' problem solving skills necessary to develop a program and the programming language should only be seen as a tool to express ideas and algorithms.

Although the above mentioned questions are important we think that the most important cause for the difficulties felt by many novice students in programming learning is their lack of generic problem solving skills. The students don't know how to create algorithms, mainly because they don't know how to solve problems. Problem solving requires multiple abilities that students often don't have namely mathematical and logical knowledge. These ideas lead us to make some experiments to study the relation between mathematical competences and programming course results. In the third and fourth sections we will describe and analyze some of these experiences.

2. Previous studies

In our opinion and according to many authors there are a number of reasons that may explain the difficulties that students have in introductory programming courses. In [10] the authors present several reasons as causes to these problems including inappropriate methods of teaching and learning linked with a strong negative connotation associated with the subject. However, the authors highlight as the main problem the students' lack of various types of skills, particularly problem solving skills [1,9]. In the set of experiences that we have been performing [11,12,13] the participating students did not show deep knowledge of basic mathematical concepts. In our opinion this situation has a negative impact in their readiness to develop problem solving abilities and, therefore, in their difficulty to learn programming.
For instance, in a specific study we concluded that the students had deep difficulties in several areas, such as basic calculus and number theory or simple geometric and trigonometric concepts [11]. Some difficulties in transforming a textual and concrete problem into a mathematical formula that solves that problem, and limitations in abstraction level and logical reasoning, were also identified.
In another study [12], including only students without any previous programming experience, we found a positive Pearson correlation ($p=0.492$ at a 0.01 level (2-tailed) in one group and $p=0.416$ at a 0.05 level (2-tailed) in another group) between the students' programming results and their calculus ability. In that study we found that the previous programming experience was the most relevant factor affecting the programming course result, at all. This suggests that programming learning requires time and maturity.
The authors also acknowledged the difficult nature of the subject, stressing that programming learning is complex and requires effort and perseverance. Indeed, the programming abilities involve a set of skills that go far beyond knowing some programming language syntax. We believe that programming is mainly a problem solving task, in which the simultaneous use of different skills and cognitive functions is involved [14,16]. There are also authors who consider that the programming capacity consists of a hierarchy of multiple competences [17]. Accordingly to what some theoretical computer scientists like Papadimitriou or Vazirani [18] have been stating in their conferences, we strongly believe that programming is mainly a problem solving task, in which different skills and cognitive functions are involved.

3. The study

This study took place in the Department of Mathematics of the University of Coimbra in the year 2007/2008. It included a group of six Mathematics first year students, enrolled in the discipline of Programming Methods I, where they are expected to learn the basic concepts of programming, using Pascal as programming language. Those students were selected because they showed many difficulties in programming learning, giving clear signs that failure was imminent.

The experiments included two sessions in which students were faced with different worksheets about mathematical concepts in a theoretical and practical mathematical approach and in a programming perspective. The main idea was to find out what the students' main programming difficulties were. We tried to evaluate their programming ability to implement a simple mathematical concept, always after a mathematical explanation or exercise about that concept. We wanted to know if the programming difficulties were only related with programming (in)ability or if the lack of some mathematical knowledge could influence this difficulty.

The sequence of questions presented to each student followed the Bloom’s Taxonomy of educational objectives [19], so that we could see at what level each student showed significant difficulties. This taxonomy is widely used by educators to judge the depth and appropriateness of their coverage of course materials. The general categories of Bloom's Taxonomy are knowledge (memorization) at the low end, comprehension, application, analysis, synthesis and evaluation (judgement) at the high end.

The authors followed the experiences trying to interfere as less as possible in the student’s thoughts, but trying to intervene, in a sensible way, when some doubts arose. During the sessions the authors provided to each student exercises appropriate to his/her knowledge, depending on the difficulties previously experienced. This allowed us to use an individualized approach in the detection of the specific difficulties shown by each student.

During the first session the students were asked to answer three worksheets that were distributed according to each student reaction to the previous one. The three worksheets are closely connected and based on the concept of least common multiple (LCM), approached, first from a purely theoretical away and latter in a programming context [Appendix A].

The first worksheet [Appendix A – 1] was distributed to all students and intended to verify if they were familiar with the LCM concept and were able to apply it both theoretically (Question 1) and in an algorithmic context (Question 2). This allowed us to guide each student through tasks that are in consonance with his/her level of knowledge and skills. Given the simplicity of the second question it is easy to identify the students who are unable to develop the program due to difficulties exclusively linked to programming and not by any lack of knowledge about the concepts involved. The students that answered question 2 without difficulties were guided to question 3. This question is divided in sub-questions so that it is possible to identify students who show ability to solve more complex programming problems and that can follow a logical and consistent reasoning that allows them to develop towards other programming levels and areas. This first worksheet has also some more complex questions, each one being an evolution of the previous one, but always around the same concept.

Students who had difficulties in the mathematical concept of LCM were directed to worksheet 2 [Appendix A – 2] that focuses on LCM mathematical aspects, while the others went directly to worksheet 3 [Appendix A – 3]. Worksheet 2 begins giving a theoretical explanation of the LCM concept and explains it using a concrete example so that students can easily seize it. Next, students are faced with a set of four questions of increasing difficulty, following some levels of Bloom's Taxonomy. This way, we can follow student’s difficulties from a very basic level (knowledge of the LCM concept) - through its direct application - to exercises where the students should make their own conclusions from previous levels and generalize/adapt them to new situations. The third worksheet includes a set of increasingly difficult programming questions, all interconnected and linked to the concept of LCM. The questions were chosen so that they could reflect the different levels of Bloom’s Taxonomy, applied to programming aspects. This method is being increasingly used and there are already several authors who have been trying to develop Programming Taxonomies based in Learning Taxonomies [2,20, 21].

The objective of those questions was to identify the student’s difficulties and at what stage they start to appear. The first question is very basic and its seven sub-questions just verify if students can recognize and understand the effect of basic programming concepts, namely variables, attribution instructions, input/output instructions and basic control structures (selection and cycle). Questions 2 and 3 are at the Understanding level. Students who can answer the fourth question (Application level) correctly show that they completely understood the code, dominate basic programming techniques and have the reasoning capacity that allows them to reshape programs according to the new requirements. Question 5 is at the Analysis level. Questions 6, 7 and 8 are at the Synthesis level. Note that all these questions are related and require a full understanding of the mathematical concepts inherent to the LCM and the fundamental concepts related to programming that were trained in previous levels. The last question, 9, is at the last level of Bloom's Taxonomy (Evaluation). Students that can solve this question have internalized both the LCM mathematical concept and the programming basic concepts required to develop algorithms related with the LCM.

In the second session we followed a slightly different approach. We wanted to detect students programming difficulties to the maximum, so we concentrated in exploring programming questions in terms of Blooms Taxonomy. All the questions were related with concepts like LMC, multiples, divisor and prime numbers. However, this time we decided not to evaluate the students' mathematical knowledge, but instead to give them a sheet with the necessary Mathematical concepts, definitions and examples required to answer the programming questions.

Thus, in question 1 [Appendix B] there were seven sub-questions to verify if students can recognize and understand the effects of basic programming concepts, namely variables; attribution instructions, input/output instructions and basic control structures (selection and cycle). We also intended to verify if students can read and interpret some parts of the code used (question 2 and question 3) and determine the general propose of this code (question 4). These questions basically assess the Knowledge and Comprehension levels. The fifth question is at Application level and the sixth question is at Analysis level. We consider question 7 at Synthesis level, since the student had to integrate ideas from the different concepts into a unique plan or coding. Question 8 is at Evaluation level where students should make a judgment, recognizing the meaning, importance and utility of a small piece of code, needed in a certain context. The next section describes the experiment results.

3.1 Global results

Concerning the first session, most of the six students answered the first question on LCM between two numbers correctly, but only a small percentage managed to solve it efficiently. Most students just listed the multiples of the two numbers to find the first that was common, showing that they actually don't know how to generalize the concept. Only a few students tried to create a program to compute the LCM between two numbers, but no one did it correctly. Apart from headers, variables declaration and output/input instructions asking for and reading values, only one student tried to create the necessary algorithm, but without success.

All the students that were directed to the second worksheet, where the mathematical concept of LCM was introduced and explored, seemed to understand the concept, as they answered the related question correctly. However, only half managed to solve a specific problem involving the concept of LCM between three numbers.

Students revealed serious difficulties in understanding some basic programming concepts. All were able to correctly identify the variables, but not all recognized initial assignments to variables. Students showed more difficulties with repetitions. Only some recognized it and indicated the line where it started and finished. The purpose of the repetition was also unknown for most students. Another aspect is that not every student recognizes the input and/or output instructions. The difficulties increased when we asked students to analyse a given piece of code. Few of them fully understood the program and indicated the output that would result from a change made in the original code. From this point on, the questions began to involve more demanding cognitive concepts and the students started to give up or to give the wrong answer.

This diversity of student's responses is in line with the conclusions of Lister et al [2] where the authors, using the taxonomy of SOLO, grouped the student's answers in different levels.

In the second session only three students appeared, so we decided to analyse their progress, defining a profile result for each involved student.

3.2 Profile1

The student knows the concept of Least Common Multiple (LCM) between two numbers, however he can only determinate it by listing the multiples of the given numbers. He also knows the concept of prime number and how to get a prime number factorization using the Prime Factorization Algorithm, but can not apply it to get the LCM between two numbers. He also failed the questions that relate both concepts.
In terms of programming, the student knows the basic instructions (initiation, input/output, selection structures...). To calculate the LCM he used the concept of Greatest Common Divisor (GCD), but could not calculate it. In the worksheet the student identified instructions and basic operations and the initiation of explicit variables. However, he did not recognize - variables not initialized. Those failures have been documented in both sessions. The student does not show true understanding of repetitive instructions, since he considered in the first session the repetitive structure FOR as a repetition, but in the second session he also considered the selection structure IF as a repetition. Nevertheless, in some situations he seemed to understand this concept since he used it correctly. Some of the times he also demonstrated a true understanding about the meaning of instructions, not limiting his answer to a simple translation.
Although the student showed in several exercises that he knows and understands the concept of prime number, he failed when asked to develop a program that computes these numbers. In fact, he only presented the initiation instructions and read/write instructions, not making any attempt to get in the process of calculating prime numbers.
The student can create a program that prints out the multiples of a number, but couldn't make a slightly different program to print all the common multiples of two given numbers. In fact, he tried, but he used again the GCD that he does not know how to determine. Although the type and the order of the selected exercises intends to direct the student to a proper resolution, in the end the student presented the same program that he had submitted in the beginning of the session (including the same type of errors).
In the second session, faced with a given program and despite some difficulty of expression, the student identified all the instructions and explained the behaviour of isolated pieces of code and the program as a whole. However, when asked to develop a slightly different program (computation of common divisors between two numbers) the student failed to do it. There was an implementation attempt that only verified if a number is divisible by another.
In a new question the student was asked to complete a program, where the logic condition that controls the repetitive structure REPEAT… UNTIL was missing. The student had to choose between four options. The student chose the right option but could not justify why. However, in another question the student revealed a good comprehension about logical conditions.
In conclusion, despite the fact that the student can interpret pieces of code and simple programs, and make simple changes to a given program, he cannot implement a complete program, even if it is similar to the one provided before.

3.3 Profile2

The student does not know the concept of LCM between two numbers. After going through the worksheet that explains the concept, he managed to answer to direct questions about the LCM between two or more numbers. However, he wasn't able to apply the concept in a specific situation involving more than two numbers. In the second session he showed

knowledge about other mathematical concepts, such as prime numbers and Prime Factorization Algorithm.
In terms of programming, the student showed knowledge about basic instructions (initiation, selection structures...) and he identified all variables. He correctly identified the output instructions, but instead of indicating the input instruction READ, he indicated the limits of the repetition FOR. This error happened in both sessions.
The subsequent analysis suggests that there was not a true understanding about the behaviour of the FOR repetition. When questioned about what happens when the first bound of the repetition has a higher value than the second, the student believes that the control variable assumes only these two values. The students were uncertain about the impossibility to enter the repetition in this situation.
In the first session he failed to make any of the required programs and his attempts made no sense.
In general, the student is able to decipher literally a piece of code, but he doesn't have a true understanding about its behaviour. He also cannot explain the real usefulness of a piece of code or a program. In many situations, he remitted his answer to the concrete program's data and literally translated the program's instructions. However, when he was asked to create a slightly different programme (computation of common divisors between two numbers) he managed to do so, although presenting an inefficient solution.
In conclusion the student appears to have ability to translate pieces of the code but is not able to interpret or generalize it. Surprisingly, the student can implement similar programs to those given before.

3.4 Profile3

Although the student correctly answered the question about the LCM between two numbers, his approach seemed to indicate that the concept was not well understood. Thus, he was redirected to the second worksheet, which he fully and properly resolved. He seemed to know the concept of prime number, and how to use the Prime Factorization Algorithm, though in the calculation of one of the factorizations he made a mistake that seemed to be caused by distraction. The student cannot calculate the LCM by using the Prime Factorization Algorithm or respond to questions that relate these two concepts.
In terms of programming, he did not identify the input/output instructions (READ and WRITE), indicating in its place the undisclosed words (INPUT, OUTPUT). Along the worksheet he recognized instructions and basic operations, the initiation of explicit variables, not recognizing, however, as variables those that hadn't been initialized.
The student doesn't have a real understanding of the repetition concept. In the first session he recognized the repetitive structure FOR, but in the second one, he also considered the selection structure IF as a repetition. He correctly identified where the FOR structure begins and ends, but did not appear to have a complete understanding about its behaviour.
The student can literally translate the instructions of a piece of code or a program, but does not seem to understand its actual usefulness. He reports his answers to concrete program's data and he showed difficulties about making generalizations when some changes were made in the input data.
When he was asked to create a program (computation of common divisors between two numbers) slightly different to the one provided before (calculating the LCM between two numbers) the student failed to do so, confining his answer to the initiation instructions for reading and writing. He tried to explain, in words, the algorithm, but not correctly.
When asked to complete a REPEAT… UNTIL structure, he chose the right option, but could not justify why.

4. Conclusion

Given the small number of students who attended our sessions it was not possible to conduct a statistical study of the results. Although the three students' answers reveal different profiles we are not able to generalize the conclusions obtained due again to the reduced sample. However, we can make a few observations about the study that may be helpful for the development of further studies and learning strategies.
The students showed lack of knowledge about basic mathematical concepts, demonstrating only a superficial understanding of the subjects. However, we realize that such difficulties could be easily overcome, with a targeted training for the development of specific mathematical skills.
We could also verify that students did not make many efforts to find ways to solve the proposed problems. They seem to lack the necessary determination to improve their programming level. However, in their attempts, they did not make too many syntactic programming errors, most of them were simply mistyping.
Sometimes students write partially correct code, but the interaction was wrong, producing a globally incorrect solution. Other times students forgot to consider special cases in the input data, making programs that failed in those cases. Occasionally we noticed that the students' code to create the output was correct, but in the wrong place in the program.
However, most errors were logic errors. Frequently students failed to program a correct solution because of an incorrect interpretation of the mathematical concept or due to their incapacity to translate that concept in an algorithm. Therefore, more studies should be done in order to have more consistent results. We are now planning a set of experiments concerning, mainly, three aspects: student's mathematical knowledge, student's logic programming errors and student's problem solving and cognitive skills needed to programming learning.

References

1 Simon et al. Predictors of Success in a First Programming Course. Proceedings of the 8th Australian conference on Computing education; 2006, Hobart, Australia.

2 Lister, R et al, Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education; 2006, Bologna, Italy.

3 Jenkins, T. On the difficulty of learning to program. Proceedings of the 3rd Annual LTSN_ICS Conference; 2002, Loughborough University, United Kingdom.

4 Lahtinen, E, Ala-Mutka, K and Järvinen, H. A study of difficulties of novice programmers. Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education; 2005, Monte de Caparica, Portugal.

5 Levy, RB, Ben-Ari, M, Uronen, PA. The Jeliot 2000 program animation system. Computers & Education 2003;40(1):1–15.

6 Korhonen, A, Malmi, L, Silvasti, P. TRAKLA2: a framework for automatically assessed visual algorithm simulation exercises. Proceedings of the 3rd Finnish/Baltic Sea Conference on Computer Science Education; 2003, Koli, Finlândia.

7 Kolling, M, Quig, B, Patterson, A and Rosenberg, J. The BlueJ system and its pedagogy. Journal of Computing Science Education, Special Issue of Learning and Teaching Object Technology 2003;12(4):249–268.

8 Naps, T. Jhavé – Supporting Algorithm Visualization. IEEE Computer Graphics and Applications 2005;25(5):49-55.

9 Dehnadi, S, Bornat, R. The camel has two humps. School of Computing, Middlesex University, UK; 2006 Feb 22. [Online]. [February 2008]. Available from URL: http://www.cs.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf [Accessed February 2008].

10 Gomes A, Mendes AJ. Learning to program - difficulties and solutions. Proceedings of the International Conference on Engineering Education - ICEE'07 [CD-ROM]; 2007 Sep 3-7; Coimbra, Portugal.

11 Gomes A, Carmo L, Bigotte E, Mendes AJ. Mathematics and Programming Problem solving. Proceedings of the 3rd E-Learning Conference – Computer Science Education [CD-ROM]; 2006 Sep 7-8; Coimbra, Portugal.

12 Gomes A, Mendes A. A study on student's characteristics and programming learning. Proceedings of the EDMEDIA08, World Conference on Educational Multimedia, Hypermedia & Telecommunications; 2008 Jun 30 – Jul 4; Vienna, Austria.

13 Pacheco A, Gomes A, Henriques J, Almeida AM, Mendes AJ. Mathematics and Programming: Some studies. Proceedings of the International Conference on Computer Systems and Technologies - CompSysTech'08; 2008 Jun 12-13, Gabrovo, Bulgaria.

14 OECD (Organisation for Economic Co-operation and Development). Learning for tomorrow's world. First results from PISA 2003. [Online]. [2007?]. [cited 2008 May 20]. Available from URL: http://www.pisa.oecd.org/dataoecd/38/30/33707234.pdf.

15 Gagné E. The cognitive psychology of school learning. Boston: Little Brown and Company; 1985.

16 Seamster TL, Redding RE, Kaempf GL. A Skill-Based Cognitive Task Analysis Framework. In: Chipman, Shalin and Schraagen, editors. Cognitive Task Analysis. New Jersey (NJ): Lawrence Erlbaum; 2000. p. 135-146.

17 Lister R, Adams E, Fitzgerald S, Fone W, Hamer J, Lindholm M et al. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. SIGSCE Bulletin 2004;36(4):119-150.

18 Dasgupta S, Papadimitriou CH, Vazirani UV. Algorithms. McGraw-Hill; 2006.

19 Bloom BS, Engelhart MD, Furst EJ, Hill WH, Krahwohl D. Taxonomy of Educational Objectives, Handbook I: Cognitive Domain. New York (NY): David McKay Company; 1956.

20 Johnson, CG and Fuller, U. Is Bloom's Taxonomy Appropriate for Computer Science? Proceedings of the Sixth Baltic Sea Conference on Computing Education Research; 2007 Feb, Uppsala University, Sweden.

21 Fuller, U et al. Developing a Computer Science-specific a Learning Taxonomy. ACM SIGCSE Bulletin 2007 Dec;39(4):152-170.

22 Mead, J et al. A Cognitive Approach to Identifying Measurable Milestones for Programming Skill Acquisition. SIGCSE Bulletin 2006;38(4):182-194.

Appendix A

1-

1. Determine the LCM between 6 and 21.
2. Write a program that calculates the LCM between two numbers.
3. Imagine that you want to make a program that calculates the sum of two fractions.
   a) Modify the previous program into a function that receives two numbers, calculate and print the LCM between them.
   b) Make an algorithm of a program that calculates the sum of two fractions, using the function that you defined in a). The four values corresponding to fractions N1/D1 and N2/D2 must be introduced and the result of the sum shown in the form N3/D3.
4. Indicate, from the following set (2, 3, 4, 6, 7, 21, 37, 125), which numbers are prime numbers.
5. Write a program that calculates and prints all the prime numbers between two values n1 and n2.
6. The Sieve of Eratosthenes is a method to obtain a table of prime numbers up to a chosen limit:
   - Write a list of numbers from 2 to the largest number, say *n*, you want to test for primality. Call this List A.
   - Write the number 2, the first prime number, in another list for primes found. Call this List B.
   - Strike off 2 and all multiples of 2 from List A.
   - The first remaining number in the list is a prime number. Write this number into List B.

- Strike off this number and all multiples of this number from List A. The crossing-off of multiples can be started at the square of the number, as lower multiples have already been crossed out in previous steps.
- Repeat steps 4 and 5 until no more numbers are left in List A.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Write a program that implements the sieve of Eratosthenes to a number n.

2-

The Least Common Multiple (LCM) between two integers a and b is the smallest positive integer that is simultaneously multiple of a and b.

For example, " A florist sells roses for 3 € each bunch, which is the price of 2, 3, 4 and 5 bunches?" Answer: 6 €; 9 €, 12 €, 15 €, respectively.

Assume now that the florist sells not only rose bunches, but also carnation bunches for 6 € each. Suppose that Mr Phillips wanted to pay a certain amount for flowers, either roses or carnations. Indicate two price possibilities for the different flowers Mr. Phillips can buy. Answer: Calculate the multiples of 3 and pick two that are also multiples of 6. For example: 6 and 12.

In such conditions, what is the minimum amount that Mr. Phillips can spend? "Answer: LCM (3,6) = 6€.

1. Identify the correct answer, the LCM between 7 and 21 is:
   a) 3
   b) 147
   c) 21
2. Determine LCM (12,20,24).
3. Two friends meet each other in December 2007, in the city where they were born. One of them returns every 3 months and the other returns every 4 months. When will the friends meet again in that city?
4. What would happen if there was a third friend who was in the city on the same day and returned every 5 months? In other words, when will the three friends meet again?

3-

1. Given the following code extract:

```
1.  var n1, n2, i, j :integer;
    ...
2.  n1:= 3;
3.  n2:= 10;
4.  for i:=n1 to n2 do
5.     if i mod 2 = 0
6.        then write(i);
```

a) Identify the variables.

b) What is the initial value of each variable? Indicate the number of the code line where each variable is initialized.

c) Let `i=57`. Which is the result of the following instruction `i:=i+1`?

d) Let `i=101`. Which is the result of the following instruction `i mod 2`?

e) Is there a loop? If yes, please state the line where it starts and finishes.

f) Is there any instruction of entry and/or output data? If yes, identify it and write the correspondent line(s).

g) Explain what makes the instruction `for i:=n1 to n2 do`.

2. Explain, in plain language, what is the behaviour of the above code.

3. Assume that lines 2 and 3 are replaced by the following code:

```
7.  n1:=10;
8.  n2:=3;
```

What is the result of this change in the behaviour of the above code?

4. Enter the necessary changes in the given program (in 1), considering the data given in the previous question, so that the behaviour of program 1 is the same.

5. Explain the reason of the condition `if i mod 2 = 0`?

6. Based on the given program, write another program that prints out the multiples that exists between n1 and n1 * n2.

7. Upgrade the previous program in order to print the common multiples of n1 and n2, within the range of n1 and n2 * n1.

8. Change the previous program in a way that only the least common multiple between n1 and n2 is shown.

9. Will it be necessary to loop through all the numbers between n1 and n2? Suggest some changes to optimise the procedure for calculating the minimum common multiple between n1 and n2.

Appendix B

1. Given the following code extract

```
9.  Program LCM(INPUT, OUTPUT);
10.   VAR D1, D2, greater, smaller, i:INTEGER;
11.   BEGIN
12.     D1:=3;
13.     D2:=6;
14.   IF (D1>D2)
15.     THEN
16.       BEGIN
17.         greater:=D1;
18.         smaller:=D2;
19.       END
20.     ELSE
21.       BEGIN
22.         greater:=D2;
23.         smaller:=D1;
24.       END;
25.   FOR i:=greater TO greater*smaller DO
26.     IF ((i mod greater=0) AND (i mod smaller=0)) THEN
27.       break;
28.   WRITELN (i);
29. END.
```

a) Identify the variables.

b) What is the initial value of each of the variables? Indicate the number of the line, in the given code, where the variable is initialized.

c) Is there a loop? If yes, please state the line where the loop starts and finishes?

d) Is there any instruction of input and/or output data? If yes, state the line(s) where those instructions appear. If not, indicate how the values of variables are obtained.

e) Explain the propose of the instruction `FOR i:=greater TO greater*smaller`.

2. Explain, in current language, what the utility of the code between lines 6 and 15 is.

3. Explain, in current language, what the utility of the code between the lines 17 and 20 is.

4. Explain the goal of the all the above code.

5. Write a program that calculates, and shows on the screen, the common divisors between the two numbers.

6. Given the following function, aiming to determine if a number is prime or not, complete the structure REPEAT… UNTIL, using one of the options below. Justify your choice.

```
1.  Function xpto (i:INTEGER):BOOLEAN;
2.    VAR j: INTEGER;
3.        flag:boolean;
4.    BEGIN
5.      flag:=true;
6.      IF i>4 THEN
7.        BEGIN
8.          j:=2;
9.        REPEAT
10.         IF (i mod j=0) THEN
11.           flag:=false;
```

```
12.      j:=j+1;
13.     TO COMPLETE
14.     END;
15.     primo:= flag;
16. END;
a)  UNTIL (flag=false) or (j>=i)
b)  UNTIL (flag=false) or (j<=i)
c)  UNTIL (flag=true) and (j>=i)
d)  UNTIL (flag=true) and (j<=i)
```

7. The "Unique Factorization Theorem" says that the representation of any whole number, greater than 1, can be written as a product of prime numbers in a unique way except for the order of factors and the presence of units. Write a program that calculates, and shows in the screen, the decomposition of a number in prime factors. For example, assuming the number 84, it will appear in the screen 2,2,3,7.

8. Suppose that "`prime`" is a function that receives a whole number and that returns a boolean value (TRUE or FALSE) to reflect the fact that the receiving number is a prime number or not. Admit that n1 is a number that you want to decompose into prime numbers. Indicate the usefulness/meaning of the following code extract.

```
i:=2;
WHILE (NOT (prime(i)) OR (n1 mod i<>0)) DO
    i:=i+1;
```

Evolving modes of student use - whither the VLE?

Roger Boyle, Nick Efford, Royce Neagle

School of Computing, University of Leeds, UK, {roger,nde,royce}@comp.leeds.ac.uk

We consider the issue of where physically Informatics students choose to work. Technological change now offers them a range of points of access, at the same time as institutions are exploiting these modes to the full. In the context of a new institutional VLE, we have conducted a preliminary study of modes of use: we learn, unsurprisingly, that use of central bulk laboratories is diminishing, but that there may be subtle patterns of behaviour evident among individuals.
We note that these behaviours are driven by strong external forces and will not be countered, and further note that there may be cause to worry about students becoming 'virtual', both for their own academic benefit and their enculturation. We conjecture that conclusions for Informatics will be relevant to the whole academy as time passes, and propose work to monitor this issue.

Keywords
VLE, Digital Natives

1 Background and motivation

The Virtual Learning Environment [VLE] is now so common as to be a default piece of the academic landscape. First appearing during the 1990s and deriving from mechanisms designed to support distance learning[1], universities, colleges and high schools now routinely provide mature products as part of their teaching infrastructure. Their capabilities are very well known, but in summary, the VLE of 2008 will provide mechanisms for curricular repositories, coursework management, messaging between students and staff, video and audio handling, blogging and collaborative working using tools such as wikis.
The VLE has gone through a maturing phase: while once it might have been regarded as novel (often a drawback in educational domains), the number of competitors has shaken out to a very few: the market is dominated by BlackBoard [3], the public domain Moodle [13] and a small number of others.

[1]Wikipedia, inter alia

1.1 Expectations and behaviour of modern students

For universities, there can be a drawback in the ubiquity of such products: new students often come to universities expecting the environment to be ‘new’, ‘exciting’, ‘different’, ..., and there can be negative reactions to software environments that are just the same, functionally at least, to those they had at school. We might hope that the actual *use* that higher education [HE] puts them to goes some way to restoring expectation.
But there is another effect at work that presents more fundamental problems for HE which will be absent, or much less evident, in high schools: the modern student is a *Digital Native* [14] and comes to us with a far more blase view of technology in general. This is not a superficial or facetious observation – the preconceptions, attitudes and motivation of our learners is something that those of us who are Digital *Immigrants* (to use Prensky’s useful definition) sometimes struggle to comprehend. This often deep social rift can be especially evident when we try to deploy the technology which they take for granted in pursuit of our aims – however well we understand things, we remain immigrants.
Prensky’s useful terminology may well be viewed as one aspect of a much broader and well documented sociological effect: aspects of post-modernism that have been well documented by, for example, Bauman as *Liquid Modernity* [1], and Beck as *Second Modernity* [2]. These authors describe aspects of societies in the late twentieth and early twenty-first centuries: ‘*inhabitants live in a perpetual present*’, ‘*people are constantly busy and perpetually short of time*’, ‘*social networks are not being added on to the national container; they are changing its nature*’, ‘*a society preoccupied with the future*’. Many would agree that features such as this are especially evident among current students and schoolchildren. The sociological thesis is that these effects are not cosmetic, but fundamentally affect the way we live, and it is easy to see that attitudes will develop that present serious challenges to traditional modes of education. Students often exhibit a ‘here today, gone tomorrow’ approach with scant regard or interest to the longer term or historic causes – credentials as Digital Natives simply represents the communication channel they choose to use. These are major issues that others address – here we consider only that aspect which might impact on mode of computer use in education, particularly as exemplified by a VLE.

1.2 Implications for technology in teaching

The consequences for these issues may become significant, and may already be so. Habits of computer use among Informatics[2] students are often advanced, and often not representative of the broader community, but *do* often become so – what our students are doing this year is standard practise in science and engineering a few years later, and across the institution shortly after that. In computer demand and use, our students can be a signpost. And in many universities, Heads and Chairs of computing departments are reporting much reduced demand for the bulk facilities that have been essential provision for decades. The reasons are easy to

[2]In this paper, we use the term ‘Informatics’ to include degree programmes in Computer Science, Computing, and a wide range of cognate areas.

understand: consumer electronics are at a price that permits many students[3] to own platforms at least as good as the university provision, and domestic broadband Internet connection is ubiquitous. In addition, laptop users frequently have easy wireless access throughout their institution. Ergo, significant amounts of work are being done 'somewhere else', leaving the university sweatshops underpopulated.

An early reaction to this is approval: bulk laboratories are expensive to procure and maintain, and need regular upgrade – how much better if the institution has no need to provide them. But there may be drawbacks:

- Institutions commonly state expectations on student hours (in the host institution, 10 hours per credit point, aggregating to 1200 per academic year). Only a small proportion of these hours is formally scheduled and expectations are made about independent study, often cited as a feature of the successful student. In Informatics, this private study is rightly dominated by some use of computers. Various studies sound warnings about the assumptions made about this use of time, and these warnings carry more weight as the pool of students becomes broader, and pre-university education does less to encourage independent learning.

  It seems clear that student time investment and, more particularly, nature of use of time, can be critical to the quality of their learning [6, 11, 12]. Moving to systems such as VLEs where by design time is much less directed, may well have unanticipated pitfalls.

- Secondly, one of the major benefits of higher education is the *enculturation* of the student into her chosen discipline (whether it be Informatics, Physics, Philosophy, ...), and this comes most easily from physical interaction with peers and academics. Acceptance into the community [19] is not a luxury, but is an essential part of the transition that HE provides: '*students are too often asked to use the tools of a discipline without being able to adopt its culture*' [17], '*student do not only learn knowledge in the classroom, they learn a set of practises*' [4].

  Acquiring community membership (in all disciplines) has historically been semi-automatic as successful students live and work physically within a department among some of that community's strongest exponents. While physical participation in the academy is not essential, it is customary, and the consequences of its loss deserve consideration and caution.

1.3 This study

So we consider a scenario in which we detect – anecdotally – major changes in the patterns of work of some of our students that may give cause for concern, both for their curricular experience and their induction as computer scientists. In the host institution, a new VLE is being commissioned at significant expense and the scope for accelerating the change in these patterns is obvious, and in some quarters applauded and encouraged. We mean, accordingly,

[3]In informal surveys, we learn that nearly all students have private provision of a quality that matches the institution's.

to discover what we can about the nature of this use, and whether it influences in any way the quality of the student experience and their evolution into proper members of the community of Informatics. Earlier related studies [6, 16] evidence that collecting accurate data of this nature is not easy, and an objective approach to this is one of our aims: but it is also noted [16] that quantitative data alone is insufficient fully to understand behaviour, and qualitative follow-up is essential.
This paper considers the nature of the modern VLE, and an overview of the range of technology-based teaching that is currently seen and being developed. We note the extent to which these technological developments provide the potential, at least, for significant changes in student working practise. We then present results and comments on a preliminary study conducted at Leeds on the effects we have discussed, and draw some conclusions on what it means for the modern academy.

2 Technologies to supporting learning

There is a wide range of technologies on offer to modern HE: some of it (bulletin boards, plagiarism detectors etc.) is well established and mature; other aspects (e.g., pod-casting) are recent, hinging on pervasive ownership of consumer electronics. At the other end of the spectrum, special-purpose installations may be procured to facilitate specific modes of learning. An example is the 'Techno-cafe' (for example, [8]) where large screens are provided in a group working environment, making cross-site collaborations an easy possibility.
It serves to stand back and think about a time 10, or even just 5, years ago and to consider the contrast in usage. Various factors are simultaneously at work;

- The range of communication and digital electronics that teenagers will expect to own – phones, cameras, iPods, . . . – has grown significantly, as has their capability.
- Their relative cost to the consumer is dropping, certainly when capability is considered.
- Internet access by domestic broadband and public (or institutional) wireless has grown enormously.
- All of the above have been noted and used or acknowledged in schools.

These are statements of the obvious, but we stress that we are seeing the Digital Native here, not people exercising luxury choice, or recreational behaviour. This is of course less true of installations such as 'Techno-cafes', where a sense of novelty and difference will perhaps exist.
Overarching all this is the VLE: well established in nature since the 1990s, it continues to evolve and will these days routinely provide teacher and students with video and audio facilities, and easy cross linking into popular repositories such as *YouTube* and *Flickr*.
The Universities and Colleges Information Systems Association[4] conduct a regular survey monitoring the penetration of VLEs and MLEs in the UK: it is being conducted in 2008, but the last published report (2005) [10] tells us that:

[4]`http://www.ucisa.ac.uk/`

- They are widespread: at that time [in the UK], post-1992 institutions (ex-polytechnics) predominated.
- Majority use was for for accessing course material. PDP use was growing.
- Central university units usually provided support and future strategy, with close interaction with external national agencies.
- It was becoming an expectation on staff to use them, where they were available.

We can confidently expect these features to be more evident rather than less in the survey underway; the VLE has demonstrably become an academic ‘must have’, and not a Learning and Teaching option or luxury. The survey by design concentrated on HE, but we have abundant, if anecdotal, evidence that VLEs are similarly widespread in the pre-university sector. For students, they are routine.

The host institution for this work has used a homegrown VLE, ‘Bodington’ [5], for ten years. Some faculties within the institution have developed considerable experience over this period in the use of a ‘blended learning’ approach that combines VLE-hosted materials and activities with traditional face-to-face teaching via lectures, seminars, laboratory sessions, etc. There is a long-held desire to build upon this experience and develop an institution-wide blended learning strategy that addresses key institutional goals such as translating excellence in research and scholarship into learning opportunities for students, or refining assessment practise and improving academic feedback.

Two years ago, it became clear that the existing VLE solution would struggle to meet the future need for widescale adoption of a blended approach to learning and teaching, and the institution agreed the business case for procurement of a new VLE. The procurement process began in January 2007 with an invitation to tender via the European Union’s OJEU tender process and concluded in June 2007 with a decision to purchase licences and services from Blackboard. This decision was motivated by a number of considerations, among them the fact that Blackboard’s system offers a wide range of functionality, along with the ability to extend this further via a ‘plug-in’ architecture and integrate with the institution’s existing e-learning tools—notably Questionmark Perception [15] and Turnitin [18]. Blackboard’s widespread use by partner and peer institutions of similar size and complexity was also a factor in its favour.

The institution has a two-year rollout strategy for the new system, based on an expectation that early adopters (largely comprised of users of the existing Bodington VLE) will spearhead its use during the 2008-9 session and that the majority of modules will be making some use of the VLE during the 2009-10 session. Implementation of this strategy is being supported by appointments within each faculty of a full-time support officer and a part-time coordinator. These individuals have the job of assisting teaching staff with the transition to blended learning techniques using the VLE and with the migration of teaching materials and activities from Bodington to Blackboard.

The two-year rollout strategy is part of a broader, five-year vision stating that, by 2011-12, the use of the VLE and other learning technologies to provide a blended learning experience will be the normal expectation for all staff and students. To help the institution achieve this goal, funding is being provided to each faculty for pilot projects that explore innovative uses of the VLE.

Question	Yes
Do you have access to a desktop computer or laptop where you live, while you're at university?	100%
Regularly use mobile	88%
Allow university to communicate using mobile	75%
Regularly use SMS	88%
Allow university to communicate using SMS	81%
Regularly use IM	81%
Allow university to communicate using IM	47%
Regularly use social networking sites	83%
Allow university to communicate using social networking sites	44%
Regularly use sharing sites	64%
Allow university to communicate using sharing sites	37%
Regularly use virtual communities	5%
'Social networking sites are to talk to friends or make new friends'	80%
'Social networking sites are to search or share information'	36%
Not using social networking sites, but with no reason	10%
Not using social networking sites, because of no Internet	0%
Always have a mobile phone	98%
Always have an mp3 player	49%
Always have a PDA	10%
Always have an iPod Nano	15%
Always have an iPod Video	14%

Table 1: Selected results from University of Newcastle questionnaire. This work was conducted as part of CETL ALiC [7].

3 Technology Utilised and Owned by Students

The Digital Native of today is very technology savvy, and we present a snapshot.
A preliminary survey performed on first year students at the University of Newcastle in 2007[5] demonstrated a high percentage of access to technology – this is summarised in Table 1. In addition to institutional provision, all seemed to have independent access to a desktop computer or laptop while at university. One aspect to materialise from this survey is how students communicate with the University: only a third of students were found to have a landline with most students using mobile phones.
Additionally, there was widespread use of PDAs and similar 'this year' consumer devices. The data summary given above is but a snapshot - behind this the trend is clearly toward such accessories.
Many students use instant messaging and social networking sites such as Facebook and MyS-

[5]In preparation for publication.

pace. About half the students allowed the university to communicate using these means, and see social networking sites as a means to communicate with friends and make new friends. Only a third of students use social networking sites to search and share information. There is a wave of interest from HE in exploiting social networking which is proceeding with mixed success [9]; the Natives do not always welcome the Immigrants onto home ground.
The details of these data serve to verify what we may expect: all our students are Digital Natives and the way they choose to conduct their lives is determined by this. Their range of opportunities is broad, and they have mixed feelings about 'the university' intruding.
We anticipate that these patterns will strengthen within a small number of years – certainly, such a survey conducted 5 years ago would have shown weaker patterns of behaviour.

4 Working Habits of Students

We have set out to determine where and when students choose to work within the range offered to them: at simplest this is within traditional laboratories, 'at home' (which may of course imply a student residence), or using a laptop on the move, probably using the intuition's pervasive wireless. This is a crude classification that conceals many other modes of use and communication such as WAP and, for example, iPhones. For this initial study, we sought to learn (i) by percentage, where work is done; (ii) at what time(s)work is done; (iii) by percentage, where work requiring specific resources is done.
This is an indicative survey only and we make no suggestion that it is exhaustive or thorough. Students were canvassed via:

- Online bulletin boards
- An e-mail canvass
- Via a 'spot questionnaire' in a lecture

(The first two here may well have a self-selecting effect on the respondents). The following questions were put

1. There are three places you can study and work: Lab Computer, Home Computer, Roving on a Laptop. Please estimate the percentage of time you spend on each, and why.
2. Approximately how many hours a week during term time do you spend on a computer to study and complete work during Morning, Afternoon, Night, and why?
3. Estimate the percentage of time you spend while accessing University resources (e.g. coursework specific applications) in the areas of: Lab Computer, Home Computer (VPN, CITRIX or other), Roving on a Laptop (VPN, CITRIX or other).
4. How do you access School of Computing and University of Leeds resources?

The survey had 27 responses via email (5), bulletin board follow-ups (5) and paper submissions (17); it provides interesting outcomes and will serve as a very crude benchmark for fuller surveys in future years.

4.1 Results – how and where Students Work

Given the rough and ready nature of the survey (a small sample, almost certainly not fully representative), we can only present the roughest of results, but nevertheless they are food for thought, suggesting a bimodal split in the use of computers. Arbitrarily selecting a threshold of 60% to determine where subjects spend a majority of their time working, we found that 11 subjects preferred to work from home, while 10 preferred to work from a computer in the university lab (Figure 1, left).

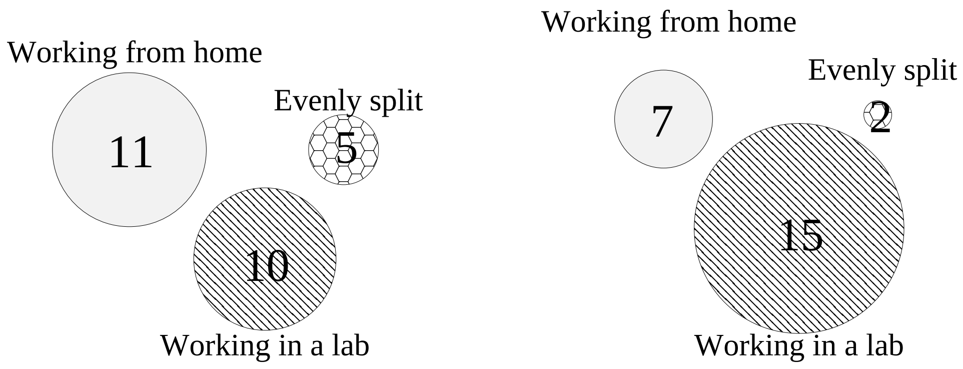


Figure 1: Number of students working in labs, at home, or both. On the left, general computer usage and on the right, doing coursework requiring specific resources.

Considering where work is done when resources within the university are required, we know that many students are able to replicate the university environment almost perfectly (perhaps to higher specification) on private systems – we might hypothesise that these are among the ‘better’ students.
Students who prefer to work from home but are unable to utilise resources from the university would probable work in the university lab. Our crude analysis of responses has some support for this (Figure 1, right)
Considering preferred hours of operation, the variability in behaviour is again very evident. There is no such thing as ‘average behaviour’, with some students preferring to finish academic work before the evening, and others preferring to work at night and only doing during daylight what the timetable strictly requires.

Working in Comfort and the Working Environment

Of greater interest than the raw numbers are the reasons and opinions that lie behind some of them; many of these appeared to be related to environmental issues. A representative sample of comments from ‘home workers’ is:

- *Lab computer chairs are quite uncomfy and the labs tend to have extreme temperature changes! I prefer the home comforts whilst working also.*
- *I would stay more in labs if there would not be freezing [sic]. Air conditioning was sometimes crazy. And also Labs were sometimes quite noisy.*

- *The main lab. is far too busy and freezing! The smaller one – quiet but far too hot! Home – nice surroundings, can grab a cuppa, quiet! 20% in uni is usually group discussion.*
- *. . . convenience, comfort, quietness . . .*
- *My work environment at home is a lot better than labs, e.g. quiet, more relaxed. Computer setup is a lot better than the labs.*

We note several comments on home comforts: interestingly, students that worked more often from the labs cited the good working environment they provided, with fewer distractions: *Prefer to use SoC computers as its easier to work in Uni (less distractions) . . .*

Support and Resource

Another prime motivator was access to help. Students who generally worked in the labs cited two main reasons: the resources available, obviously, and help was available from staff and peers if there were problems.

- *Work in labs when help is available or don't have tools that I need at home. Working at home makes it easier to take a break, and save time walking in and back.*
- *Lab is good for group work and moral support.*

The comments here are not all from weaker students who might be expected to be seeking help: we have held follow-up conversations in a small number of cases with the strongest individuals of the cohort, who intentionally occupy space near the staff they like to access. They are overtly joining the 'community' [19].

5 Discussion

We consider, despite a small and probably biased dataset, that we have identified an issue that may develop into a problem. The choices in front of students lead to very variable modes of behaviour, and the move toward more elaborate electronic support – by design – will increase this. Of course, increased affordability of suitable electronics will at the same time do so as well. This variety is largely untracked (although we uncover it here), is evolving, and is consequential. Does it matter? Our entry point for this study was twofold: the time students actually spend studying, and their success or failure, howsoever, in engaging with the community that is the academic department. Disentangling either of these is probably a deep and long-term issue, most unlikely to be answered by superficial surveys. Thus, our conclusions here are primarily a catalyst for future work, described below.
This brief survey does go slightly further: in collecting data conversations were held with two students who, entirely by coincidence, represented extremes of the spectrum:

Student A: (Very strong - a clean sweep of First Class results). A adopted a mode of working with 10% at most of his time 'at home', although he had a highly sophisticated domestic installation on which everything was possible. He preferred to avoid laboratories because of environmental concerns, but made tactical use of his personal laptop, occupying space frequented by other students and the staff that were key to his study. Being strong, he was a major asset to other students he worked with and presumably derived benefit from speaking with them: he derived much more direct benefit by frequent, planned interaction with staff.

Student B: (Very weak - a 'results struggler' who failed his capstone project . . . most unusual). B conducted nearly all his practical work at home, citing technical superiority and 'convenience'. The former reason is probably unarguable (laboratory machines look old very soon after bulk purchase), but in conversation he went on to explain that he often had difficulty motivating himself in an environment of distraction. Having established an absence habit, he was rarely seen in the Department and was a poor attender at project supervision meetings.

A here is the student we all want: clever, motivated, communicative, good study skills, good problem solver; B is the antithesis. The interest in these examples is that they have both managed to maximise their strength/weakness by chosen mode of operation – working at home, maybe A would be just strong, not very strong, while more academic interaction might have pulled up B's performance just enough to get a degree.
Of course, other examples will exist of home dwellers excelling and laboratory denizens failing, and for many reasons. This just confirms our view that there is a range of behaviours that we need to understand and track, and then plan (or compensate) for.
This study thus defines a range of questions that we seek to answer

- Across a full cohort, is our sample in any way representative?
- Do these patterns evolve as students become more senior, and how?
- Is there any correlation between student performance and mode of working?
- Is there any correspondence between a sense of community membership and mode of working?

The advent of an entirely new VLE environment is opportune: institutional policy directs that maximum use will be made of it (trivially, every module will have at least a rudimentary presence). VLEs come equipped with monitoring facilities, but we will augment these with the wherewithal to monitor the points of access of students, thereby automatically deriving the raw data illustrated in a sample here.
This is simply raw material, however: it then becomes important to disentangle truths in a qualitative fashion and we will conduct interviews and deeper studies to try to determine the academic effects of modes of working, and how these evolve during and after university study. The benefit should be some understanding of what – for the Digital Natives – works and what doesn't, and thereby an opportunity to play to strengths; to seek a balance between working virtually and direct contact with staff that is optimal for the individual. This is likely to imply modes of VLE use that are different to historical approaches.

6 Conclusion

Conclusions from a brief and unrepresentative survey are clearly of little value, but we consider we know enough to present an issue. We feel we have confirmed that a significant number of students are voting with their feet, and choosing to work away from the institution; probably a significant number of others are doing this ‘to some extent’.
We take the view that wholesale absence from the university environment is not good for dual reasons:

- Private study time is critical to winning a degree of quality, and there are doubts that it will be optimally or well spent in private home comfort, especially among newer or weaker students.
- Physical interaction with the discipline – the staff – should not be seen as optional in the education of the next generation of Informaticians.

Nevertheless, Knut-like we recognise that we cannot roll back the tide. The Digital Natives will behave as they wish and it is up to us to bend our processes to help them.
We will continue to monitor the mode and nature of use, with more precision than the simple survey presented here. We suspect there are patterns of use related to student prior experience and possibly to intellectual aptitude, but that may be hard to demonstrate. We are confident that these patterns are fast evolving in time, and it behoves us to be ready for what is to come.

7 Acknowledgements

We gratefully acknowledge the input of Martyn Clark in some of the background ideas presented here. Thanks are also due to the many students who described to us their experience, and to the VLE team at the University of Leeds.
Some of this work was funded by HEFCE as part of the ALiC Centre of Excellence in Teaching and Learning [7].

References

[1] Z Bauman. *Liquid Modernity*. Polity Press, 2000.

[2] U Beck. *The Reinvention of Politics. Rethinking Modernity in the Global Social Order*. Polity Press, Cambridge, 1996. Beck has written widely on Second Modernity and the Risk Society.

[3] Blackboard academic suite, 2008. http://www.blackboard.com/products/Academic_Suite/index.

[4] J Boaler. The development of disciplinary relationships: Knowledge, practice and identity in mathematics classrooms. *For the Learning of Mathematics*, 22(1):42–47, 2002.

[5] The Bodington open source project, 2008. http://bodington.org/.

[6] D Carrington. Time monitoring for students. In *FIE '98: Proceedings of the 28th Annual Frontiers in Education*, pages 8–13, Washington, DC, USA, 1998. IEEE Computer Society.

[7] CETL ALiC: Active learning in computing, 2008. `http://www.dur.ac.uk/alic/`.

[8] A Hatch and L Burd. Creating a working environment for group-work: Techno-cafe experience report. In *Proceedings of the 7th Annual Conference of the Subject Centre for Information and Computer Science*, Trinity College, Dublin, 2006.

[9] S Hoare. Students tell universities: Get out of myspace! *Education Guardian*, Monday November 5, 2007. At `http://education.guardian.co.uk/students/news/story/0,,2205512,00.html`.

[10] M Jenkins, T Browne, and R Walker. VLE surveys. 2005. At `http://www.ucisa.ac.uk/groups/tlig/surveys.aspx`.

[11] D Kember, Q Jamieson, M Pomfret, and E Wong. Learning approaches: Study time and academic performance. *Higher Education*, 29(2):329–343, 1995.

[12] S Kolari, C Savander-Ranne, and L Viskari. Do our engineering students spend enough time studying? *European Journal of Engineering Education*, 31(5):499–508, October 2006.

[13] Moodle course management system, 2008. `http://moodle.org/`.

[14] M Prensky. Digital natives, digital immigrants. *On the Horizon*, 9(5), October 2001.

[15] Questionmark perception, 2008. `http://www.questionmark.co.uk/uk/perception/index.aspx`.

[16] A Sandström and M Daniels. Time studies as a tool for (computer science) education research. In *ACSE '00: Proceedings of the Australasian conference on Computing education*, pages 208–214, New York, NY, USA, 2000. ACM.

[17] J Seely Brown, A Collins, and P Duguid. Situated cognition and the nature of learning. *Educational Researcher*, 18(1):32–42, 1989.

[18] Turnitin digital assessment suite, 2008. `http://turnitin.com/`.

[19] E Wenger. *Communities of Practice: Learning, meaning, and identity*. Cambridge University Press, 1998.

Evolution of an integrated course towards a sandwich course

Vincent Ribaud[1], Philippe Saliou[1]

[1] *University of Brest, Computer Science Department, C.S. 93837, 29238 Brest, France*

Recently, local software companies in Brest asked for work placement students at Masters level. In 2007-2008, an innovative programme in software engineering was adapted to the work placement requirements. In this programme, students learn software engineering by doing, with a long-term project as the foundation of all apprenticeships. Apprenticeship periods are intertwined with the work placement periods. We present adaptations we made whilst keeping the programme objectives. The programme uses several hierarchical models that meet exactly at the two first levels: an activity model coming from the ISO/IEC 12207; an apprenticeship model; an ability model with 3 domains, 13 families, and 48 abilities together with 11 transversal competencies. At 4 key moments of the year, each student is asked to self-analyse the activities he/she did with regards to the immersion system's ability model and self-assess abilities on a scale from 1 to 5. Self-assessment averages of the 2006-2007 and the 2007-2008 cohorts are used to compare existing and adapted systems. Finally, we draft perspectives on the repercussions resulting from the work placement system.

Keywords
Learning by doing, self-assessment, software engineering, work placement.

1. Introduction

Sandwich courses were so named to describe the alternation of a study period in college and a training period in industry which characterizes them; in France a common arrangement is one week in college followed by three weeks in a training situation during the period of studying for the degree, but a variety of periods and sequences exist.

In spring 2007, local employers in Brest decided to implement a recent French law on professional training. This law requires that 3% of employees be under 'sandwich' (or work placement) conditions. Companies asking for work placement students in the software field choose to use a system called "Contrat de professionnalisation" (*professionalization contract*) over a period of 12 months. During these 12 months, the work placement student is a full-time employee, although also attending university for certain periods. Salary is about 80% of the salary corresponding to the job that the course leads to.

For such contracts involving Brest University, the employers' demand has been essentially for an innovative program called "Software Engineering by Immersion" (*'Ingénierie du Logiciel par Immersion').* The main feature of this last year of the Masters programme is to learn software engineering by doing, with a long-term project as the foundation of all apprenticeships. Due to demand from companies, this programme has been adapted and the academic year 2007-2008 is running as a work placement course.

This paper presents, in section 2, the main changes made to the programme structure; an assessment framework in section 3; some comparison elements in section 4; and ends with perspectives on the repercussions resulting from the work placement system, and a conclusion.

2. Structure of the program

This section is intended to present how we adapted the program to the work placement system requirements in order to maintain its original objectives, structure and assessment.

2.1 Overview

Since 2002, Brest University has provided the software engineering by immersion paradigm as an alternative to other education systems. The immersion system is born from the necessity (due to the Bologna process) to design a fifth year for an existing and well tried 4-year technological education system. Since the 4-year system was sound and complete, this opportunity allows us to address the problem of educating software engineers from an unusual perspective for a university: to actually perform a significant software project - that is a sequence of stages organizing the activities intended to transform initial client requirements into a software product – always bearing in mind the goal of learning how to carry out these software engineering activities.

The year is divided into three periods, called iterations:

- a tutored apprenticeship period allows students to acquire software engineering knowledge and skills,
- an accompanied application period must transform knowledge and skills into competencies,
- and finally an internship period to put this into practice in a firm.

Learning is entirely based on a 7-month project, performed by a 6-student team within a virtual company, and tutored by an experienced software engineer. With the exception of English and communication courses, no lectures are given. The apprenticeship process was designed to be achieved in two iterations. During the first iteration (4.5 months), students are swapped around the different tasks required by engineering activities, and strongly guided by the tutor. During the second iteration (2 months), roles are fixed within each team and teams are relatively autonomous in completing the project, the tutor performing mainly a supervising and rescuing activity. During this second iteration, students rely on the apprenticeship process in order to autonomously perform a software development process supporting the production of the software product. An example of project is given in the figure 1 below.

Functions - The main goal of the project is to provide a semantic annotation tool able to annotate (indexing through metadata) Web resources, search (on metadata) in different modes, browse (hierarchically or with facets), manage RDF vocabularies (semantic schemas), and deal with the scope of annotations (public or private). The project uses Jena - http://jena.sourceforge.net/ an open-source Semantic Web programmers' toolkit - as RDF API.

Technical environment - The system uses a three-tier architecture in which the user interface, functional process logic, computer data storage and data access are developed and maintained as independent modules, on separate platforms. Sub-systems are: Oracle database, Hibernate persistent layer, Spring framework running on Tomcat, JSF for the user-layer.

Documentation - The tutor wrote the statement of work with expected needs. Main deliverables provided by the students are: Meeting report, Project Plan, Requirement Specification, Software Analysis, Software Design, Code, Integration and Validation Plan, Software User Manual, and Software Operator Manual.

Besides these engineering documents, students produce other kinds of document related to their apprenticeship: case study, usage guide, evaluation report, book or article summary, best practices, etc.

Figure 1 An example of a long-term project: a semantic annotation tool.

Whilst keeping the programme objectives, we adapted the system in the following way:

- the tutored apprenticeship period remains unchanged, but is alternated with the second period (two weeks each per month from September until mid-May),

- the accompanied application period is performed in the company, and may use a different development process than the process learned by doing during the first period,
- the last period is no longer a period of work placement, but rather a salaried period because the student is now a full employee of the company.

2.2 Reference framework

The immersion system uses a breakdown of apprenticeships into software engineering processes subdivided into software engineering activities, together with a set of apprenticeship scenes which provide the learning environment and defining tasks. This hierarchical process/activity/scenes model is adapted from the ISO/IEC 12207 [1] and is used as a reference framework for the learning objectives. Table 1 presents the two first levels of this hierarchical breakdown. From the university point of view, this division is the reference framework in a diploma-awarding perspective. Processes are course categories within the programme, activities are courses and scenes are classes.

From the 25 processes of ISO/IEC 12207, we concentrate on those related to the software development cycle, that is: 5.3 Development, 6.1 Documentation, 6.2 Configuration Management, 6.3 Quality Assurance, 6.4 Verification, 6.5 Validation, 7.1 Management, and 7.2 Infrastructure. The ISO/IEC 12207 Amendment 1 proposes a grouping of processes into categories. We reorganized the selected processes above in a same manner. The main process considered is the Development process. The other considered processes are not as wide as the development process, so these are retrograded to activities and reorganized in two processes: Project Management and Development Support.

In the ISO/IEC 12207, the Development process consists of the following activities : System requirements analysis and system design; Software requirements analysis; Software design; Software construction; Software integration; Software qualification testing; System integration and qualification testing. Our breakdown differs slightly because we do not need system activities (only software) and we put emphasis on computing constraints, essentially Technical Architecture.

Table 1 Activities breakdown.

Processes	**Activities**
Software project management	Project management Quality insurance Software configuration management
Software development engineering	Requirements capture Software analysis Technical architecture Software design Software construction Integration and validation
Software development support	Technical support Methods and tools support Documentation Installation and deployment

2.3 Assessment

2.3.1 Evaluation of industrial production processes

The process approach (as advocated in the ISO 9001:2000 standard) allows a company to describe its organization in order to produce defined results. The more visible processes are those undertaken to perform services or produce products that the company provides, which are often referred to as operational processes. In the software area, the supply activities chain provides deliverables (mainly documents) at each intermediary step of the development cycle. An important part of the quality management system deals with the evaluation of delivered documents, supported by two kinds of evaluations: those which inspect products during their elaboration and those which verify and validate (V&V) requirements.

2.3.2 Authentic assessment

The immersion system belongs to the constructivism approach and Tardif [9] states that the impacts of constructivism on assessment are numerous. In accordance with teaching practices, assessment necessarily relies on complete, complex and meaningful tasks. The assessment has to reflect effective, individual achievement of learning outcomes. The assessment must also take into account - directly or indirectly - cognitive strategies used by the learner. The constructivism paradigm implies that the teacher directly and frequently intervenes in the knowledge organization and hierarchy construction performed by learners. The role of the assessment is to report on the state of knowledge building. Assessments should take place in a context that is familiar to the student, using standards that are well known and presented in multiple forms [9].

2.3.3 Two kinds of activities in the immersion system

Samurçay and Rabardel [7] distinguish two faces in human activities. The productive activity is an activity made, oriented and controlled by the human subject to perform tasks he/she has to achieve with regards to the situation features. The constructive activity is oriented and controlled by the subject that performs it in order to build and develop competencies with regards to the situation and professional areas of action.

A formally organized education system has to make a distinction between a support activity which is a productive activity with learning objectives, and a constructive activity of competency development which happens on the occasion of the productive activity (with learning objectives) [5]. Our learning process organizes situation-problems present in the work activity in order to schedule productive activities (with learning objectives) and to control constructive activities that happen on these occasions.

2.3.4 Two kinds of assessment in the immersion system

Distinguishing two kinds of activities lead us to distinguish two kinds of assessment.

The immersion system attempts to mimic an operational environment, and productive activities (with learning objectives) have to be assessed with assessment procedures mimicking industrial usages. Hence, a first type, called Verification and Validation assessment, consists of evaluations based on the review of apprenticeship stages and on the qualification of software products. These evaluations are Verification and Validation (V&V) activities, performed in line with a predefined schedule.

However, most activities (especially during the first iteration) are constructive and need an (authentic) assessment that is fully integrated with the learning process. Thus, the second type is constituted by the tutor/author feedback cycle, the weekly progress meeting and peer reviews. These activities directly sustain knowledge building because they provide continuous feedback to learners. We call it regulation assessment referring to De Ketele "[…] an open process whose priority function is to improve the working order […] of a part or of the whole system" [2]. These evaluations are performed continuously and there is no firm separation between assessment and apprenticeship processes. Assessment sustains learning, giving useful information to students about the evolution of their learning process, and making them aware of knowledge they have, or are lacking.

2.4 Without work placements

2.4.1 Structure

Until 2007, the year was divided into three iterations: a 4.5-month tutored apprenticeship period, a 2-month accompanied application period and a training period of 4-6 months in a firm (see table 2). The acronym UE (*Unité d'Enseignement*) means Course Unit.

Table 2 Structure of the previous course.

Iteration	Period	Content
Tutored apprenticeship	From September to January	UE1 : Software project management UE2 : Software development engineering UE3 : Software development support UE4 : Communication and English language
Accompanied application	From February to March	UE5 : Putting into practice the knowledge, skills and competencies acquired during the first period
Internship	From April to August	UE6 : Performing an operational mission

The guideline for the two first iterations is the software development project, in which students will be immersed. Each company is associated with a company tutor, who plays different roles in the first and second iteration. The objective of the project entrusted to each company is to manufacture an information system, which responds to a real need of the computer science department (see figure 1 for an overview of a past project).

2.4.2 Objectives

The training course starts after the response to solicitation phase. The role-play partly consists of simulating the client-supplier relationship. The company tutor has to write requirements, then a response to solicitation including a technical and commercial offer responding to the anticipated requirements. These documents are very similar to real documents. They are only different in order to incorporate needs induced by our apprenticeship system:

- Two work packages are expected in order to map the two first iterations.
- Lead-time and cost are adapted to the size of teams and the time available for the training course (4.5 then 2 months).
- Technical constraints imposed by the client correspond to the objectives and means of the course.

During the first tutored apprenticeship iteration (4.5 months), an incomplete version of the software is built within a framework entirely driven and tutored by the company tutor. All software engineering activities are put into practice within a complete software development cycle. The expected software product at the end of this iteration corresponds to the first work package as defined in the requirements document. Our assessment process fits exactly into the apprenticeship situations. Students are continuously building knowledge and developing skills. Tutors regularly observe what is happening in order to provide a sound feedback and to perform authentic assessment.

During the second accompanied application iteration (2 months), each company is relatively autonomous in putting into practice the knowledge, skills and competencies acquired during the first iteration. A set organisation of the team is set up for the second iteration, structured around roles: project manager, analyst, architect, configuration and version manager, developer, integrator and qualifier. The expected software product at this end of this iteration corresponds to the whole software product as defined in the requirements document. During this iteration, the company is still accompanied by the company tutor, whose role is mainly to supervise and rescue.

Finally, the third iteration, i.e. the internship period, should ensure the continuity of the training course. Inside a firm, a software project (from A to Z) could be entrusted to the student, in line with the development process learned during his/her education. Practicing of software engineering activities is an alternative to the training period - for example, managing software configuration, capturing users' needs, analysis or design modelling, etc.

2.4.3 Assessment

Each iteration has its own pedagogical goals, and hence its own assessment characteristics. Throughout the first iteration, evaluations are regulation assessments (belonging to the second type defined in § 2.3.4) intended to support knowledge and skill-building whilst providing continuous feedback to learners. Each apprenticeship scene gives rise to one or several deliverables. Each deliverable is carefully examined and annotated by tutors, then the tutor feeds back comments to the authors together with improvements to be brought about. This assessment and feedback process is iterated (at least twice) until that the deliverable is considered good enough for future exploitation (problems could arise if the final delivery is not judged to be good enough). Each apprenticeship scene is assessed and awarded a mark. Because each scene is related to a software activity belonging to a software process, marks are consolidated in order to provide an average assessment for each activity/process.

The first iteration ends with the delivery of the information system corresponding to the first work package, qualified according to the validation protocol which was elaborated by the project team. This qualification is a V&V activity (belonging to the first type defined in § 2.3.4). Success of this activity (and hence of the evaluation) is related to the quality of the system produced, and is no longer tied to the completion of the learning process (it is still a formative experience because the tutor is giving feedback during and after the V&V activity).

The objective of the second iteration is to put into practice the knowledge and skills acquired during the first iteration. This second period always takes place over a period of 8 weeks. Thus, students spontaneously shift in terms of production rhythm. Assessment is performed on achieved deliverables according to production criteria: compliance and schedule. We did not have enough time to measure individual competencies and performances. We gave three marks for this iteration: one for the project itself (and the work carried out), one for the group viva voce examination, and one for individual reports on personal and group work. The mark given for the project relies on assessments of the essential deliverables of a software project, and is the result of a V&V activity.

Internship periods, as the third iteration, are assessed in our department, and awarded three marks: one for performance at work, established by the industrial tutor using a questionnaire provided by the faculty, one for an individual viva voce examination (with the participation of the industrial tutors), and one for an individual report on the work carried out during the internship.

2.5 With alternation

2.5.1 Structure

The principle of a program with work placements is that the educational objectives and assessment of course units are nearly the same as those of a program without work placements, but that some of the course units can be performed in a different way or during the periods in industry. In our case, two course units, UE5 Accompanied application and UE6 Internship were good candidates to be performed during the industrial periods.

In the previous system, the UE5 Accompanied application was intended to autonomously perform software engineering activities under the supervision of a faculty tutor. In the work placement system, the objectives are the same - but now it happens in a firm under the guidance of an industrial tutor. Assessment is still performed by the faculty under the rules of an internship period.

The UE6 Internship was designed to be an operational mission, and played the role of a pre-hiring period. In the new system, students are salaried employees and de facto in an operational position. Hence, the objectives are stuck to, and assessment performed as before.

The remaining course units have to be intertwined with the industrial periods. We choose a two weeks / two weeks rhythm from September to mid-May.

The year is now divided into two periods, the former with movement between university and company, the latter with a full-time period at the company (see table 3).

Table 3 Structure of the new course.

Iteration	Period	Content
Tutored apprenticeship	Half-time from September to mid-May (9 * 2 weeks)	UE1 : Software project management UE2 : Software development engineering UE3 : Software development support UE4 : Communication and English language
Work placement period in company	Half-time from September to mid-May (8 * 2 weeks)	UE5 : To perform software engineering activities in a real situation
Full-time period in company	From mid-May to September	UE6 : To perform an operational and salaried mission

The structure remains identical to the previous one, except in that the Course Unit 5 is renamed “Work placement period in company” and Course Unit 6 is renamed “Full-time period in company”. For the two modified course units, ECTS are identical, objectives remain the same and assessment differs only slightly, as explained above and below.

Unfortunately, first and second iterations are performed on different projects. The former is an apprenticeship project driven by the university and the latter is an industrial project driven by the companies with whom students are placed. However, the same reference framework (see table 1) and a unique abilities assessment framework (see section 3) are used throughout the year, providing students with a link between apprenticeship and work experiences.

2.5.2 Objectives and assessment

As before, the guideline for the first iteration is the software development project, in which students will be immersed. But most of the objectives from the two previous iterations are now assigned to the new, single one. That means, for one thing, that the shift from apprenticeship to production must be made during this iteration, whereas previously the first iteration was focused on apprenticeship and the second on production. Hopefully, students are maturing in parallel, thanks to the work placement periods, and are naturally shifting towards a professional attitude.

As before, the training course starts after the response to solicitation phase. Reference documents are project requirements and a response to solicitation. A unique work package is expected, and lead-time and cost are adapted to temporal organization of the training course (4.5 months over a period of 8.5 months).

During this unique apprenticeship/production iteration, the whole software product as defined in the requirements document is built within a framework entirely driven by the company tutor. But in contrast with the previous system, the tutor has to gradually reduce the help on offer to the student. During the first phases of the development cycle, products are assessed at the moment they are delivered, then feedback and corrective measures are provided by the tutor. In the latter phases, less feedback and help (or none at all) are given to students, and they have to do it by themselves. The tutor's main tasks gradually become broad supervision, assessment and rescuing if needed. Assessment shifts from being regulation evaluation to being V&V evaluation.

During the work placement period (4 months over a period of 8.5 months), each student is autonomous so as to put into practice (in his/her industrial position) the knowledge, skills and competencies that he/she is currently acquiring during the first iteration. It could happen that skills required by the industrial missions are learned later in the academic cycle and students have to resolve this deficiency by themselves. Assessment is still performed by the faculty under the rules of an internship period: one mark is awarded for performance at work and established by the industrial tutor, one mark for an individual viva voce examination, and one mark given for an individual report on the work done during the intertwined work placement periods. The mark given by the industrial tutor assesses the student's activity and is mostly measured from the produced deliverables - and is therefore the result of a V&V activity - as before.

For the viva voce examination and the individual report, the student is asked to establish clear links between the work carried out and the apprenticeship reference framework. A kind of reflection-on-action [8] is still required in order to show examiners how they have matured in the course of their work placement. In order to facilitate this difficult exercise, a half-day is devoted each month to a group meeting at which each student presents an account of what happened at work, how and why he/she acted as he/she did, to his/her peers (and tutors). Thus, an attempt at assessing meta-cognitive abilities is introduced - but it needs improvement.

The third iteration (4 months) is no longer an internship period. Students are no longer new employees; they are fully integrated within their companies and are paid the going rate. Companies both see and use them as full employees. Assessment is performed jointly by the faculty and industrial managers (work, report, oral) but using industry expectations.

3. Abilities assessment framework

3.1 Linking processes and activities to apprenticeship situations

Each activity represented in table 1 can be analysed from various points of view: expected knowledge and skills; stakeholder roles; input and output deliverables; required tools and resources, etc. At work, what makes sense for these multiple viewpoints is their articulation within the activity situation (the cohesion of the work situation). During the apprenticeship, it is also the (apprenticeship) situation which has enabled the multi-dimensional nature of activity to be understood. The (apprenticeship) situation has to transform students into learning humans, allowing them to put their own skills to work; the task to be performed allows the learning to take place. Situations are not natural: they have to be provided by the education system [4]. Putting students in a learning position - the situation - is the vital lead of educational design and practice. We distinguish the conceptual situation from the lived situation. A conceptual situation (called apprenticeship situation) is a situation-problem that "has to place fundamentals acquisition in actions that provide a goal to the students" [4]. An apprenticeship scene is the materialization - in action - of the conceptual apprenticeship situation. The system by immersion is a theatre play where, during a scene, different actors play different professional roles in order to learn professional [software engineering] activities.

There are two reference decompositions that meet exactly for the two first levels; the former is an activity model coming from the ISO/IEC 12207: process, activity, task; the latter is an apprenticeship model: process, activity, apprenticeship situation (scenes).

3.2 Linking processes and activities to abilities (competencies)

Meirieu [3] defines a competency as "the ability of a person to act in a pertinent way in a given situation in order to achieve specific purposes". Competencies are means of action which a person has to perform his/her activity with regards to the situation he/she has to deal with. The immersion system aims to acquire professional competencies that we prefer to call abilities. We carefully analysed the entire apprenticeship scenes for each activity in order to establish the abilities that theses scenes are intended to develop. We tried to answer to the question "what is the student able to do, once the scene has been performed?". This analysis gave us a set of abilities for each activity.

So we kept the 2-level breakdown of our reference framework, the first level being called competency areas (corresponding to processes) and the second level competency families (corresponding to activities), and we placed knowledge and abilities within each family (see an example in table 5). This breakdown contains 3 domains, 13 families, and 48 abilities together with 11 transversal competencies. We call the whole breakdown an ability model of our education system [6].

Table 5 An example of a competency family: "Software project management".

Knowledge area	Associated abilities or skills
• Software life-cycle model • Estimation and follow-up of development of software components: organization, workflow … • Traceability and requirements conformity • Project Plan	To use an ISO 9001 development baseline
	To apply a Project Plan and to update it if necessary
	Planning and project progress

3.3 Auto-assessment of abilities

The structure and definitions of this ability model are recorded in a document called the competencies compendium. Applied to the software engineering apprenticeship field, our ability model establishes a structure that directly supports the personal and team construction process of the knowledge and skills required to practice engineering of a software project. For each ability or transverse competency, the student assesses himself/herself at a maturity level. The assessment scale grows from 1 to 5; - 1 - Smog: vague idea (or even no idea at all); - 2 - Notion: has a notion, a general idea but insufficient to an operational undertaken; - 3 - User: is able to perform the ability with the help of an experienced colleague and has a first experience of its achievement; - 4 - Autonomous: is able to work autonomously; - 5 - Expert: is able to act as an expert to modify, enrich or develop the ability.

It is not easy for a student plunged into the 'doing' to keep in mind the abilities aimed at by the apprenticeship scene (and by the education system) and to establish links between all kinds of learning. That is the reason why, at four key moments of the year, each student is asked to self-analyse the activities he/she did with regards to the immersion system's ability model. So, four times during the year, students have to establish this inventory and communicate it to their tutor. We call this periodic inventory the Personal Follow-up of Competencies (PFoC) and it is, among others, intended to initialize a personal follow-up of competencies that could be pursued in a professional career.

4. First elements of comparison

4.1 Quantitative approach

It is not easy to measure the efficiency of an education system, and the impacts of evolutions within an existing system. In order to compare the system with work placements from the previous one, an indication can be drawn from the personal follow-up of competencies. For the 13 competency families, Table 6 presents the self-assessment average of the 2006-2007 cohort and the 2007-2008 cohort. Each cohort has 12 students.

Table 6 Personal follow-up of technical competencies for the 2006-2007 and 2007-2008 cohorts

		2006-2007 cohort			**2007-2008 cohort**		
Competency area	**Competency family**	**Sept. 2006**	**Feb. 2007**	**May 2007**	**Sept. 2007**	**Feb. 2008**	**May 2008**
Software project management	Project management	1.5	2.8	3.4	1.3	2.7	2.9
	Quality insurance	1.1	2.4	2.8	1.4	2.3	2.4
	Configuration management	1.2	1.8	2.9	1.6	2.9	3.0
Software development engineering	Requirements capture	2.1	3.2	3.6	1.8	2.8	3.0
	Software analysis	3.6	3.7	3.9	2.4	3.0	3.3
	Technical architecture	1.4	2.4	3.0	2.0	2.8	2.9
	Software design	2.8	3.2	3.5	2.3	3.1	3.6
	Software construction	2.7	2.7	3.1	2.5	2.9	3.4
	Integration and validation	1.2	1.3	2.7	1.3	2.0	3.2
Software development support	Technical support	2.3	3.0	3.4	2.4	3.1	3.5
	Methods and tools support	1.7	2.6	3.2	2.0	2.5	2.9
	Documentation	2.8	3.3	3.5	3.1	3.3	3.7
	Installation and deployment	2.4	3.3	3.5	2.9	3.3	3.7

All families follow a regular and roughly equivalent progress, with or without work placements. Due to the low number of students in cohorts, and the paucity of our statistical knowledge, no statistical comparison was performed. Yet some small differences could be pointed out . We remark that the final levels are slightly lower in the new structure than in the old. That is probably due to the loss of the second iteration in the new structure, because the breaking down into two iterations provided a learning process that was easier to follow. In the new structure, the intertwined iterations are, for some students, experienced as two intertwined learning processes which may be in conflict, slowing down the overall learning process. But the new structure achieves other goals, and is an answer to external pressure from local employers that could not be ignored.

As a point of detail, both cohorts established their February compendium before having performed the validation phase, and that should explain the low level of progress in the

“Integration and validation” family. But as you can see, the 2007-2008 cohort is much more aware of this activity, probably due to exposure during the placement periods.

For the 11 transversal competencies, Table 7 presents the self-assessment average of the 2006-2007 cohort and the 2007-2008 cohort.

Table 7 Personal follow-up of transversal competencies for the 2006-2007 and 2007-2008 cohorts

	2006-2007 cohort			**2007-2008 cohort**		
Transverse competency	**Sept. 2006**	**Feb. 2007**	**May 2007**	**Sept. 2007**	**Feb. 2008**	**May 2008**
Organization	2.8	3.2	3.6	2.3	2.9	3.3
Cooperation	3.5	3.8	4.0	2.5	2.9	3.7
Communication	3.0	3.5	3.6	2.5	3.2	3.7
Transfer / Sharing	3.0	3.5	3.7	2.6	3.3	3.5
Analysis	3.1	3.6	3.7	2.5	2.9	3.4
Abstraction	3.1	3.3	3.5	2.4	3.1	3.3
Cleverness	2.9	3.1	3.4	2.8	3.0	3.1
Innovation	3.1	3.3	3.7	2.8	3.2	3.3
Listening / Flexibility	3.3	3.7	3.8	2.8	3.2	3.5
Adaptability	3.2	3.8	3.9	2.6	3.0	3.5
Reflection	2.8	3.2	3.6	2.3	2.4	3.5

Progress is nearly the same, depending on the initial assessment. As for technical competencies, on average these are lower in the new structure than they were in the older one. But industrial experience is higher, which improves students’ employability.

This year, reflection activities were performed on the industrial periods rather on the university periods and we delayed it until January. This could explain the slow start of progress on this transverse competency for the 2007-2008 cohort.

4.2 Students’ remarks and expectations

Missions entrusted to students during industrial periods could differ significantly from those that the programme prepares for. Only six students out of twelve have to perform a project which corresponds to the skills learned from A to Z. Four out of twelve students perform maintenance activities for which only a part of the acquired skills apply, as well as some essential skills which are lacking. And two students have to perform multiple tasks and missions for which the skills needed do not exactly fit with those of the programme.

So, it is not surprising that several students pointed out that there was little application of the education in the industrial environment, or that there was a gap or a shift between required skills in the industry and learned skills at university. Some students were afraid that the time spent in the industry reduces the amount of skills that can be acquired in the program.

Obviously, almost all students expected a significant gain in terms of experience and a subsequent payment from the placement system. Most of them were attracted by the idea of long-term work with regard to shorter periods of work placement performed before. One student pointed out that industrial periods raise the desire to evolve within a company – an idea that had never occurred to him before.

4.3 Tutors’ remarks

We designed the immersion program in 2002 with a year divided into three periods, known as iterations: a 4.5-month tutored apprenticeship period, a 2-month accompanied application period and an internship period of 4-6 months in a firm. Objectives, achievement and assessment were different for each period, but were aligned with Donald Schön's idea of the reflective practitioner perspective [8].

It was at the request of local employers that the changes to the immersion system were instigated. The major difficulties encountered were in increasing the time spent on placement (and therefore decreasing time spent at university) and in intertwining the periods.

The second iteration (accompanied application) was a good candidate for taking place in a firm rather than at the university. Because the main objective is to acquire autonomy as a software engineer, and thanks to an assessment that is production-criteria oriented and based on achieved deliverables, there is no betrayal of the iteration in transferring it to a company. Unfortunately, intertwining apprenticeship and operational periods may cause additional difficulties for the apprentice - it all depends on what happens during periods in the firm, which is no longer under the university's control.

The third iteration (formerly an internship) is now an operational period but the goal - production – remains, and assessment remains roughly the same - although the performance level expected by employers may be higher.

The curriculum of our programme is based on a software engineering profession reference model, and it helped us transfer a part of the educational objectives (and related ECTS) to periods spent in a firm. However, the adaptation we made to our programme may be much more difficult to achieve with programmes based on a knowledge-oriented curriculum, and we are very conscious of the limits of our approach. The next session is an attempt to identify lessons that might be applicable in our situation, as well as to situations others may find themselves in.

5. Perspectives: repercussions of work placement

Adapting the immersion system to a work placement system produced and will continue to produce several repercussions, some on the system itself, and some on our university department. We tried to deal with this as well as possible, but we must engage in a long-term thinking process about it. We will try, in this section, to give some elements of these repercussions along the engineering perspective: strategic engineering, training engineering, and educational engineering.

5.1 Strategic engineering

There can be few links between a company hiring fresh graduates and the educational establishment from which these young employees graduated. But in a sandwich course, companies are paying students during the studying time and are paying the educational establishment for the education provided. So, the relationship between companies and educational establishments is quite different. There is a real need to build a veritable partnership with employers in order to, at least:

- Provide young graduates with a minimum level of professional experience and the skills required (technical and non-technical), making them rapidly adaptable and operational.
- Prepare students for the recruitment process.
- Tailor a single work placement programme to the context of each company

5.2 Training engineering

Several programs are candidates for adaptation to work placement systems. In our university, for example, there are work placement students in the science faculty (mechanics, electronics, computing, etc.), in the business faculty (management, bank-insurance, etc.), in the social science faculty (human resources, disability management, etc.) and so on. An educational establishment has to engage itself in a global strategy related to training engineering, putting emphasis on, at least:

- Promotion of a competency approach in teaching, learning and assessment
- Building of new degrees or adaptation of existing ones to the real needs of companies
- Encouragement of faculties, departments to develop work placement programmes
- Assessment of existing and new programmes

5.3 Educational engineering

The work placement system leads educational staff to re-think their teaching methods, re-organize programmes, and set up new modes of intervention. At the very least, we must pay attention to:

- Anchoring each student's individual learning path with his/her industrial experience and exploiting these experiences for educational purposes.
- Accompanying each student in the construction of his/her professional project.
- Favouring those periods outside the university by providing students with new skills gained through innovative teaching practices.

6. Conclusions

We presented adaptations we made to an existing 'learning software engineering by doing' programme in order to transform it into a work placement programme. The structure and the objectives remain identical to the previous course, and assessment differs only slightly.

Self-assessment of competencies of the 2006-2007 and the 2007-2008 cohorts were used to compare existing and adapted systems. No major differences were found. Students and employers are satisfied overall, but it is vital that we engage in a long-term thinking process about the repercussions of work placement, in terms of several engineering perspectives.

References

1 ISO/IEC 12207:1995, Information technology -- Software life cycle processes, International Organization for Standardization (ISO), Geneva, Switzerland.

2 De Ketele J.M., Roegiers X. Méthodologie du recueil d'informations. Bruxelles, De Boeck, 1993.

3 Meirieu P. Si la compétence n'existait pas, il faudrait l'inventer In IUFM de Paris Collège des CPE, 2005, http://cpe.paris.iufm.fr/spip.php?article1150 (last accessed May 20th, 2007).

4 Morandi F. Pratiques et logiques en pédagogie, Paris, Nathan, 2002.

5 Pastré P. Introduction In Recherche en didactique professionnelle, edited by Samurçay, R. and Rabardel, P., Toulouse (France), Octarès, 2004.

6 Ribaud V., Saliou P. Towards an ability model for software engineering apprenticeship. Italics, July 2007.

7 Samurçay R., Rabardel P. Work competencies: some reflections for a constructivist theoretical framework In Proceedings 2nd Work Process Knowledge Meeting: Theoretical approaches of competences at work, Courcelle sur Yvette (France), 1995.

8 Schön D. The reflective practitioner, New York, Basic Books, 1983.

9 Tardif J. L'évaluation dans le paradigme constructiviste In L'évaluation des apprentissages. Réflexions, nouvelles tendances et formation, Sherbrooke (Canada), Université de Sherbrooke. 1993.

Shift from teaching to learning with Web 2.0

Isa Jahnke[1], *Volker Mattick*[2]

[1]*Dortmund University of Technology, Center for Research on Higher Education and Faculty Development, Germany, isa.jahnke@tu-dortmund.de*

[2]*Dortmund University of Technology, Department of Computer Science, Germany, volker.mattick@tu-dortmund.de*

Abstract: The shift from teaching to learning means reversing the traditional teacher-centered understanding of learning, putting students at the center of the learning process and letting them participate in the evaluation of their learning. It is a shift from the teacher, who possesses and communicates knowledge with a particular aim, to the students, who acquire the knowledge they need to solve a problem with the help of the teacher. This paper presents a socio-technical community approach, which supports the shift from teaching to learning, and a first prototypical realization of a new kind of computer and Internet based teaching and learning systems at a Department of Computer Science in Germany. Numerous students (approx. 80-400 per lecture) participated in these lectures. Based on our empirical experience from 2002 until today, we present a good practice that combines face-to-face meetings and online discussions.

Keywords

CS Curricula, Concept and tools for e-learning, Community-based learning, Learning paradigm, Supporting learning processes

1. Introduction

The shift from teaching to learning is a vital point of discussion in higher education [13]. This paradigm means a shift from teacher-centered to student-centered teaching and learning concepts. Student-centered learning means reversing the traditional teacher-centered understanding of learning, putting students at the center of the learning process and letting them participate in the evaluation of their learning. It is a shift from the teacher, who possesses and communicates knowledge with a particular aim, to the students, who acquire the knowledge they need to solve a problem with the help of the teacher.

It is important to note that a student-centered learning approach also means that students need to be better qualified in managing their own learning process and therefore need more information about how their curriculum is structured. Not just pure professional information must be presented but also administrative information. The teacher should be accepted as a moderator of the learning process thanks to her/his professional expertise, not her/his formal status as a lecturer. This is not as simple as it seems because teachers lose their native rights, which they were used to have by their role alone.

Traditional computer-based learning concepts are teacher-centered, e.g. vocabulary training software. More up-to-date learning systems are more flexible, adaptable to different existing levels of knowledge and learning strategies, but are usually controlled by the teacher as well. Both do not implement concepts that embed the whole learning process into the given curriculum and empower the students to manage their own learning. These didactic concepts follow the same philosophy as the techniques used in the early days of the world wide web.

In contrast, Web 2.0., a buzzword created by O'Reilly in 2003 [9], emphasizes social software applications that are heavily reliant on human interaction, collaboration and social networking. The role of the user is changing from reader to author, from con*sumer* to *pro*ducer and both: "prosumer". Web 1.0 is still 'information download', whereas Web 2.0 is evolving into communication about information, and cooperation. Therefore, Web 2.0 also stands for Internet-based human interaction.

Current investigations of web-based communication show how groups can be adequately supported. Forte and Bruckman [6] as well as Wasko and Faraj [11] have investigated persons' motivations for contributing to Wikipedia, and its social change.

Web 2.0 works very well in the public and private sector. In contrast, it is just beginning to be used in universities. How can we use Web 2.0 or other technologies for supporting a shift to student-centered learning in Informatics education? How can we use these learning technologies to improve students' learning and the outcomes of our academic programs? How can we improve e-learning in student-centered settings?

This paper presents a socio-technical community approach, which supports the shift from teaching to learning, and a first prototypical realization of a new kind of computer and Internet based teaching and learning systems. Numerous students (approx. 80-400 per lecture) participated in these lectures. Based on our empirical experience from 2002 until today, we present a good practice that combines face-to-face meetings and online discussions.

2. Initial Situation

In 2001, the Department of Computer Science (at the Dortmund University of Technology, in Germany) had approximately 2,000 students. From 1996 to 2001, many students had not graduated with Computer Science degrees, according to an internal statistical report published in 2001. This report showed that many students ended their Computer Science courses after three or four semesters without a degree[1] or even moved to another university; others did not take the written examinations. However, we did not know exactly why the students were failing, and so, we wanted to find out why students were dropping their Computer Science studies.

We assumed that the problem was not only related to the content of the courses but also to students' management of their studies. So, the primary question was: how do German Computer Science students organize their studies at a university? Do they have enough information, and the right information, about how to organize their studies successfully?[2]

A first observation was that it is not enough for students to be experts in their subject, but they also need to be experts in managing their lives as students professionally. As a consequence, the faculty not only has to provide content, which belongs to the curriculum, but also has to provide all available information that enables students to manage their

1 The suggested study time for an Computer Science degree in Germany is nine semesters (4-5 years). The majority of students take 12-14 semesters to complete their degree (6-7 years).

2 German students often have a high degree of freedom: the decision in which semester to attend lectures or seminars or even in which semester to take examinations is left to the discretion of each student.

learning process. It is less relevant how much information is provided by the faculty, but it must be the right information.

A second observation was that learning and coordinating the learning process often take place not at the university but at home, whereas teaching is usually located at the university. So we saw the need for a system that permits a continuation of the learning process over distances. However, it is necessary to understand the status quo of the learning processes first, before integrating adequate technical applications.

A third observation was that traditional status groups and organizational structures are not very relevant for the learning process, because roles in a learning process are not identical to roles in organizational structures.

3. A new practice for teaching and learning

Starting from the problem of self-organization and study management, we launched the WIS project[3] in 2001. The aim was on the one hand to find out what the barriers to studying were, and on the other hand to establish what factors led to success for students of Computer Science. Finally, we wanted to give the results back to the students in order to initiate a discussion about these issues. The primary purpose of the empirical procedure was to help students build their own online community that would be concerned with study management.

In addition to the practical purposes, we used the project to study people's behavior as well as emerging changes of social structure and social roles in the online community. The project was based on an exploratory research method that includes ethnographic observations, qualitative interviews and questionnaires. The research design was triggered by an action research process [1].

The empirical exploratory method was essential, since we did not have sufficient hypotheses to explain why the students dropped their studies. From interviews conducted in 2001, it was clear to us that a software system was needed that was tailored specifically to the needs of the social system it was meant to foster. To develop this special software system, we tried to amalgamate the software-technological approach of the spiral model according to Boehm [4] with the phase model for communities according to Wenger et al. [12], as described in Jahnke, Mattick, Hermann [7]. The technical product could be similar to Blackboard or other VLE; however, an essential difference is that InPUD only contains a small subset of their functionality. Our approach focuses on students' learning processes; Blackboard is rather teacher-oriented.

3.1 Steps of the implementation process

Our empirical procedure included the following four phases of action research:
A) Identifying the problem(s): In semi-structured interviews, we looked at different students' problems with study management. In face-to-face interviews, held between 2001 and 2002 with an open-ended interview guide, we talked to 14 people (8 students and 6 professors/lecturers). The various aspects of how students manage their studies were summarized in different areas:

- Students knew the importance of attending lectures and learning groups even when

3 WIS is an abbreviation for the project 'Development of Computer Science' at the University of Dortmund (Prof. Dr. Thomas Herrmann). It was promoted by the state of North Rhine-Westphalia (Germany) from 2001-2004.

they did not attend[4];

- The city of residence was often not the same as the place where the students studied (many students traveled to the university by car or bus every day);
- The majority of students took on jobs to fund their studies; consequently they had less time to attend courses;
- New students at German universities needed a high degree of self-organization, but they had not learned it (and it had not been taught).
- There was a significant amount of information about Computer Science courses available; however there was no single portal that organized this information. As a result, students were forced to search through a jungle of information to find a suitable course.
- A large number of students said that they had become disoriented during the regular nine semesters (4-5 years), becoming unsure of when to attend which lectures and seminars and when to register for and sit specific examinations.

Based on these practical problems, a standardized questionnaire was sent out to the Computer Science students at the Technical University of Dortmund. 384 completed questionnaires were returned. This represented a total of about 20 percent of all Computer Science students enrolled in the bachelor courses. The results confirmed the following thesis: The majority of students knew in theory how to organize themselves for a successful Computer Science course, but they did not practise it.

B) Creating an information portal: The interview results from phase A prompted us to create an Internet-based information portal that would offer an overview of each lecture, seminar and course each semester, and a graphical plan of the first four semesters (corresponding to a bachelor course). We decided to use computer support for two reasons: (a) due to the large number of students who would be involved, and (b) in order to document the process for the next generation of students. Additionally, the portal would enable information from the study-management advisors and other university roles to be shared. In May 2002, the first prototype of the community system called "InPUD" (Informatics Portal University of Dortmund) was launched. The software system was revised twice and adapted to the changing technological standards on the web, to make it better maintainable and to be able to enrich it with new features that users ask for. The current version is realized by using a standard content management system, enriched with some domain specific components, programmed manually[5]. It is thus increasingly becoming a real Web 2.0 application.

The implementation of the portal leads to great difficulties, not with the students, but with some lecturers. This is not very surprising, because things get more transparent, which is not consistent with their traditional role as teachers. A lot of work and time was necessary to anchor this technical system in the teacher community. In the end, this took up far more resources than the whole programming work. To a developer of an e-learning system, it is new that you must motivate the teachers more than the students to use the system.

C) Supporting ways of active communication and collaboration: Based on empirical insights about the InPUD prototype, we added a discussion board about study management, the mentioned 'problem areas', and selected undergraduate courses in September 2002. The aim was to improve the transparency of successful study management factors. Information about study management and seminars was interwoven with online discussion boards. Thus, a computer-mediated knowledge sharing system was created. The knowledge sharing

4 It is not obligatory for German Computer Science students to attend lectures in order to take the examinations.
5 A publication that describes the technical realization is under preparation.

process was based on voluntary participation. As we will describe later, this was the beginning of an online community.

D) Continuous improvement: From 2002 to 2006, the project team enhanced the technical system and changed some things, for instance in order to improve the performance of the technical system. Meanwhile, numerous new discussion boards were added, and more information about study management was included. The InPUD community grew continuously[6].

3.2 Analyzing the implementation process

Especially from 2002 to 2006, we analyzed the InPUD community and its evolving social structures based on the following research methods.

First, in 2003-2004, face-to-face interviews with 8 experts were held. The experts came from the area of study management, had experience of 'university management', and knew web based IT systems very well. We asked what the crucial factors for successful study management were, in order to compare the experts' statements with the development of InPUD. Based on the empirical results of the interviews with the experts, we supported the InPUD community with new ideas. One example was giving members with formal roles a role name and making roles visible; for instance, the study-management advisors were labeled explicitly. Furthermore, we conducted participant observation of the online discussions in InPUD from 2002-2006. Moreover, the analysis also encompassed user statistics, communication structures as well as qualitative content analysis focused on social relationships in order to understand the social interactions.

As a result, in this exploratory action research process we identified empirically based theses about the emergence of social structures through interactive technologies. The results can be found in the following section.

4. The InPUD community

The InPUD community[7] can be described as a 'socio-technical knowledge sharing system' for Computer Science students at the University of Dortmund, Germany. It is available online at www.inpud.de. InPUD was launched in 2002. The InPUD community differs from other online communities that are built in people's spare time and are not a part of a company.

According to the characteristics given by Preece [10], the InPUD community is characterized by a large size (more than 1,300 people). The community is an extended part of a Department of the above-mentioned university and supplements the existing formal organization of the university. The primary content of InPUD is knowledge – and its collaborative creation – about the study of Computer Science, its courses and study management. The students get information about how to study successfully, and the opportunity to discuss study management, content and exercises of lectures as well as seminars. Thus, InPUD helps provide and share information to improve study practices. The community exists primarily online, but also has a physical presence through physical connections, e.g., networked students in different courses, seminars or lectures.

6 In 2007, InPUD 2.0 was installed. The community portal has been converted to account for the transition from the earlier national degrees ("Diplom") to Bachelor and Master degrees.

7 InPUD is an acronym for Informatics Portal University of Dortmund (Germany), http://www.inpud.de.

In more detail, the InPUD community includes an overview of all classes and lectures that are offered during the course of a semester. This information is structured consistently across all lectures or seminars. Included are information about the lectures, including any tutorials which are being held (and when they are being held), course materials, notices for examinations, lecturer contact information and often an open discussion forum, as well as news and search functions.

The information and content in the area of study management have been integrated with online discussion boards. These have enabled interested stakeholders to build active social interactions. The discussion boards exist for each lecture as well as for study management. They are embedded into an information website that includes facts about course guidance as well as graphical maps that suggest which course best to study at which time.[8]

InPUD has at least two main functions:

First, InPUD combines face-to-face lectures and seminars with online discussions (blended e-learning approach). At the time of writing, 30 boards are on-line, each with its own moderator. The discussion boards include discussions about selected lectures. It is possible

- to ask something about the content (e.g., a student asked "I don't understand why the following example isn't a socio-technical system: <a married couple talking over the phone>. Can anyone help me to?")
- to discuss exercises and their solutions on the discussion boards (e.g., a student asked: "With regard to the exercises <What is an appropriate definition for human-computer-interaction in contrast to human communication?>, my idea is the following. Who of you have similar or different solutions, and why?)
- to post something about the organization of a lecture (e.g., a student asked "Do we have to write an informal test at the end of the summer semester - or is it a formal examination?")
- to make some comments with respect to the evaluation of a seminar (e.g., the students created an brief quantitative online survey about the written examination: "The written test was easy – medium – difficult.")

Second, there are information and discussion boards that have been initiated by study management advisors, and course guidance. The discussion boards include questions and answers referring to study management, for example "*how to study successfully*", "*how and where to register for written examinations*", "*where to find the university calendar (timetable)*", "*what are the contents of Computer Science courses*", "*which semester is best suited for studying abroad*".

The InPUD discussion-board software also provides an awareness tool that provides information about activities of the users, formal roles and current status, and shows who and how many users are online at the same time.

The community members are primarily students from the Department of Computer Science, but also persons who are considering taking up studies, e.g., high school students. Other community members are advisors from course guidance and study management. The InPUD community comprises students who could theoretically meet at lectures. However, this face-to-face communication is unlikely due to the fact that the courses are oversubscribed. Sometimes there are more than 600 students on a single course - direct social interaction

8 German universities offer a multitude of lectures, and students have to create their own semester plan for lectures; meaning they can choose which lectures they attend and when to attend them.

with each person seems to be difficult to achieve.

The InPUD community is continuously expanding, although it has grown without any marketing or any external advertising.

Since its launch in May 2002, more than 1,330 registered participants have written more than 34,000 contributions. Registration and login are only necessary when actively contributing. Observation and reading are possible without registration and without logging in. Each user has access to all available information. InPUD is used by more than 60 percent of students within the Department of Computer Science at the University of Dortmund.

The number of requests has grown consistently, and the access rate usually peaks at the beginning of a new semester. In October 2002, there were only 171,408 requests. A year later, in October 2003, there were 292,155 requests, and in October 2004 this had increased to 491,330 requests. In the last years, InPUD has doubled its requests in every year.

About 2,000 students (100 percent) are enrolled at the Department of Computer Science in Dortmund. In April 2008, more than 1,330 (67 percent of all computer science students) were registered in InPUD. About 670 students (33 percent) were not registered. We do not know if these ‘non-registered persons’ were lurkers or if they did not use the information portal. With some exploring interviews, we have found out that they might be lurkers because almost all students use the information portal.

Figure 1 shows the analysis of the communication structure: About 1,100 members (of 1,334 registered members) contributed actively. The other 229 members were registered but still did not post. We assume that these registered InPUD lurkers (11 percent of 2,000 students) wanted to show their interest in the community although they did not actively participate. According to Preece [10], there are different reasons why they do not post, for instance, no motivation, no personal need, and curiosity without exposure. Maybe they are waiting for the “right” moment to post.

A core of 263 individuals regularly provided contributions: ranging from 26 to 482 postings (questions/answers) per individual. That is a significant number. The core members are the elders, leaders and partly the regulars [8]. The other 842 active members (617 and 225) made between 1 and 25 postings each. These members can be described as regulars, too, but also included novices and visitors.

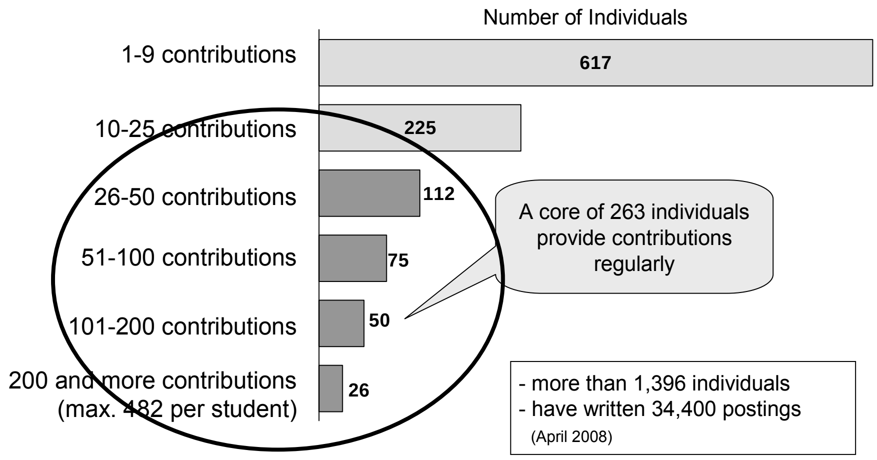


Figure 1 InPUD’s communication board - Number of contributions per individual

The success of the InPUD community can be measured by the significant number of students who actively participate. More than 60 percent of Computer Science students participate and use the community's content. The large number of participants indicates that a significant number of students appreciate this form of knowledge sharing. They discuss, ask questions, answer others' questions, come up with new ideas and help each other.

5. From Teaching to Learning – supported by InPUD?

A paradigm shift is taking hold in European universities. According to Barr & Tagg [2], the old traditional paradigm that has governed our universities is this: A university is an institution "that exists to provide instructions" (Instruction or Teacher-Centered Paradigm). Such an education refers to authoritarian instruction in which the teacher directs all learning. The new paradigm is: A university is an institution "that exists to produce learning" (Learning Paradigm) in order to support learners. Student-centered, learner-centered and outcomes-based education acknowledges the learner's participation in the learning experience. It refers to strategies which put the learner in control of constructing their own learning. This paradigm takes into consideration the pace, repetition, learning styles, motivation, self-regulation, and responsibility to learn. In such an approach, there is a shift from teaching to learning where the teacher needs to take responsibility for ensuring that all students learn – and make progress. Therefore, it requires a shift from the teacher as director of learning to facilitator of the learner's direction and creator of learning opportunities.

In order to apply effective learning methods, Dale [5] has created a learning pyramid with the most effective learning methods or means. These are: "teach to others / immediate use / explain something new to other people", "practice by doing" and "discussion in a group". Methods like "follow a speaker during a 90 minutes lecture", "reading", "audiovisual perception" and "demonstration" are less effective. They do not support the learner. Instructions are still important; however, there is a balance between learner's constructional processes and well-organized processes of instructions from the teacher [3]. Table 1 shows main aspects of the learning paradigm and how they are supported by InPUD.

Table 1 Supporting the Learning Paradigm with InPUD

The Learning Paradigm (according to [13])	*Characteristics of Web 2.0 with regard to the Learning Paradigm*	*InPUD and the Learning Paradigm*
Student-centered approach: focusing on students and learning processes	Software systems that support on-line human communication; creating new knowledge by many-to-many users' communication	++ InPUD creates feedback channels: It has dissolved one-way-communication from teacher to learners, and has enabled communication channels from learners to learners and teacher
Changing the teacher's role: from instructions to creation of learning environments and situations (teaching how to learn)	Web 2.0 applications support interaction, cooperation and collaborative learning, for example with discussion boards, Wikis, blogs etc.	0 InPUD has discussion boards. This supports the methods "teach others", "discussion in a group" and "explain something new to other people". [Research question: Does InPUD need more than one collaboration opportunity?]
Supporting the learner's role	From consumer to producer ("Con-ducer")	+ InPUD supports the "Learner 2.0": From consumer to learner who actively integrates new knowledge into personal context

Focusing on learning outcomes and goals		-- InPUD enables a 24 h online participation; however, it needs more organizational and didactic commitments in order to focus on learning outcomes
Promoting self-organization and active learning arrangements	Web 2.0 enables the building of new social relationships and social networking over the Internet	+ InPUD enables students to find others with similar interests, preferences or in similar situations, and supports discussions
Connecting knowledge acquisition and learning strategies	Communication about teacher's content	+ InPUD enables students to share different perspectives

InPUD can support the shift from teaching to learning; however, today this shift is often triggered by the students. A more reflective practice is needed in order to support the shift to the learning paradigm and to improve the InPUD scenario. This includes, for example: First, in order to support a mix between face-to-face and computer-supported collaborative learning, the teacher should define what cooperation is and how a group can or should cooperate. The connection between lecture and InPUD should be taught. Second, teachers should give orientation and foster convention on how to learn with InPUD or other Web 2.0 based learning applications ("learning model"). The teacher should explicitly explain rules and expectations for using InPUD at least at the beginning of a lecture.

What we have learned with InPUD is: There is *no single learning scenario with Web 2.0* because it depends on how people participate in cooperation, it depends on different learning cultures, on the teacher's role, goals, and content and students. According to the Learning Paradigm, in our future work we will research the above-mentioned criteria for enabling "creativity supported learning environments".

6. Conclusions

We have provided some insight on how the implementation of a system based on Web 2.0 technology can successfully support the shift from teaching to learning in Informatics. The InPUD community is a good practice scenario that shows how combining face-to-face lectures and online communication works.

Designing a socio-technical community to support student-centered learning is no longer primarily a problem of programming or tailoring a technical system. Good standard software such as content-management systems can be adapted in reasonably fair time. The main task is fostering the acceptance of a system that is not compatible with the traditional structure of a university or school. In a student-centered setting, there is little need to motivate students, but a great need to motivate lecturers. Curriculum coordinators can be a great help in designing such systems. Some crucial facts for acceptance, beside self-evident facts such as ergonomic use, are:

All information must be correct. If it is not, the error must be corrected as soon as possible. All information must be available when it is needed, better a bit before that. Not all information is relevant for students, even if faculty members sometimes believe it is. To implement such a software system in a department, you have to bear in mind that you are not the only one who promises that everything will become better when someone decides to use your system. In most cases, you are not the first either, so you must distinguish your system even from systems you never thought would be competition. If you convince committees of the advantages of such a system, it does not mean that they see any necessity to pay for it, and often there really is no money. So use standard technologies as much as possible and try to tailor them according to your needs.

From the study we may also derive some insight about Informatics students and culture:

- First: Students of Computer Science are not easy to handle when you try new teaching methods that involve computer technology, because every student is convinced that he or she can do the technology better. So do not even try to impress them by sophisticated graphical design or enormous features. You will lose.
- Second: Use only technical systems which are secure and well tested. There is the danger that students spend hours and hours in trying to hack the system or in finding errors, but not in dealing with the content you want to teach.
- Third: Students of Computer Science have, of course, in their majority a great affinity to using computers. So be very careful when offering incentives to use the system. Do not make it too cozy. Otherwise students might never come out of the system to visit their lectures or meet other students in real life. Anonymous logins are crucial.

It would be interesting to combine traditional e-learning systems, e.g., Blackboard or other VLE that rather support a teacher-centered approach, with the InPUD philosophy as well as Web 2.0 concepts, and examine the results.

References

1. Avison, D., Lau, F., Myers, M., & Nielsen, P. Action Research. Communications of the ACM, 42, 1 1999, pp. 94-97.
2. Barr, R. & Tagg, J. From teaching to learning. A new paradigm for undergraduate education. In: DeZure, Deborah (Ed.): Learning from Change. Change Magazine. 1995, pp. 198- 200.
3. Behrendt, B. From Teaching to Active Learning in Higher Education. In: UNESCO (Eds.): The University of the 21st Century. Muscat, Oman, 2001, pp. 415 – 434.
4. Boehm, B. W.: A Spiral Model of Software Development and Enhancement. In: IEEE Computer, Vol. 21, No. 5, May, 1988, pp. 61-72.
5. Dale, E. Audio-Visual Methods in Teaching. Dryden Press: New York, 1954.
6. Forte, A. & Bruckman, A. Why Do People Write for Wikipedia? Incentives to Contribute to Open-Content Publishing. 2005, Proceedings of GROUP 2005.
7. Jahnke, I.; Mattick, V. & Hermann, Th. Software-Entwicklung und Community-Kultivierung: ein integrativer Ansatz. I-COM – Zeitschrift für interaktive und kooperative Medien. 2 /2005, pp. 14-21.
8. Kim, A. J. Community building on the web. Secret strategies for successful online communities. Berkeley: Peachpit, 2000.
9. O'Reilly, T. What Is Web 2.0? Design Patterns and Business Models for the Next Generation of Software, 2005, from http://tim.oreilly.com/
10. Preece, J. Online Communities. Designing Usability, Supporting Sociability. Chichester: Wiley 2000
11. Wasko, M., & Faraj, S. Why should I share? Examining social capital and knowledge contribution in electronic communities of practice. Management Information Systems, 29, 1, 2007, 35-57.
12. Wenger, E., McDermott, R. & Snyder, W. M. Cultivating Communities of Practice. A guide to managing knowledge. Boston (MA): Harvard Business School Press, 2002.
13. Wildt, J. On the Way from Teaching to Learning by Competences as Learning Outcomes. In: Pausits, Attila / Pellert, Ada (Eds.): Higher Education Management and Development in Central, Southern and Eastern Europe. Münster: Waxmann, 2007, pp. 115-123.

Lectures on CS Taught to Introduce Students with Different Background

Paolo Rocchi

IBM, via Shangai 53, Roma, Italy, paolorocchi@it.ibm.com

Broad assortment of professional activities generates different perspectives on computing. Researchers on the educational field emphasize the need of specialized curricula on computer science (CS) which should comply with those diverging perspectives. But this is not enough, because different curricula stem from uniform principles of necessity. Multiple perspectives on computing compound the didactical problems in the introductory lessons on CS which revolve around common topics. The present paper puts forward an approach to computer science conducted over some years which addresses the teaching problems opposed by the different background of students. The didactical experiment exploits a parallel theoretical research driven to clarify basic concepts of informatics. The synopses of the very initial lectures are sketched and in the close we discuss some features of the whole course which adopts the inferential method typical of the educational style of mature disciplines.

Keywords
Deductive teaching in CS, lessons at university level, theoretical models.

1. Introduction

Broadly speaking the goal of education is to develop competence in a domain, but a broad variety of applications does not make the computing domain univocal. The relationships of aspects of computers to each other and to the living environment appear very intricate. The Report of ITiCSE'97 [1] highlights the pluralist nature of the computer sector and discusses six major perspectives on computing.

1) The theory of algorithm was invented by mathematicians and usually people see '*computing as mathematics'* because it is the most ancient paradigm in programming. The structures of linear algebra (vectors, trees, and graphs) provide the fundamental models for computer algorithms. Abstraction leads programmers in their everyday job; experts follow mathematical criteria when they approach complex solutions.

2) The notion of 'software engineering' was introduced in 1968 on the occasion of the Nato Software Engineering Conference that gave origin to '*computing as engineering*'. Software experts do not confine themselves to the theoretical study of algorithms, but design and build up software products in the living environment. Practitioners develop software projects and

prepare all the operations (manual and mechanical) surrounding the programs. Software experts go throughout the productive life-cycle including feasibility stage, analysis and design, implementation, testing, installation and maintenance. The entire field of computing has an enormous factual impact in the world because of computing engineering.

3) Recent progress opens the door of computing to poets, humanists, artists and other non-technical people and the Report of the ITiCSE'97 pinpoints the perspective '*computing as art*'. Nowadays we see the application of skills according to aesthetic principles, especially in the production of visible works of imagination that are pictures, poems, works-of-art etc. Graphical design skills are becoming increasingly important, particularly in multimedia and hypermedia applications.

4) Science includes both the systematic classification of knowledge and the discovery using observation and experiment. Computing is most definitely considered an empirical science where the subjects being studied are artefacts and natural phenomena. Computing is the study of natural and artificial information processes alike: '*computing as science*' emerges with evidence. The recent emergence of computation in the biological fields has opened new horizons. The old definition of computer science as the study of phenomena surrounding computers is definitively obsolete.

5) Computers influence human life through procedures that people execute in all the continents. The wide-spread introduction of Internet and the related change of people's lifestyle throughout the world manifest '*computing as a social science*'. Computer users shape the system by their use and in turn are shaped by the system itself. Systems become part of a broader culture and of an ever changing emergent phenomenon through which people create and recreate the worlds in which they live. It may be said that computing melds in part the future world within which humans are to live.

6) Computing touches on so many aspects of human activity that we could say that the major challenge is to keep a degree of tolerance of differing views and to permit cross fertilization of ideas from different strands. Anthropology, applied psychology, economics, ergonomics, ethics, history, linguistics, management, mathematics, philology, philosophy, semiology, sociology, and politics are some of the disciplines relevant to computing. Computing infiltrates a broad variety of fields and in order to maintain this flexibility the Report of the ITiCSE'97 concludes '*computing as interdisciplinary*'.

Multiple perspectives classified in the Report of ITiCSE'97 mirror professional activities and interfere with the design of curricula at graduate and undergraduate level. The multifaceted computer science complicates education design and entails different specialized curricula. It is clear that we cannot do justice to this diversity in current educational programs by applying monolithic didactics. But this is not enough because all the curricula start with introductory lessons which deal with basic topics and bring about a crisis of the pluralist approach. Diverging educational pathways are to stem from a rather uniform base of necessity. Inevitably basic tenets are universal.

The present paper aims at plunging into those questions, namely we deal with the demanding contents of the initial lectures which should cross all the paradigms on principle.

2. Theoretical Support

Normally a student at the graduate level has a certain background with programming in some language and with computer-based tools. The initial stage is the most favourable for forming a general understanding of computing which sustains the subsequent educational steps and the special professional purposes. Given the widening use of computers in the living environment universities should provide a solid basis from the cultural perspective. Important researchers highlight the demanding duties of the first stage lessons which should focus on the foundations of computing:

> ”We therefore recommend that the introductory course consist of regular lectures (..) the lectures should emphasize fundamentals (..) Lectures emphasize enduring principles and concepts”. [2]

The report "Computing as a Discipline" was the basis of a major curriculum revision in 1991, undertaken jointly by ACM and the IEEE Computer Society. It also suggested methods for educational researches [3]. Basics of computing are the right place for equipping students with cultures and languages which assist future professionals and users in whatsoever context.

Opening lectures in computer science should include depth principles valid for the paradigms from 1 to 6. By definition the basic tenets of a discipline are universal therefore the contents of introductory lessons on CS should be rather uniform. Introducing lessons should perform the arduous task to serve multiple perspectives, different didactical methods, and to prepare a variety of professionals-to-be using unified keys.

Presently, introductory lectures on computing do not live up to expectations. Educators base the initial stage of computer science on two cornerstones; they normally open a course with the illustration of a personal computer and the Turing machine. Even if these topics sound as apparently obvious starting points, some points arise against these educational pillars.

It is evident that the computer anatomy does not emphasize enduring principles and concepts. The physical description of a computer deals with down-to-earth facts and not with concepts. Teaching the hardware and the software components does not have the degree of generality necessary to improve the deep understanding of systems. A student becomes fully conscious of a machine when he knows the reasons that influenced engineers when setting up the various technical solutions of the machine. Illustration of a system component on the as-is” basis does not clarify its origins and sounds a low level approach to teaching.

The Turing machine is a conceptual model and a significant intellectual reference in the pioneering age of computing, but does not prove to be an adequate educational aid nowadays. Turing’s model hints at the idea that computers exclusively scale to abstract-mathematical problems. Students are driven to believe that computers basically calculate mathematical functions. This orientation is suitable for the ‘computing as mathematics’ perspective but does not fit the other perspectives. In fact the Report of the ITiCSE’97 underlines some critical aspects of the mathematical paradigm:

> “Historically, calculus was of immense importance to computing, which was primarily concerned with scientific and engineering problems. But presently, such computing occupies a much smaller percentage. Many of the mainstream fields such as artificial intelligence, database, software engineering, programming languages, and hardware rely extensively on discrete mathematics; hard-core scientific computing, relying on

calculus and analysis, has become a small sub-specialty within the computing community".

The Turing machine causes a student to turn away from the practical approach to his future or current profession. Jeff Kramer has written an interesting article upon the unnatural operational behavior caused by systematic abstract education. [4]. Moreover he highlights how the abstract orientation diverts the curricula designers who devote excessive space to mathematical topics. The survey conducted by Timothy C. Lethbridge exhibits the effects of pure mathematics on professional practice [5]. He asked software developers and managers from around the world what they think about 75 educational topics. The replies concur that some widely taught topics (most of them mathematical) have little impact on an everyday job. Their formal computing education does not always match the knowledge they need to apply to their daily work. Lethbridge closes:

"Mathematics, especially calculus, is extensively taught in computing programs. (...) On the other hand, relatively little mathematics turns out to be important for software engineers in practice and it tends to be forgotten. If we are to continue to teach the amount and type of mathematics [that we presently teach], we must justify it by other means than by saying it is important to a software developer's work: our data show that is normally not the case".

In conclusion, the physical description of computers and especially the abstract models do not support the pluralistic needs of education in the computer domain. The first stages of curricula on computing turn out to be somewhat inappropriate and should be renewed using more suited theoretical references.
The didactical problems do not rely entirely on educators. The true question is that theoretical studies on computing lag behind and didactics suffers for the lack of an exhaustive logical support. The initial lectures in computer science are the most precious from the didactical viewpoint instead they fail due to the lack of sufficient theoretical ground. [6].

The present paper illustrates a didactical experiment conducted in my company. The students whom we normally introduce to CS belong to different fields and share different perspectives on computing.
The work illustrated in this paper is theoretical and didactical at the same time. In particular the present account details the first educational passages and recalls the logic framework of this research [7] and other theoretical explanations [8].
I adopt the standard scientific method to illustrate this didactical experiment. The scientific method usually:

I) Forms a hypothesis,
II) Constructs a model and make a prediction,
III) Designs an experiment and collects data,
IV) Analyzes results.

The ensuing four sections follow this standard scheduling step by step. Roman symbols mark this correspondence. Because of the shortage of space lectures' summary is very concise and sounds rather autocratic of necessity.

3. The Didactical Hypothesis

I) - My educational investigation starts with this observation: take the machine X that brings forth the product x, it is obvious that the features of X depend strictly on x. Notably the structure and the various properties of X are dictated by x and in consequence educators must teach x in advance of X and must infer X from the features of x.

Concluding this didactical hypothesis may be reasonably taken: computers X manipulate information x, thus firstly it is necessary that a teacher elucidates what is information and later on a teacher derives the hardware and software solutions using the concept of information.

4. A piece of Information to Start

II) - Tens of theories on information has been devised in the last decades but no theory has gained the universal consensus. Luckily this failure does not impair the present project. Instead of taking a model of abstract information, I take the model of a piece of information. It is rather easy to see that the vast majority of experts have two important ideas in common upon informational items. Authors agree that *a sign has a body* and *this physical body stands for something in the world*. Pieces of information – signals, signs, messages, news, pictures, sounds etc. – take a material origin and represent something in the world. Claude Shannon calculates the signals conveyed in a channel which are physical quantities and symbolize letters. E.g. the string `1001` stands for the alphabet letter A, `1100` does B etc. Researchers belonging to different areas concord over two statements that I sum up in the following terms:

i) - *A piece of information is a physical and distinct entity*,

ii) - *A piece of information signifies something*.

Information is the root-cause of computer technology, I adopt hypothesis I) hence I derive a cascade of conclusions from **i)** and **ii)**. A didactical pathway introduces students to CS starting from the couple of issues **i)** and **ii)**. The *intended educational approach is deductive* in accordance to the style possessed by mature disciplines such as mechanics, chemistry etc.

5. The Didactical Pathway

III) - I held this educational-deductive path in several basic courses. My students varied from young people to mature professionals and they shared very different perspectives in informatics. Even if each student had a different perspective on informatics, I delivered introductory lessons that make a unique track. I sketch the synopses of the initial four lessons taken from this educational experiment. The chain-structure of the topics should emerge without any doubt even if the account is rather skeletal. The present small sample of lectures is confined to the very initial hardware topics and does not touch software programming.

5.1 *This lecture deals with the physical property of a piece of information.*

Definition **i)** sounds generic because it is introductory; it is necessary to specify the basic property of an object that makes this object a sign. Common experience shows how any material entity is capable of informing due to the property of being distinct. Every thing that is neat is information. E.g. the present writing conveys information as long as the printed characters are neat. As soon the characters whiten the reader has no longer information available. In conclusion: the object *E* is a piece of information if *E* differs from an adjacent comparison term *E**

$$E \neq E^* \qquad (1)$$

The entire world diversifies. The Grand Canyon, the Moon, the Sun, a street, a friend are distinct entities: they are examples of pieces of information in *the natural state*. There are free and natural pieces of information - like the foregoing examples - and *artificial items* - like printed words and electric signals that men/women create for the special purpose of informing. Distinct objects are either natural or artificial pieces of information of necessity.

5.2 This lecture deals with the essentials of the digital technology

Analog designers are inclined to imitate or to use the natural pieces of information existing in the world. For example an analog device handles the human voice which is a spontaneous piece of information. Digital designers show a far different behaviour: they ignore the natural signals and want the pieces of information to be perfectly distinct. Digital technology applies inequality (1) since the first step and prepares the bits `1` and `0` that verify `1≠0` in accordance to (1). In addition digital experts 'reinforce' the distinctiveness of bits using the *excluded middle principle* which constitutes a axiom of Boole's algebra [9]

$$\overline{1} = 0 \qquad \overline{0} = 1 \qquad (2)$$

Digital engineers go on through rational methods on the basis of (2), and construct pieces of information and circuits by two standard ways. Whereas analog experts adopt thousands of eclectic techniques, digital experts achieve all the technical solutions they want through standard procedures and methods. The first rule is the *progressive standard assembly*. Digital experts prepare pieces of information and circuits using standard components that are respectively bits and logic gates.

a) - Roughly the progressive assembly of a complex message includes five steps:
   1 - Experts establish the elementary information items: the bits.
   2 - Then they prepare a binary word using bits. A binary word stands for a character, or a figure, a symbol, a sound, a colour etc.
   3 - Experts prepare a word, a number or other simple structures by joining binary words.
   4 - Users prepare a text, a document, a picture, a piece of music etc. which are complex structures by joining the previous components.
   5 – Lastly, authors join various informational forms created in the previous steps and obtain a hypermedia.

b) - The assembly of a circuit encompasses four major steps:
   1 – Experts build up the *logical gates*: AND, OR, NOT. These gates never ever act in a manner different from the defined plans due to the

excluded middle principle. Hence it is easy for a designer to connect the outputs of one gate to the inputs of another gate. Engineers create a digital circuit from the logical gates like a sort of building blocks through the ensuing steps.

2 - Designers prepare *complex gates* such as NAND by joining AND and NOT.

3 - Later on they create *combinatorial circuit*s using the previous gates, and *sequential circuits* that are to be synchronized such as *memories*.

4 – Lastly engineers create the *finite-state machine*, the definitive solution, using the components of the previous steps.

The previous lists are useful for students to grasp the standardization as the essence of digital technology.

5.3 This lecture deduces the hardware models for the computer system

By definition, a computer system is capable of manipulating signals of any kind, hence the computer hardware has to include two genders of units that conform to **i)** and **ii)** respectively.

1st) - There are units that change the physical nature of data – namely they affect point **i)** – and are named *conversion* units. For example a printer transforms electric information into ink and makes a conversion; a keyboard change mechanical impulses into electric bits.

2nd) - The automatic *information-processing unit* manipulates the contents of data – see point **ii) –** and provides original representations of the reality. I introduce this concept by means of the following subtraction: **`250 - 50`** = **`200`**. Suppose that **`250`** means 'gross weight', **`50`** stands for 'tare weight', the outcome of the calculation **`200`** provides the 'net weight', namely the output-number illustrates a reality quite different from those represented on entry. Computation creates pieces of information with novel meanings. Automatic information-processing generates not only new numerical values, and even carries on new visual information, acoustic information, textual information etc.

The information-processing unit PR manipulates pieces of information that are uniform from the physical viewpoint, thus a computer system encompasses the information-process unit at the centre and several conversion units TR at the periphery to provide homogeneous messages in favour of PR.

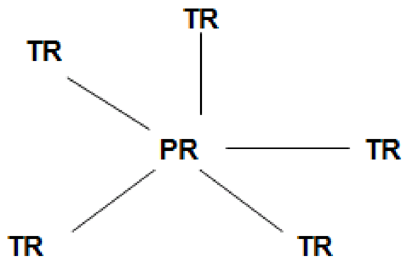


Figure 1 – Star Model

The central unit receives and delivers homogeneous information thanks to the peripherals specialized in conversion. Personal computers and laptops, embedded systems and mainframes comply with the radial model.

Different units need control and engineers complete PR with the *control unit* and make the *central-processing unit (CPU).* Because the cpu manages the whole system, it is easy to infer that the computer structure has a leader unit and a number of peripherals depending on the vertex. A hierarchical tree sums up this special quality of a computer system

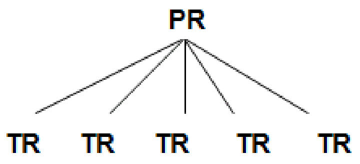


Figure 2 - Tree Model

5.4 This lecture extends the tree model to the monocentric networks

When a computer covers a large geographical area, experts move the peripheral device mile away and keep the hierarchy of the digital system. The peripherals placed all around the country take the name of *terminals* due to their subordinate role, while the computer system place at the vertex of the hierarchical tee acts as the *host* of such a *monocentric* or *hierarchical network.* All the operations lie under the control of the host that ensures secure communications amongst the nodes.

When the net is equipped with several hosts, the hierarchical order exhibited in Figure 3 cannot be preserved. No host has an edge over the others and the tree is impossible to establish. All the nodes rank the same level and the hierarchical control vanishes. The *polycentric net* has peer hosts and the Internet complies with this second scheme.

6. Analysis of the Results

IV) - The advantages that emerged in the educational experiment are commented as follows.

Comparative analysis of the contents: The present proposal does not suggest a 'way' or a 'method' to teach consolidated contents, but puts forward new contents which show the following advantages.

Traditional introductory lessons illustrate obvious topics such as the anatomy of a computer and frequently have the defect of mismatching with the variety of professions due to excessive mathematics and abstraction. Instead, the present research starts with general aspects of computing which are easily accepted from whatever perspective listed in Report of ITiCSE'97. Deductive learning of CS goes to the essence of things and in this way meets the requirements of the multiple perspectives on computing.

Comparative analysis of theoretical models: The Star Model clarifies the challenging to-day phenomenon called digital convergence. The Star Model is much more exhaustive than the IPO model popularized in current introductory lessons.

The Turing machine leads to the ethereal view of computer systems and to abstract reasoning, instead the present Star Model and Tree Model provide practical images of the computer hardware and even of networks and software programming (omitted here).

The Star Model and the Tree Model adhere to experience and do not require previous mathematical studies.

Comparative analysis of the deductive didactical method: Nowadays introductory courses show the computer system on the as-is basis and it may be said that current course force the mind of students to accept the facts without sufficient ground. Educational approaches which lack of deductive justification impose the contents and may be said dogmatic. Instead, the present lectures try to convince a student of the reasons that guide technicians' decisions, in this way the student's mind is not coerced and becomes progressively aware of the solutions created in the computer sector.

Currently there are thousands of electronic solutions which make a daunting challenge in education. Inferential reasoning overcomes this problem because it turns out to be very concise. In fact it is easier to learn a sequence of topics which are logically linked than to learn topics without any logical order. In the present work complex matters are introduced in brief – e.g. the properties of networks - whereas normally teachers waste a lot of time to illustrate those matters that do not have logical connections with the others.

Comparative analysis of students' psychology:

Students forced to acknowledge computer products on the as-is basis are poorly motivated, instead the present account absorbs students' attention. In fact a student is able to grasp a lesson provided he/she is familiar with the previous topics.

By definition deductive reasoning reveals the root-causes of technology and students take pride in the deductive logic. They become aware of understanding the most important sides of computer systems and the connections with other domains. I experienced that also students who rarely use computers - such as top-managers - showed interest in the present approach.

Listeners having the humanistic culture were pleased to discover how principles **i)** and **ii)** - shared in their own environment - lead to relevant aspects of the computer technology.

The history of computers integrates the foregoing lessons in a fair manner. The fortune of the hardware and software products completes the introductory lessons and sometimes provokes an emotional participation.

References

1 Report of the ITiCSE '97 Working Group on Historical Perspectives in Computing Education - Supplemental Proc. of the Conf. on Integrating Technology into Computer Science Education, Uppsala, 94 – 111, (1997).

2 P.J. Denning (ed) D.E. Comer, D. Gries, M.C. Mulder, A. Tucker, A.J. Turner, P.R. Young - Computing as a Discipline - Comm. of the ACM, 32 (1), 9-23, (1989).

3 D.J. Bagert - On Teaching Computer Science Using the Three Basic Processes From the Denning Report - SIGCSE Bullettin, 21(4), (1989).

4 J. Kramer – Is Abstraction the Key to Computing? – Comm. of the ACM, 50(4), (2007).

5 T.C. Lethbridge - Priorities for the Education and Training of Software Engineers - J. Systems and Software, 53(1), 53–71, (2000).

6 J. Gehl -The Future of the IT Profession: an Interview with Peter Denning - Ubiquity, 1(5), (2000).

7 P. Rocchi - Technology + Culture = Software - IOS Press, Amsterdam (2000).

8 P.Rocchi, L. Gianfagna - An Introduction to the Problem of the Existence of Classical and Quantum Information - Proc. Quantum Theory: Reconsideration of Foundations (QTRF-3), AIP vol. 810, 248-258, (2005).

9 R.Tocci, N.Widmer, G.Moss - Digital Systems: Principles and Applications - Prentice Hall, (2006).

Promoting Computer Science programmes to potential students: 10 Myths for Computer Science

Thanos Hatziapostolou, Anna Sotiriadou, Petros Kefalas

City College, Affiliated Institution of The University of Sheffield, 13 Tsimiski Str, Thessaloniki, Greece, {a.hatziapostolou, sotiriadou, kefalas}@city.academic.gr

During the last decade, our involvement with discussing with potential students and their parents before they apply for an undergraduate Computer Science programme, made us realise that there exist patterns in people's minds about Computer Science studies and profession. These patterns form misconceptions, which we identified as myths. In this paper, we present ten of them. We argue that these could be used as a promotion tool to attract potential students. Bearing in mind that Computer Science programmes all over the world have suffered a decrease in admissions, we believe that a good marketing policy that will lift public misconceptions about Computer Science will contribute to attracting more students to the discipline.

Keywords

Computer Science Education, Admissions, Promotion & Marketing CS programmes

1. Introduction

Admittedly, admissions in Computer Science (CS) or related programmes in Higher Education (HE) have dropped significantly over the last five years all over the world [1,2]. Although there has been considerable investigation for the reasons that lead to this decrease [3,4], and some of the results reported are intuitive, it is not always justifiable why potential HE students do not have CS anymore among their first choices.

There is a number of corrective actions that Universities and in particular CS Departments took or are planning to take in order to restore the popularity of their CS programmes [5,6]. Briefly, some of them are:

- Update the curricula by change of titles and content, so that prospective students identify (buzz-) words that look trendy;
- Enhance the curricula with courses that focus on industrial applications and business;
- Change the mode of delivery to make CS courses more fun to students and integrate ICT more into teaching and learning;
- Abolish or minimise as far as possible theoretical courses, while dressing them up with a more attractive wrapper;
- Apply their influence on secondary education teachers who might inform and guide their students;
- Introduce mutated programmes that include many elements from business studies, entrepreneurship and innovation;
- Promote and advertise programmes separately from others;
- Upgrading their web site promotion by presenting up-to-date staff and student achievements (research and industrial), figures of graduate recruitment, etc.

However, the above actions, although looked intuitive from the business side of view, were not always well received by academic staff.

In our Department at CITY College, Thessaloniki, we also took a number of actions similar to the ones listed above. As a result, some seem to work, some others do not, without, however, having gathered long-term data to qualify and justify the results. In addition to this, we are currently considering another way of promoting our programme with the objective to change the potential students' perception for CS. This approach is based on a number of misconceptions that secondary education students have about a CS programme and a CS professional. The aim is to list 10 myths for CS, argue against these and if possible convince the public to drop these misconceptions, which at large, constitute a preventing factor to follow such a programme, choosing what is perceived as "the easier way to graduate", for instance non-engineering programmes.

In this paper, we present these 10 myths for CS studies and profession in Section 2. Our intention is to initiate debate on these among fellow colleagues and finalise the list by strengthening our counter arguments. In section 3, we discuss our graduate students' perception on these in order to informally validate our approach. Finally, we discuss our Departmental promotion policy, which includes these misconceptions of the public, with the aim to attract more students to the discipline.

2. Ten Myths for Computer Science

Since the first cohort of Computer Science students in our Department, 15 years ago, we have been interviewing each one of the potential applicants. In Greece, it is common that no student applies for a programme in a private institution before they talk to the College administration and academic staff, requesting information on all aspects of education provision and career development. Although this may not be common for the rest of Europe, most of the times, candidates are accompanied by other family members, usually parents, who also participate in the discussion around the programme of study. Such meetings last between 30 and 60 minutes and are extremely important for both parties. For us, it is a perfect opportunity to find out what people think about the College and its programmes, and in the current context and culture, what people think specifically about Computer Science.

So far, we met multiple hundreds of cases, perhaps few thousands. At first, it was sometimes surprising to listen to people's opinions about Computer Science. The recurrent cases over the years made us suspicious that there might exist patterns in the public perception about CS. Our accumulated experience helped us in addressing in personal discussions some wrong impressions candidates and their parents have. Lately, we decided to form a list of such misconceptions and call them "*myths about CS*". We managed to compile a list of ten such myths (Table 1) and a set of counterarguments to use in interviews with potential candidates.

In the following, we present the myths and a brief analysis. Each one is accompanied by a counterargument as it is presented in our leaflets. Note that the wording and the analogies used in our promotion materials and brochures may sometimes look simplistic or even naïve or provocative but remember that these are just a marketing tool to address the public's misconceptions and help us carry out discussions with the average potential students and their parents.

Table 1 The Ten Myths about Computer Science in brief

#	**Myth**
1	Computer Science is sending emails, browsing the Internet, word processing and learning to use specific application programs
2	Computer Science is programming
3	Computer Science is maths, maths … maths!
4	Computer Science undergraduate studies restrict the choice for postgraduate studies
5	Computer Science jobs are boring, lonely and are all taken
6	Computer Science graduates never reach higher management positions
7	Computer Science studies is only for men
8	Computer Science, Information Systems, Computer Engineering, Computing … are all the same
9	Computer Science is not as important to society and business world as other disciplines
10	Computer Science is for “nerds/geeks”!

1 Computer Science is sending emails, browsing the Internet, word processing and learning to use specific application programs.

Lots of candidates ignore what CS is. Their encounter with computers is limited in secondary education and restricted to use of applications for word processing but mainly the Internet. If they are lucky they can get some good outline in the career development seminars among all other disciplines. Parents are in much worst situation. They keep on asking whether, as graduates, their children will “*have equal career opportunities to those having acquired ECDL*” or similar certification. Others, say that their children “*do not need CS because they were into computers every day since they were five years old*”, obviously meaning the encounter with email, games and chats on everyday basis.

We state in our promotion material: “*Anyone can do this! None needs to enter a HE programme in Computer Science in order to perform such tasks. In a similar way that none needs a HE degree in Mechanical Engineering or Electronics to drive their cars or use their TV sets. However, Computer Science is targeted towards developing such software, services and applications solutions so that other people can use them*”.

2 Computer Science is programming.

This comes from potential students rather than parents and it is the famous narrow view about CS. Potential applicants ask a lot of questions about programming languages, how many they are going to learn, in what depth etc. They are surprised by the fact that the number is much less than expected, because they cannot imagine what else they can do through the three years of study. Occasionally, a similar misconception is brought up, but this time with respect to hardware, that is, “*CS is about fixing PCs when they break*”. Especially, parents would love to see their kids fixing something, because most of them cannot understand programming.

We state in our promotion material: “*Again, most people can do this with a bit of reading and practice! Programming is a technical skill for which someone does not need a degree to acquire. Should you be an engineer to fix a leaking water pipe in your place? Someone with an engineering degree knows everything about materials and hydraulics including perhaps hands-on practice with tools. Programming is just a tool for Computer Scientists. Current complex software systems require software engineering methods, methodologies and approaches*”.

3 Computer Science is maths, maths … maths!

There are two categories. The usual math-phobic question: "*Is maths involved? Can I make it with the maths I know so far?*" and the math-alibi statement: *"I am not good in maths and therefore I am not interested*". Both come as a result of inadequate knowledge of what maths is and what is good for. In high school and lyceum, students did, admittedly, a lot of maths, without however being able, or have time, to appreciate it. In an intensive examination system, teachers prefer to get students going with maths rather than engaging on philosophical discussions.

We state in our promotion material: "*Not really! Maths in a Computer Science programme is specific to this discipline and is taught almost from point zero. It is not just maths for maths. Yes, we do need maths, because we are required to establish correctness and soundness of the applications developed. This is much alike the way that mathematics guarantees that a newly designed airplane is safe to fly without killing people before we spend millions in constructing it. Maths is what makes it a Science!*".

4 Computer Science undergraduate studies restrict the choice for postgraduate studies.

People get the wrong impression that the only postgraduate studies they can pursue are those relevant to Computer Science, with emphasis in one of CS areas, such as networking, software engineering, artificial intelligence etc. They seem surprised to hear that the horizons for postgraduate studies are wide open to other programmes from different disciplines. Some, they have not even thought of such possibility and although they do not believe it, they look happy from such prospect.

We state in our promotion material: "*To the contrary, a CS degree opens a wide range of potential! Actually, it is evident that graduates who follow Master's degrees in other, even unrelated, disciplines like Management, Music, Politics etc. are extremely successful because they possess the fundamental intellectual skills and long-life learning abilities that help them towards such an attempted conversion*".

5 Computer Science jobs are boring, lonely and are all taken.

Boring comes together with lonely as a result of a stereotype developed in people's perception about a person who spends most of its day and night time in front of a PC. The idea that all jobs are taken comes from the general impression about unemployment figures. CS related job offers, although from time to time affected by the general socio-economic trends, are still and will be in abundance. It is not always easy to convince parents when discussion comes to careers and employability, but few good examples of graduates' successful career path could ease their concerns.

We state in our promotion material: "*New problems and new needs by the society and the industry never leave space for boredom because new solutions must be devised and offered. The complexity of the problems addressed is such that none can manage alone. Computer Scientists more than any other related discipline need to work in well-structured teams. The need is increasing because demands for automated solutions are increasing. During the last decade, Computer Scientists are highly employed. The demand is still constantly high. With the current trends of the market, it is predicted that Computer Scientists would be hard to find*".

6 Computer Science graduates never reach higher management positions.

Again the stereotype of a “*hard working developer wearing t-shirt and jeans*” is the source of this myth. In people’s mind, CS professionals are there to be instructed what to do from top level tie-dressed managers, without arguing much, they work at a dark room with PCs and junk food all over the place because there is nothing else they are able to do. However, there is plethora of examples of Computer Scientists and Engineers in general, who make high management positions due to their skills and pragmatic approach to businesses. This is also supported by the fact that no small proportion who choose to do an MBA at their late thirties, are top executives in their companies with a CS or engineering background.

We state in our promotion material: “*Computer Scientists acquire such organisational and communication skills that are highly suited for high management positions. The disciplined ways in which they face problems and engineer solutions make them able to undertake and successfully cope with many managerial tasks. In addition, IT has become so important for businesses that Computer Scientists are directly involved in decision making for the future development of businesses*”.

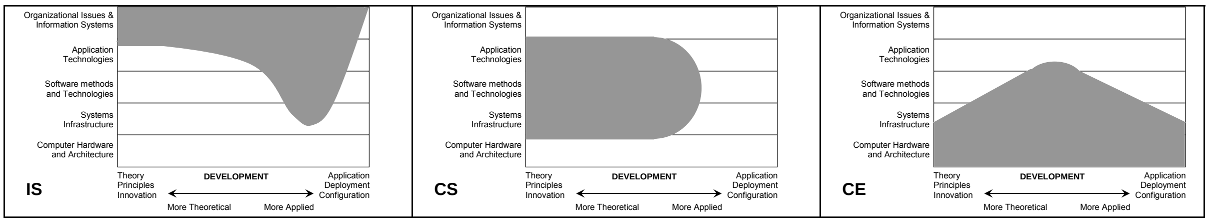
7 Computer Science is only for men.

While CS always attracted more men than women, the last ten years the gender gap is expanding [1] and fewer young women are entering the discipline every year. The fact that CS does not seem to appeal to young women originates from the distorted image of the computing career and the misconceptions that programming is a solitary activity and that jobs are boring and lonely. Young women believe that they will be alone inside a room writing code all day long or talk to others with acronyms; they simply can not imagine themselves doing these and inevitably become a “geek”. Furthermore, although women are creative and innovative by nature, they have the misconception that CS does not require such skills and as a result they do not pursue a career in this area.
We state in our promotion material: “*It is true that men have outnumbered women in computer science in the past, but this is changing. Increasingly, women are becoming extremely successful professionals. On average they might even do better than men! Computer Science is about helping others solve problems, learning about new ideas, face challenges, dreaming up new situations, products, and ideas. Contemporary women contribute to all the above in an innovative way*”.

8 Computer Science, Information Systems, Computer Engineering, Computing are all the same.

There is a tendency to confuse these. Especially in countries where “*Computer Science*” and “*Informatics*” basically mean the same thing, while “*Science*” and “*Engineering*” are used as synonyms. The ACM/IEEE CS Curricula [7] helped a lot in trying to differentiate these with the figures provided in table 2 below.

Table 2 Differentiation between Computing Programmes [7]



We state in our promotion material: "*Computer science spans a wide range, from its theoretical and algorithmic foundations to cutting-edge developments in robotics, computer vision, intelligent systems, bioinformatics, and other exciting areas. Information systems focus on integrating information technology solutions and business processes to meet the information needs of businesses and other enterprises, enabling them to achieve their objectives in an effective and efficient way. Computer engineering is concerned with the design and construction of computers and computer-based systems. Although may share common grounds, all three are distinct from each other*".

9 Computer Science is not as important to society and business world as other disciplines.

While the number of occurrences that this specific misconception was revealed during interviews has decreased during the last three years, it still does exist. The primary reason that contributes to the formation of this misconception is the belief that CS is concerned with sending emails, browsing the Internet, word processing etc. As a result, candidates and parents who lack understanding of the area justly fail to see the importance of the discipline since they can not realize how the use of such application programs improves quality of life.

We state in our promotion material: "*Are the following important for the society and business world? Safe driving and flying? Privacy in communications? Correctness and integrity of sensitive business and personal data? Safety of bank accounts, anytime-anywhere access and management of knowledge? Reduction of cost in business operations? Effective health services? If so, then Computer Science is at least as important to society and businesses as other disciplines*".

10 Computer Science is for "nerds/geeks"!

Mostly used in secondary education, the stereotype of a "nerd/geek" refers to a student who is highly competent especially in traditionally difficult courses such as mathematics and physics. As a result, anyone who believes that CS is about mathematics and algorithms assumes that only such students can attend and succeed in computing courses. In addition, the stereotype of a "nerd/geek" is presented in many movies as someone who is awkward, without any social life and with friendships limited to other "nerds/geeks". As a result, other students may be strongly discouraged to attend a discipline in which they will have no 'fun'.

We state in our promotion material: "*This is a distorted image that Hollywood films impose to the general public. Computer Scientists seem to talk to each other with some undecipherable technology terms but this is no different from what Doctors or Engineers or Lawyers or Philosophers do. Computer Scientists are trained to acquire good communication skills and general knowledge that will help them to interact with people for whom they provide solutions*".

Interestingly enough, the vast majority of the people met with staff were not concerned with research at all. This is somewhat disappointing for academic staff that spend considerable amount of time in conducting research with, among others, the aim to enrich teaching and learning with up-to-date developments. It is something that the public ignores its importance or at least it seems that it is not one of the first priorities in choosing CS programmes.

3. The perception of final year CS students

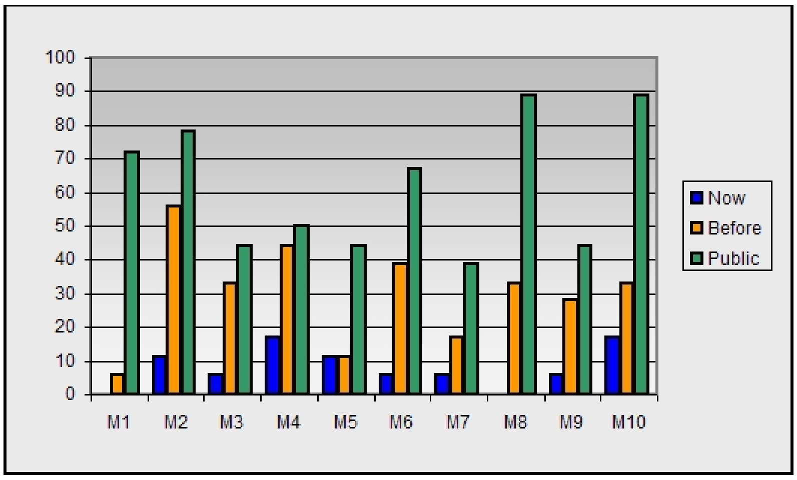
As said above, there is strong evidence that these myths do exist, as thousands of discussions held over the last decade with potential candidates and their parents revealed.

We thought of asking final year students about what they think about these myths. This is by no means a way to validate the myths stated in this paper. However, we thought that by asking our students what they believed before they started their studies and what when they are about to graduate, we can demonstrate that some (or all) of these myths do exist.

We conducted a short survey on our final year undergraduates by distributing a questionnaire with these 10 myths and asking them to express their beliefs in order to consolidate our suggestions. Furthermore, we also thought that it would be interesting to see their opinion (view) as to what current high school seniors believe about the 10 myths. Twenty (20) students completed the questionnaire the results of which are presented in figure 1 below.

Figure 1 Questionnaire Results

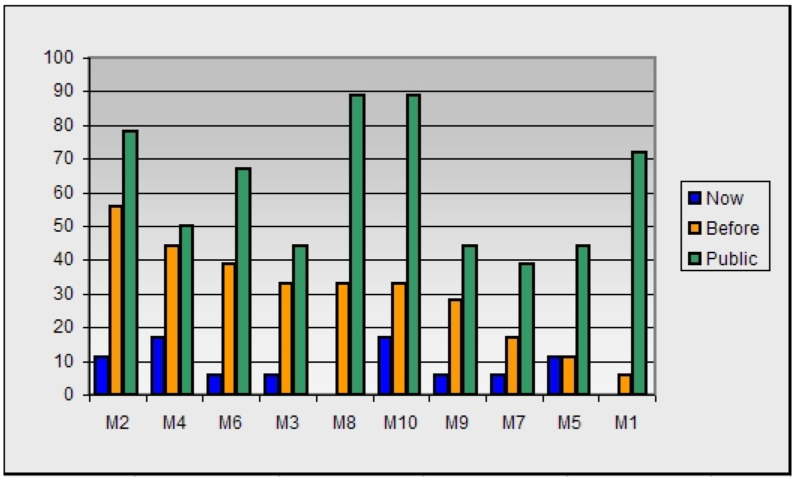
	YES (%)		
	Now	Before	Public
M1	0	6	72
M2	11	56	78
M3	6	33	44
M4	17	44	50
M5	11	11	44
M6	6	39	67
M7	6	17	39
M8	0	33	89
M9	6	28	44
M10	17	33	89



In the above figure, the first column of the table and the first bar of graph represent the percentage of the students that agree with the 10 myths just before graduation whereas the second column and the second bar denotes the percentage of the students' who agreed with each myth before they started their studies. The third column and the third bar depict what the students think that current high school seniors think. Figure 2 below presents the same results but sorted by popularity of the myths before our senior students started their studies.

Figure 2 Questionnaire Results Sorted by popularity of Myths

	YES (%)		
	Now	Before	Public
M2	11	56	78
M4	17	44	50
M6	6	39	67
M3	6	33	44
M8	0	33	89
M10	17	33	89
M9	6	28	44
M7	6	17	39
M5	11	11	44
M1	0	6	72



While the results of the questionnaire are simple to understand, a few key indications are the following:

- our students did have some misconceptions regarding the CS field before they started their studies but after three years almost all myths were significantly dropped
- the second myth was the strongest one and 56% of our final year students thought that 'CS is about programming' before they started their studies
- a high percentage of the students believe that current high school seniors have the wrong perception about the field of computer science and the CS profession with myths 10, 8, 2 and 1 being the strongest ones
- unfortunately, it seems that students still believe that CS jobs are boring, lonely and are all taken. This is expected from students that have not yet entered the marketplace but also, that perhaps the market itself gives such an impression to potential employees.

We also conducted the McNemar test for each of the ten myths in order to see the actual significance of change in the views of the students. The McNemar test compares paired samples and tests the significance of their difference. The results revealed that there is a significant change in the number of students who changed their perception before and after their studies for myths 2, 3, 6 and 8 with significance levels *p* 0.021, 0.031, 0.031, 0.031 respectively.

4. Conclusions

During the last five years, the Computer Science discipline is facing a crisis mainly due to the great decline in student enrollment. Having acknowledged this problem computer scientists and computer science educators conduct investigations in order to pinpoint the reasons that CS is not appealing to students any more and to propose solutions that will restore the popularity of the field. According to these investigations, one of the major sources of the problem is that prospective students and the public in general have a distorted view about the discipline itself, what a computer professional does and a computing career in general. After fifteen years of interviews with potential applicants of our undergraduate CS programme and their parents, we were able to determine a number of misconceptions that exist. In this paper we present these misconceptions as myths about CS studies and profession. Our intention is not to present a finalised list of prevailing misconceptions but to launch a discussion on these myths among fellow colleagues in order to strengthen our arguments. We believe that this approach can facilitate the marketing plans of CS departments that would like to promote their programmes to potential students. As part of our promotion of our undergraduate CS programme, we have created a leaflet entitled "10 Myths about Computer Science". This leaflet presents and then drops these misconceptions. The leaflet is disseminated along with the other promotional material, such as examples of our graduates who today have a lot of success in different fields and countries, such as e-learning, security, telecommunications, etc., to anyone who seeks information about the specific programme of study. We have also written articles in local newspapers and finally we are planning to develop a web version of the myths which will be added to the department's web site.

References

1 Vegso, J. Interest in CS as a Major Drops Among Incoming Freshmen. Computing Research News 17, 3 (May 2005); www.cra.org/CRN/articles/may05/vegso.

2 Foster, A. Student interest in computer science plummets. Chronicle of Higher Education 51, 38 (May 27, 2005), A31.
3 McGettrick, A., Boyle, R., Ibbett, R., Lloyd, J., Lovegrove, G., Mander, K. Grand Challenges in Computing Education. BCS, 2004, ISBN 1-902505-63-8
4 Patterson, D. Restoring the popularity of computer science. Commun. ACM 48, 9 (Sept. 2005), 25–28.
5 Denning, P., McGettrick, A. Recentering Computer Science. Commun. ACM 48, 11 (Nov. 2005), 15–19.
6 Klawe, M., Shneiderman, B., Crisis and Opportunity in Computer Science. Commun. ACM 48, 11 (Nov. 2005), 15–19.
7 Computing Curricula 2005. ACM, IEEE-CS, AIS. http://www.computer.org/portal/cms_docs_ieeecs/ieeecs/education/cc2001/CC2005-March06Final.pdf

Working for a leap in the general perception of computing

Angelo Lissoni[1], Violetta Lonati[2], Mattia Monga[3], Anna Morpurgo[2], Mauro Torelli[2]

[1]Kangourou Italia, Via Cavallotti 153, 20052 Monza (MI), Italy, lissoni@kangourou.it

[2]Dipartimento di Scienze dell'Informazione – Università degli Studi di Milano, Via Comelico 39/41, 20135 Milano, Italy, lonati, morpurgo, torelli@dsi.unimi.it

[3]Dipartimento di Informatica e Comunicazione – Università degli Studi di Milano, Via Comelico 39/41, 20135 Milano, Italy, monga@dico.unimi.it

The name "informatics" is often associated with the set of skills necessary to use specific software applications and not to the discipline itself. We believe it is urgent to change this misperception, as it has negative impacts. To this end, we started a project aimed at spreading the discipline of informatics among school children through a game-contest called *Kangourou*, that has a successful tradition in the field of mathematics. This paper reports the first steps of this initiative: we present a concrete proposal, providing some examples of the content and the formulation of the questions proposed and reporting the initial feedback obtained by testing them in pilot classes.

Keywords
Computing principles, game-contest, primary education.

1. Computer science and informatics

The debate about the role of computers in the science of informatics is an ancient one. The popular expression *computer science* has upset several scholars: Dijkstra [1] compared its use to naming surgery the knife science or astronomy the telescope science; others accept the idiom just as a relic of its historical roots, as descriptive as the etymology of the word *geometry* (Greek *γεωμετρία*: geo = earth, metria = measure). All in all, among the experts of the field the question is almost settled: even those who use the term *computer science* – as common in the US – are normally quite convinced that the topic concerns the computation process, rather than the computational devices that have made it possible and convenient. Actually, the design of efficient computational devices remains an exciting research issue, but it is more in the scope of computer engineering than informatics. Also, informatics should be distinguished from the study of computer systems and their deployment, a subject that should be more precisely called *information systems*.

Although well established in the field, these distinctions are much less clear to the general public. Furthermore, the three areas of informatics, computer engineering, and information systems, obviously connected with one another, have little to do with the skilled use of a bunch of specific applications: in fact, being fluent in using a given computer application is largely independent from the knowledge one may have in the aforementioned disciplines. In other words: to be able to read a clock one needs virtually no knowledge about the laws of pendulum.

Another interesting remark about the various interpretations of the term informatics is due to Claudio Mirolo [3], one of the promoters of the Task Force for the Research on Teaching of Informatics at the University of Udine, who identifies at least three possible acceptations, corresponding to different cultural approaches:

1. informatics as a s*cience,* providing its own peculiar key to interpret reality and its specific approach to problem solving;
2. informatics as a t*echnology*, concerning the characteristics, structure and working principles of the now ubiquitous hardware and software devices;
3. informatics as an *instrument,* providing practical tools to manage information in many different contexts.

For the sake of clarity, in the following we will adopt the term *applimatics* to denote the use of specific applications and *computing* to denote the field of computer engineering, informatics, and information systems. While the latter term is widely accepted, the former is coined ad-hoc: we actually think that the use of a distinct term may help in reducing the ambiguity affecting the subject.

Getting some skills in applimatics can be very useful. For example, the "European Computer Driving Licence" (ECDL) [6] initiative has successfully contributed to the spreading the basic literacy of office automation tools among millions of people, who certainly took advantage of that knowledge. ECDL is a certification program based on seven different modules:

1. *Concepts of Information Technology*
2. *Using the Computer & Managing files*
3. *Word Processing*
4. *Spreadsheets*
5. *Databases*
6. *Presentations*
7. *Information & Communication*

As a matter of fact ECDL teaches very little about sheer computing (virtually only the first module concerns what we consider computing) and the acquisition of basic computing concepts is often perceived as an unwanted overhead by ECDL candidates.

This is the symptom of a general misperception: whereas everybody feels it is important to have a basic knowledge about word-processors and web browsers, an understanding of computing is often considered a special domain knowledge to be acquired only by experts of the field, since it is believed to have no immediate interest or usefulness in the real world. This opinion is supported also by many educated people, as shown by the fact that the conceptual contribution of the science of computing to other disciplines (such as cognitive science, economics, mathematics, physics, and linguistics) is seldom acknowledged. However, we believe some peculiar aspects of computing are sufficiently basic to be taught as a fundamental formative subject. For instance, consider:

- the focus on the precise description of objects, processes, and protocols;
- the management of complexity through encapsulation and reuse;
- the synthetic power introduced by the constructs of formal languages;
- the flexible use of abstractions, that can be dynamically coerced to what is more useful in any given moment, as in the case of data used as instructions and vice versa.

As it is clear to all the people working in computing, the misperception of informatics has negative impacts: brilliant students tend to be attracted by other sciences because they are not familiar with the challenges of our discipline, freshmen in computing courses sometimes have distorted expectations, public funding of basic computing research is hard to raise, etc. What can be done to change this matter of fact?

Our first and most fundamental suggestion is that our research community has to undertake a cultural battle to clarify the difference between computing and applimatics, and to disseminate the root principles of computing, starting from children education.

As far as education is concerned, we complete the picture by reporting some simple facts about the teaching of informatics in Italy. Recently, in its "Curriculum directions for K-12 education" [2], the Italian Department of Education (MPI) has indicated "Information Technology" as a means to increase the communicative power of students. Thus, the introduction of information technology tools is encouraged in all subjects (from arts to science) in order to "expand the space, the time, and the mode of social interaction and experience". Even technology teachers seem to be often predominantly interested in the fact that computers may be used to process texts, images, and other multimedia content, or to provide communication facilities. And, although MPI mentions that most of today technological artifacts have to be operated by signals and instructions, the mastering of this kind of issues seems to be suggested more as a service to higher level tasks than as an intellectual challenge in itself.

The actual implementation of MPI's directions is even conceptually poorer, since schools lack resources to acquire information technology means and teachers are rarely competent in computing. In the practice of teaching, at least in Italy, the scientific aspect of computing is virtually absent and the term “Informatica” refers prevalently to what we have called applimatics. Paradoxically enough, in the '80s, the rare pioneers that experienced the teaching of “Informatica” intended it in the sense of computing, by proposing didactic initiatives mostly connected with programming (for instance through the *Logo* language, cf. e.g. [4]). Such initiatives are today considered outdated by most teachers, more attracted by many appealing and user-friendly applications. However, these applications convey very little about computing, indeed they might even obscure the interesting computational and algorithmic aspects of such tools. We are convinced, on the contrary, that the awareness of these intrinsic features may be essential to a critical, profitable and mature use of computing. Vice versa, by overlooking these features, there is a danger that these tools, which are by now ubiquitous and pervasive, may be perceived in some sense as mysterious and indistinguishable from magic.

To cope with this situation, we started a project to introduce children to computing through a game-contest called *Kangourou* that has a successful tradition and a well-established experience in the field of mathematics. This paper reports about the first steps of this initiative and is organized as follows: in Section 2 we briefly present the Kangourou, in Section 3 we describe our current proposal for a Kangourou of informatics, and in Section 4 we draw some conclusions stemming from our first experiences.

2. The Kangourou game-contest

A game-contest, the *Kangourou des Mathématiques*, was created in 1991 in France by André Deledicq on the model of the *Australian Mathematics Competition*, with the goal of contributing to the popularisation and the promotion of mathematics among young people.

The success was immediate also thanks to the associated distribution of a massive and pleasant documentation on mathematics to the participating pupils and their teachers.

The French experience was exported abroad, first to Europe and then to other continents through an international association, *Kangourou sans frontières*, founded in France in 1995.

The association's aim is to promote the spreading of a basic mathematical culture by all means and, in particular, by organising the annual game-contest to be held on the same day in all participating countries. The game, whose intent is to attract the maximum number of pupils without aiming at any national selection nor at a comparison between countries, has had a great success and it now counts millions of participants among elementary and secondary school kids (47,000 in Italy in 2008).

In conjunction with the contest, and under the trademark *Kangourou*, books on mathematical games, brochures on mathematical dissemination to the general public, documents and software are realised and widely spread, meetings and exchanges between children and between teachers, colloquia, and training periods are organised.

In Italy, which joined the association in 1999, the game is organised in cooperation with the Mathematics Department of the Università degli Studi di Milano [5].

As a consequence of the effectiveness of the event, the game-contest was extended to other disciplines. In Italy, in cooperation with the British Institutes and the patronage of the Università degli Studi di Milano and La Sapienza of Rome, the Kangourou for English as a second language was created two years ago, which saw 11,500 participants in 2008. A team of members from two Informatics Departments of the University of Milan, AICA and SDA-Bocconi is now studying a formula for an informatics game-contest. In particular the following issues must be defined:

- the cultural goals,
- the way the game-context should be carried out,
- the content and formulation of the questions,
- the ages of the participants.

3. The ambitions of the Kangourou of informatics

The Kangourou of informatics might offer, to both pupils and teachers, a correct view of informatics and the oppurtunity to face the actual nature of computing, with particular regard to scientific aspects of informatics, often unstressed in school syllabi. The main tool is, of course, play. In fact, in primary schools play may have a strong educational valence, while the competitive feature is utterly subordinate.

3.1 A concrete proposal

At present our proposal is conceived as follows.

1. The organization should be similar to that of the Kangourou of Mathematics. An individual competition to take place in the classes, followed by a national final for the best competitors. The questions of the first stage will involve multiple choice answers, in the style of the other Kangourous, without using computers, since that would severely complicate this stage. In the final competition, however, we expect to propose open questions which might require the use of computers or specific programs.
2. For the first edition of this Kangourou we only expect to involve classes in secondary schools (11-14 year old pupils), but the idea is to extend the competition to primary schools (6-11 year old pupils) as soon as possible.
3. The subject of study of informatics is actually such a vast one that it is not easy to determine which contents and levels of deepening should be suitable for young pupils. Moreover, no definite programs exist for the subject and the choice of topics is left to the teacher. For this reason a first set of test questions has been proposed to pilot groups of pupils and teachers with the aim to collect their feedback and reactions.

3.2 Contents and formulation of the questions

The aim is to present the questions in a playful form, by creating fanciful contexts, and to make topics and problems accessible even without previous experience with informatics.

One has to tackle technical terminology, reference to computer components, codes and specific representations of information, jargon from signal processing, cryptography, data structures (trees, graphs and so on), primitives and composition rules, algorithm representation, execution and complexity, recursion, sorting and searching, automata, languages and grammars.

Presently, only a few classes have already experienced our proposal but we can testify the first reactions of teachers and pupils were quite different. Teachers often appeared worried: the questions proposed often dealt with completely unknown topics and they felt they would not have been able to answer; on the other hand, even if students considered many questions difficult too, they appeared to be less scared and more curious.

As an example let us consider a couple of the questions proposed in the test set.

> *Santa Claus has prepared a few gift parcels having different colors: red, yellow, blue, and he has put them in two stores, mixing colors. Now he needs to know how many red parcels he has stored. He has got some elves to help him, but each elf only knows how to perform one operation and moreover Santa can choose only three of the following elves.*
>
> ***Arvo*** *moves blue parcels from a store to the other.*
>
> ***Bjork*** *moves red parcels from a store to the other.*
>
> ***Ceula*** *moves parcels from a store to the other but he is color-blind.*
>
> ***Dino*** *counts the parcels in a store.*
>
> *Which elf will Santa NOT choose?*

In this specific case, answering the question requires the ability to realize a complex plan by composing a few basic operations. These abilities may be considered of a logical or mathematical type, however the existence of explicit constraints endows the problem with some typical computer science features. We estimated such a question to be quite difficult for the audience, however, surprisingly enough, students declared it to be quite easy. Actually, it turned out that very few of them were able to answer correctly, recognizing that the unneeded elf is Arvo; on the contrary, almost all of them excluded Ceula. Clearly, this means they did not build a complete solution to deduce their answer, but probably followed this wrong shortcut: keeping the more specialized elves and exclude the more generalist one.

Another example is the following.

> *Philip needs to choose a password to protect his e-mail. Which of the following passwords ensures greater security?*
>
> 1. *Philip1995* [adding his birth year].
> 2. *Ph1l1p* [changing a few letters into numbers].
> 3. *PhiLiP* [using some capital letters].
> 4. *Philipemail* [to remember what the password is for].
> 5. *TpitfIto!* [the initials in the sentence “This password is the first I thought of!”].

In this case the aim is to convey attention to a basic operation common to most computer activities – the choice of reliable credentials – which much too often is made with dangerous superficiality. According to the first feedbacks, this question actually seems to be quite easy for young students, probably more acquainted with such issues and less naïve than we would think.

4. Conclusions

Notwithstanding the ubiquity and pervasiveness of the results of the science and craft of computing in everyday life, the impact of its conceptual roots on modern thinking is still unclear to the general public. The success of the computational approach is often confused with the popularity of successful applications and the need for mastering computing principles is mixed up with the skills needed to use a given system proficiently. To cope with this situation we think it is urgent to change the misperception of computing, by introducing children to its root principles as soon as possible. To this end we started to study how a game-contest as the Kangourou could help in fostering the interest about computing through amusement and play. Our first experiments are promising, although, as expected, the major obstacles seem to stem more from the misunderstanding of the multiple facets of computing on the side of adults rather than from a lack of interest in informatics on the side of school pupils.

References

1 E. W. Dijkstra. On a cultural gap. *The Mathematical Intelligencer*, 8(1):48-52, 1986.

2 Ministero della Pubblica Istruzione. *Indicazioni per il curricolo per la scuola dell'infanzia e per il primo ciclo* d'istruzione. Tecnodid Editrice, Napoli, 2007.
http://www.pubblica.istruzione.it/normativa/2007/allegati-/dir_310707.pdf

3 C. Mirolo. Quale informatica nella scuola? 2003.
http://nid.dimi.uniud.it/pages/materials /discussion/educazione.pdf

4 J. Muller. *The Turtle's Discovery Book!* Harvard Associates, 1996. Italian translation by Silvia Gallina, Laura Menicagli, Guido Ramellini. http://www.tiziana1.it/ebooks/Risorse/TDBitv1.pdf

5 http://www.kangourou.it (Italian web site) and http://www.mathkang.org (official site of the French association)

6 http://www.ecdl.org

Tools that support contribution-based pedagogies

Paul Denny, John Hamer, Andrew Luxton-Reilly[1]

[1]*The University of Auckland, Private Bag 92019, Auckland, New Zealand, {paul, j.hamer, andrew}@cs.auckland.ac.nz*

The "contributing student" approach, introduced by Collis, turns the student from passive consumer of information to an active and engaged co-creator of resources for others. While the sharing of student produced content may be handled manually in small classes, tools are required to effectively support this approach in large classes. We report here on our experiences with, and student perceptions of, the tools we currently use to support contribution-based pedagogies.

Keywords
Aropä, Contributing student, Contribution-based pedagogies, Peer assessment, Peer review, PeerWise, Self assessment

1. Introduction

Contribution-based pedagogies require students to engage in activities that involve the creation and sharing of learning resources that are used by other students [1] [2]. This contribution is typically (although not necessarily) peer reviewed. A number of significant benefits arise from their use. Higher-order cognitive processes such as evaluation, reflection and critical thinking are emphasized. Communication, teamwork and self-assessment, skills integral to effective operation in knowledge economies, are developed, building a foundation that supports lifelong learning. Students are transformed from passive receptors of information to active and critical members of a community engaged in the process of constructing knowledge.

Wenger, McDermott and Snyder [3] describe communities of practice as "groups of people who share a concern, a set of problems, or a passion about a topic, and who deepen their knowledge and expertise in this area by interacting on an ongoing basis". A focus on student-driven learning by emphasising contribution and feedback helps to form a community of practice amongst the student population, in which students learn from and with each other. Brookfield [4] notes that peer learning is crucial for success. Students learn from their peers through advice, information and skill modelling. He states: "The learning activities of successful self-directed learners are placed within a social context, and other people are cited as the most important learning resource." (p. 9).

Contribution-based pedagogies may have additional significant benefits for female students. Barker et al. [5] suggest that retention of women in Computer Science may be improved by creating a classroom culture in which learning is a social or community practice rather than a solitary one, and Cohoon [6] recommends more interaction among classmates and the development of learning communities.

The information age is characterized by a massive amount of rapidly changing information distributed across wide geographic and information spaces. Learning in the information age *“increasingly requires 'learning to participate' in social learning systems”* [7]. Birenbaum [8] states:

> *… successful functioning in this era demands an adaptable, thinking, autonomous person, who is a self-regulated learner, capable of communicating and cooperating with others. The specific competencies that are required of such a person include*
>
> *a) cognitive competencies such as problem solving, critical thinking, formulating questions, searching for relevant information, making informed judgements, efficient use of information, conducting observations, investigations, inventing and creating new things, analysing data, presenting data communicatively, oral and written expression;*
>
> *b) meta-cognitive competencies such as self-reflection and self-evaluation;*
>
> *c) social competencies such as leading discussions and conversations, persuading, co-operating, working in groups, etc. and*
>
> *d) affective dispositions such as for instance perseverance, internal motivation, responsibility, self-efficacy, independence, flexibility, or coping with frustrating situations (p. 4).*

Students must develop the skills that enable them to operate effectively in this environment if they are to be successful. A radical shift in pedagogy is required to prepare students appropriately. According to Biggs [9], good course design should focus on student learning, and activities should align closely with the desired outcomes. As teachers in higher education, we are responsible for designing assessments and activities that help students to develop the ability to:

- work independently,
- filter large amounts of information,
- critically evaluate the quality of information,
- act as part of a community, and
- use online tools to communicate effectively.

Peer- and self-assessment activities are used to help students develop exactly these kinds of attributes [10]. Stefani [11] states that if we want our students to be autonomous, reflective and independent, then our assessment practices should include these qualities. Boud [12] argues that self-assessment is central to effective learning and that students should be making complex judgments about the criteria for good practice in a given area. Furthermore, courses that do not encourage self-assessment can actually undermine lifelong learning [13]. The literature on both self-assessment [14] [15] and peer assessment [16] report numerous benefits for students, including:

- help to consolidate, reinforce and deepen understanding, by engaging students in cognitively demanding tasks: reviewing, summarising, clarifying, giving feedback, diagnosing misconceptions, identifying missing knowledge, and considering deviations from the ideal;
- highlight the importance of presenting work in a clear and logical fashion;
- expose students to a variety of styles, techniques, ideas and abilities, in a spectrum of quality from mistakes to exemplars;

- provide feedback swiftly and in quantity. Feedback is associated with more effective learning in a variety of settings. Even if the quality of feedback is lower than from professional staff, its immediacy, frequency and volume may compensate;
- promote social and professional skills;
- improve understanding and self-confidence; and
- encourage reflection on course objectives and the purpose of the assessment task.

In smaller classes with few students, contribution based pedagogies can be successfully employed manually. However, additional support is required in large classes to handle the workload associated with the distribution and management of student contributed resources.

We have developed two web-based eLearning tools that facilitate meaningful student contribution and collaborative learning. Using our tools, students can share and comment on each other's documents, practice peer assessment, and develop demonstrably valuable learning resources with minimal effort required from academic staff.

- Aropä [17] has been developed to support peer assessment as a routine coursework activity.
- PeerWise [18] has been designed as a tool to help students revise course material by generating, sharing, evaluating and practising self-test questions in a collaborative environment.

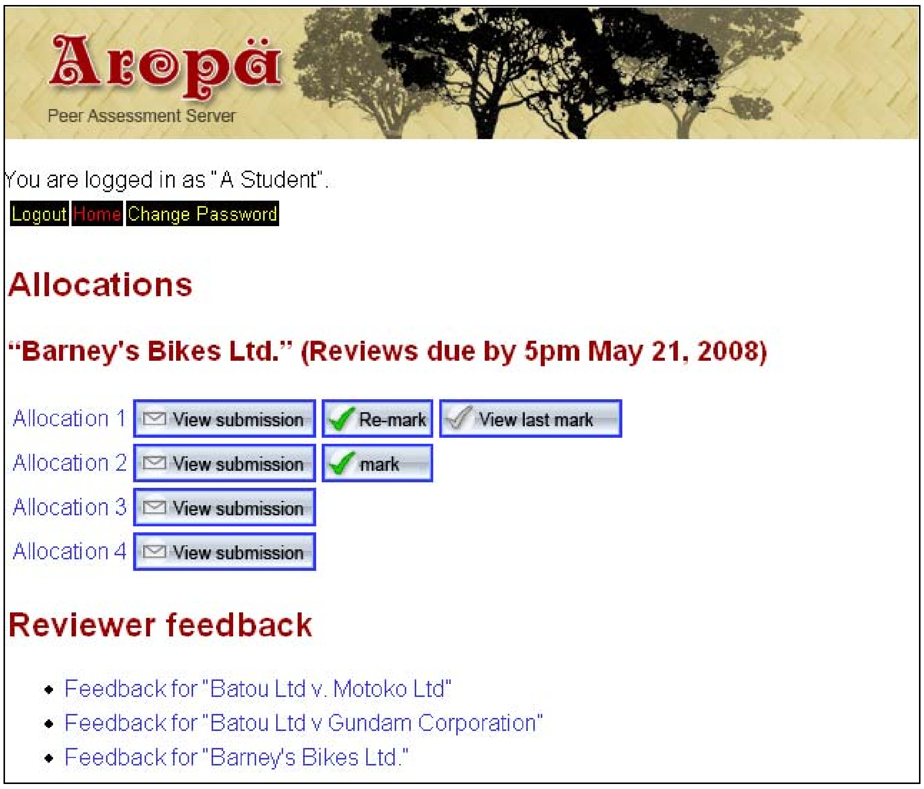
In this paper, we report on the tools that we use to support regular, large-scale self- and peer- assessment. Our tools have been used in over a dozen courses and have involved more than 1,000 students each semester over the last four years.

2. Aropä

Aropä was developed as a tool to manage the administration of peer assessment activities. It removes the need to manually distribute and collect paper copies of student work. Documents to be reviewed are uploaded online, and downloaded for viewing by the appropriate reviewers. Reviews are entered through a web form that can be fully customised by the instructor. Aropä also manages the allocation of reviews, calculation of (weighted) average grades, and an assortment of other administrative tasks.

Our main interest in creating Aropä was to make peer assessment possible as a routine activity even in large, undergraduate classes. In this, Aropä has been very successful, with over 1,000 students using it each semester. Aropä has been in use since 2004 by classes in Academic Practice, Business, Civil Engineering, Commercial Law, Computer Science, English, Electrical Engineering, Environmental Science, Information Management, Medical Science, Pharmacology, and Software Engineering. The class sizes range from 12 to 850.

For a student, using Aropä involves three steps. First, she uploads a document (typically a report, computer program, essay, etc.) through a standard web file upload page. The second step is to read and review the allocated submissions. Finally, she reads the feedback provided by her anonymous reviewers. A snapshot of the main interface is shown in Figure 1.


Aropä

Peer Assessment Server

You are logged in as "A Student".

Logout Home Change Password

Allocations

"Barney's Bikes Ltd." (Reviews due by 5pm May 21, 2008)

Allocation 1 View submission Re-mark View last mark

Allocation 2 View submission mark

Allocation 3 View submission

Allocation 4 View submission

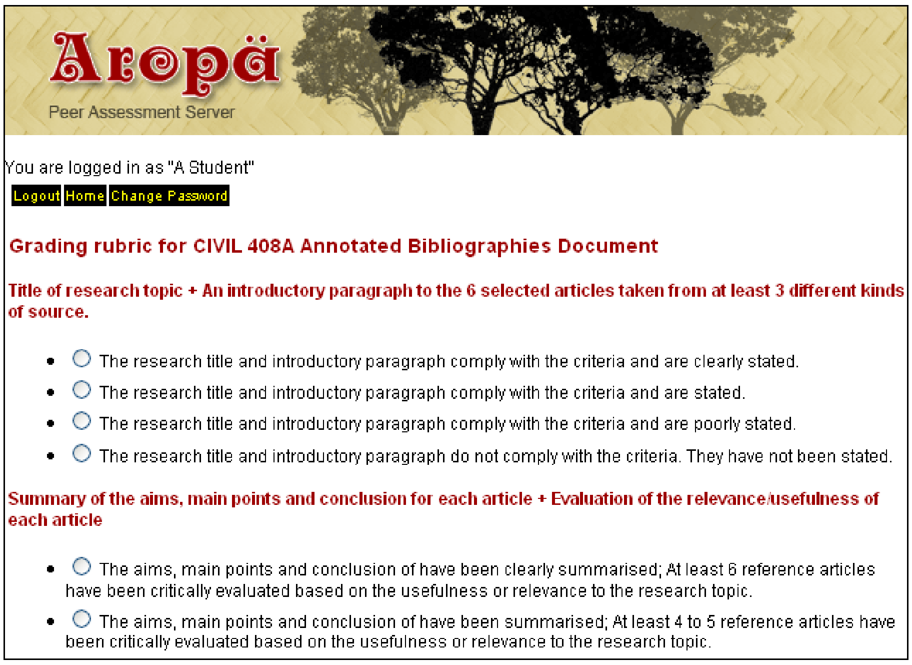
Reviewer feedback

- Feedback for "Batou Ltd v. Motoko Ltd"
- Feedback for "Batou Ltd v Gundam Corporation"
- Feedback for "Barney's Bikes Ltd."

Figure 1: The main page for a student using Aropä. All three student activities (uploading, as author; reviewing; and receiving feedback) are available from this page (uploading is not shown).

For an instructor, using Aropä involves: deciding on the date submissions are due, the length of the review period, the number of reviews to be allocated to each student, and designing the grading rubric. Allocating three to five reviews to each student seems to give a good balance between the need to provide a variety of feedback and the time required to write the reviews. Long review periods have largely proven unsuccessful. One of the strengths of peer assessment is in providing rapid feedback, and we have found reviewer participation rates fall away if the reviewing does not start immediately after the submissions are complete. Allowing two days to at most a week for reviewing has generally worked well.

We have found that Aropä tends to be used in three distinctive ways: (a) formative feedback on a draft; (b) critical reflection after an assignment; and (c) for summative assessment. These differ in the timing of the activity, the style of the grading rubric, and the level of compulsion and associated marks. The rubric partially shown in Figure 2 is a typical example of a summative assessment activity. The criteria are quite precise and detailed. An instructor using this style of rubric is primarily interested in giving feedback to the author, rather than having the reviewer reflect closely on the marking process, which is tightly controlled. Consistency of marking is a major consideration when the assessment carries significant weight toward the student's final grade, and Aropä provides facilities to identify wayward reviewers and to give greater weight to the marks from reviewers who have done a good job. Marks are often awarded for both the authoring and the reviewing, using a "review of the reviews" for the latter. This provides a strong motivation for students to take the reviewing seriously (although there have been remarkably few incidents of problems in this regard).



You are logged in as "A Student"

Grading rubric for CIVIL 408A Annotated Bibliographies Document

Title of research topic + An introductory paragraph to the 6 selected articles taken from at least 3 different kinds of source.

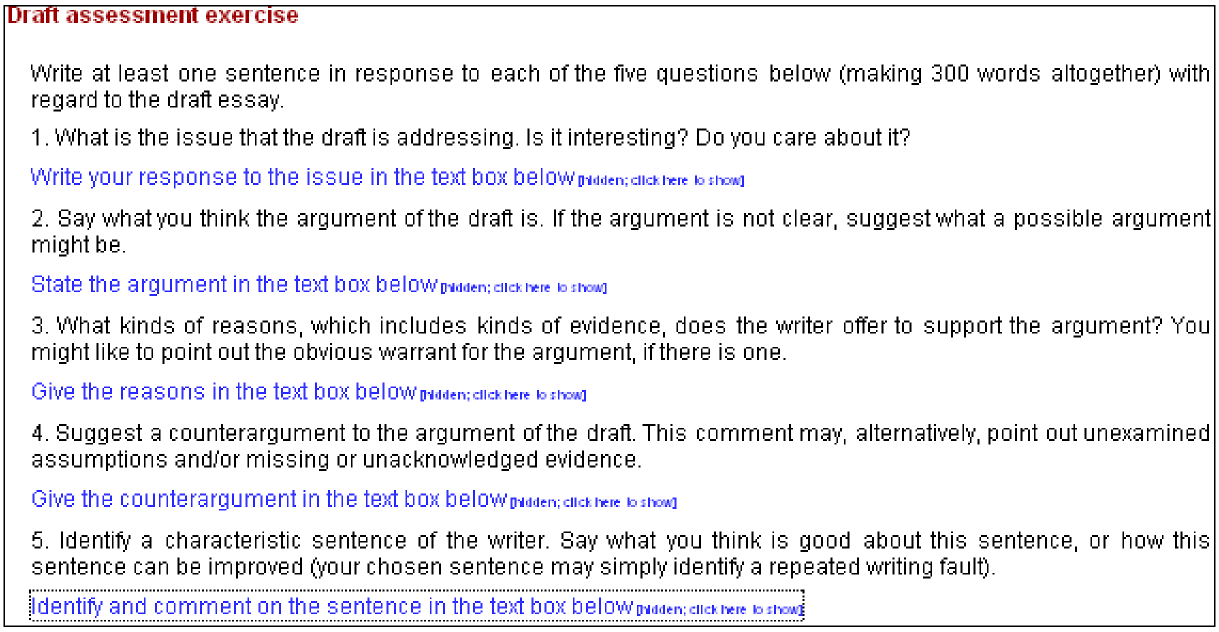
- ○ The research title and introductory paragraph comply with the criteria and are clearly stated.
- ○ The research title and introductory paragraph comply with the criteria and are stated.
- ○ The research title and introductory paragraph comply with the criteria and are poorly stated.
- ○ The research title and introductory paragraph do not comply with the criteria. They have not been stated.

Summary of the aims, main points and conclusion for each article + Evaluation of the relevance/usefulness of each article

- ○ The aims, main points and conclusion of have been clearly summarised; At least 6 reference articles have been critically evaluated based on the usefulness or relevance to the research topic.
- ○ The aims, main points and conclusion of have been summarised; At least 4 to 5 reference articles have been critically evaluated based on the usefulness or relevance to the research topic.

Figure 2: A grading rubric, showing Likert-style selections with detailed commentary provided by the instructor. Rubrics can also include open-ended comment fields and yes/no check boxes, as well as images, tables, multiple levels of headings, and character and paragraph styles.

An example of a grading rubric for formative feedback is shown in Figure 3. This rubric guides the reviewer in commenting reflectively on a draft essay. There are no quantitative elements, just a series of open-ended comment boxes. Since the essay is just a draft, the author has an opportunity to incorporate any feedback into their final version. Significantly, students have reported that reviewing other essays helped them identify faults in their own writing. The feedback process is two-pronged.


Draft assessment exercise

Write at least one sentence in response to each of the five questions below (making 300 words altogether) with regard to the draft essay.

1. What is the issue that the draft is addressing. Is it interesting? Do you care about it?

Write your response to the issue in the text box below [hidden; click here to show]

2. Say what you think the argument of the draft is. If the argument is not clear, suggest what a possible argument might be.

State the argument in the text box below [hidden; click here to show]

3. What kinds of reasons, which includes kinds of evidence, does the writer offer to support the argument? You might like to point out the obvious warrant for the argument, if there is one.

Give the reasons in the text box below [hidden; click here to show]

4. Suggest a counterargument to the argument of the draft. This comment may, alternatively, point out unexamined assumptions and/or missing or unacknowledged evidence.

Give the counterargument in the text box below [hidden; click here to show]

5. Identify a characteristic sentence of the writer. Say what you think is good about this sentence, or how this sentence can be improved (your chosen sentence may simply identify a repeated writing fault).

Identify and comment on the sentence in the text box below [hidden; click here to show]

Figure 3: A formative grading rubric. The rubric is comprised of open-ended responses only, and explicitly requests the reviewer state their opinion, with no expectation of there being a single "correct answer"

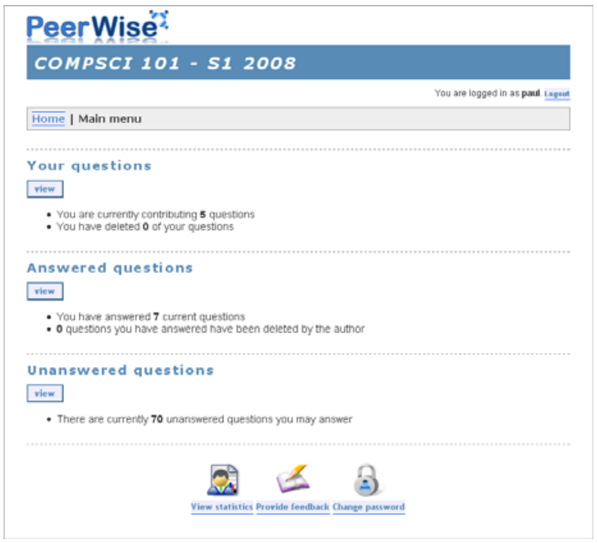
The third type of use is a combination rubric that includes both formative and summative elements. These are often used in courses with a series of short weekly or fortnightly assignments, to add a brief peer assessment step onto the end of each. Students in these courses often report finding value in simply reading work from their peers. This allows them to judge their own performance relative to the class. Over-confident students are given a reality check, and students suffering from a lack of confidence (in computing, this frequently means women) are reassured that they are not doing so badly after all.

3. PeerWise

PeerWise was developed to enable students to create, share and evaluate multiple-choice questions with accompanying answers and explanations. All of the content on PeerWise is developed by students as a course progresses, and remains available for revision purposes prior to final examinations. All activity on PeerWise, such as developing new questions, answering existing questions, and rating and providing feedback on questions is confidential, which is designed to encourage participation from students who may otherwise find it intimidating to contribute to a public resource.

3.1 Design Overview

For a particular student, the questions in PeerWise can be classified into three groups: those that have been created by the student, those that have been created by others and which the student has answered, and those that have been created by others but which the student has not yet answered. The main menu (Fig. 4), displayed once a student logs in, is divided into these three sections: "Your questions", "Answered Questions" and "Unanswered questions", which are described in detail next.



You are logged in as **paul**.

Your questions

- You are currently contributing **5** questions
- You have deleted **0** of your questions

Answered questions

- You have answered **7** current questions
- **0** questions you have answered have been deleted by the author

Unanswered questions

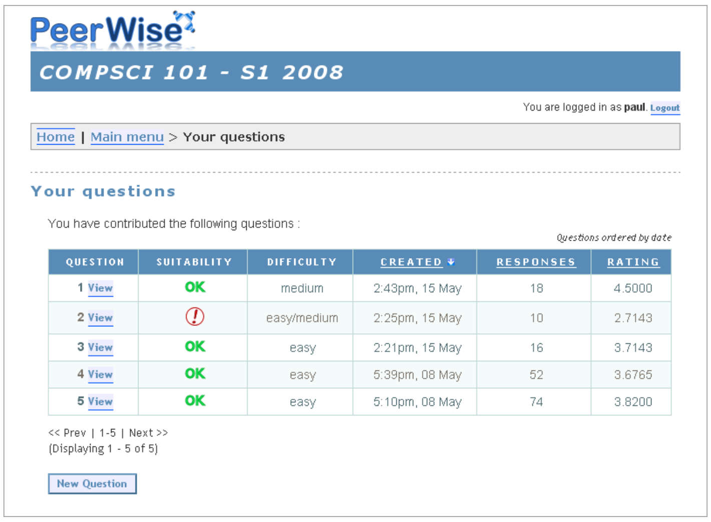
- There are currently **70** unanswered questions you may answer

Figure 4: The main menu for the PeerWise system

3.1.1 Your questions

This section (Fig. 5) displays all of the questions that a student has contributed. The items are displayed in a table with columns listing the date the question was created, how many times the question has been answered and the current rating of the question. The table can be sorted with respect to any of these columns. There is also a column that displays the difficulty of the question, as perceived by students who have answered it, and a column that indicates whether or not the question is "suitable". A simple metric is used to determine the suitability of a question, which requires that the rating of the question is greater than 2 and that the most popularly selected answer matches the answer suggested by the author of the question. If either of these conditions is not met, it may indicate the question is tricky or contains errors.

The details of a question are displayed when it is selected from the table. These details include the question text, as well as a histogram showing how often each alternative has been selected, and any feedback that has been provided by students who have answered it. While viewing their contributed questions, the author is able to provide a written response to any feedback about their question.


PeerWise
COMPSCI 101 - S1 2008
You are logged in as paul. Logout
Home | Main menu > Your questions

Your questions

You have contributed the following questions :

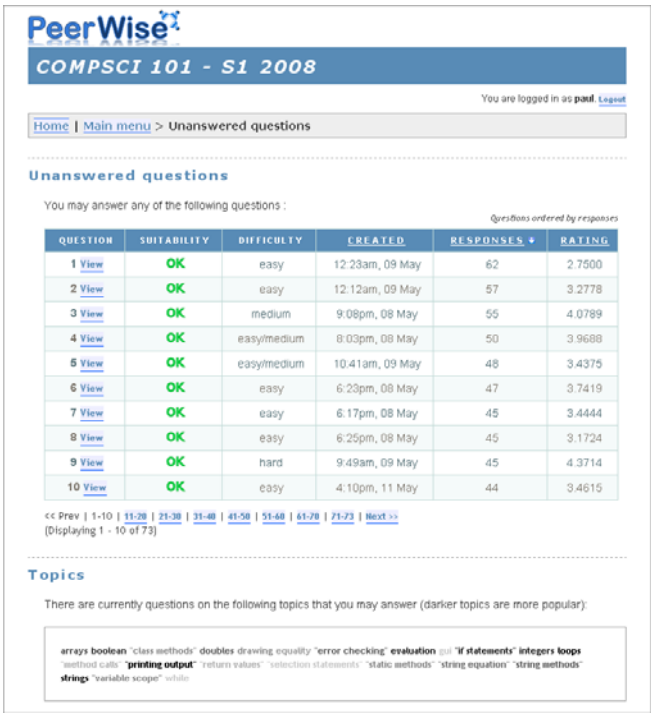
Questions ordered by date

QUESTION	SUITABILITY	DIFFICULTY	CREATED	RESPONSES	RATING
1 View	OK	medium	2:43pm, 15 May	18	4.5000
2 View	❗	easy/medium	2:25pm, 15 May	10	2.7143
3 View	OK	easy	2:21pm, 15 May	16	3.7143
4 View	OK	easy	5:39pm, 08 May	52	3.6765
5 View	OK	easy	5:10pm, 08 May	74	3.8200

<< Prev | 1-5 | Next >>
(Displaying 1 - 5 of 5)

Figure 5: Page showing the questions written by the student

To create a new question, the student provides a question stem and between two and five alternatives. The student indicates which of the alternatives is correct, and provides an explanation for the answer. The explanation is displayed to all students upon answering the question, and is designed to assist students who have answered the question incorrectly to understand their error. A simple tagging system allows question contributors to indicate the relevant topics that their questions assess. The tags are presented in a cloud that enables students to quickly locate questions on topics of interest. As soon as a new question is contributed, it immediately becomes available in the "Unanswered questions" section (Fig. 6) for all other students in the course.


PeerWise

COMPSCI 101 - S1 2008

You are logged in as **paul**. Logout

Home | Main menu > Unanswered questions

Unanswered questions

You may answer any of the following questions :

Questions ordered by responses

QUESTION	SUITABILITY	DIFFICULTY	CREATED	RESPONSES	RATING
1 View	OK	easy	12:23am, 09 May	62	2.7500
2 View	OK	easy	12:12am, 09 May	57	3.2778
3 View	OK	medium	9:08pm, 08 May	55	4.0789
4 View	OK	easy/medium	8:03pm, 08 May	50	3.9688
5 View	OK	easy/medium	10:41am, 09 May	48	3.4375
6 View	OK	easy	6:23pm, 08 May	47	3.7419
7 View	OK	easy	6:17pm, 08 May	45	3.4444
8 View	OK	easy	6:25pm, 08 May	45	3.1724
9 View	OK	hard	9:49am, 09 May	45	4.3714
10 View	OK	easy	4:10pm, 11 May	44	3.4615

<< Prev | 1-10 | 11-20 | 21-30 | 31-40 | 41-50 | 51-60 | 61-70 | 71-73 | Next >>
(Displaying 1 - 10 of 73)

Topics

There are currently questions on the following topics that you may answer (darker topics are more popular):

arrays boolean "class methods" doubles drawing equality "error checking" evaluation gui "if statements" integers loops "method calls" "printing output" "return values" "selection statements" "static methods" "string equation" "string methods" strings "variable scope" while

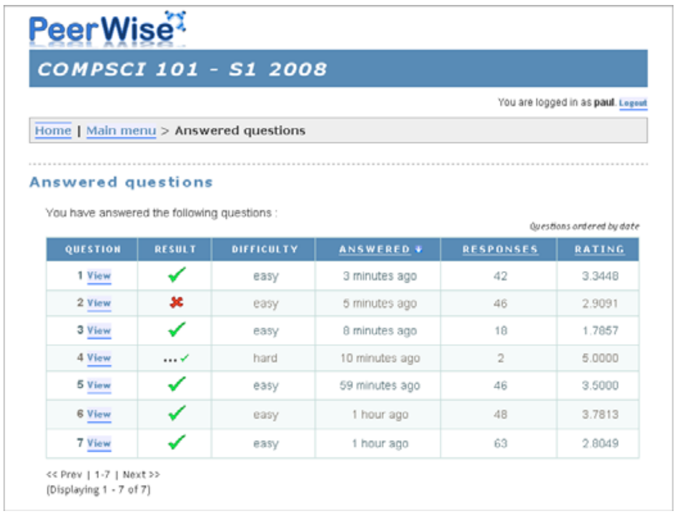
Figure 6: Page showing unanswered questions and topic cloud

3.1.2 Unanswered questions

Every student has access to all of the questions in the system. The unanswered questions are presented in a table from which the student can select individual questions to answer. The columns of this table include the perceived difficulty, number of responses, and current rating of each question. As the questions are also tagged by topic, students using PeerWise for drill-and-practice revision can spend their time answering highly rated questions on topics of interest to them, at a difficulty level they feel comfortable with.

Once a student selects an answer to a question, they are immediately shown the correct answer suggested by the author of the question, and the number of times each alternative was selected by other students in the course. The explanation provided by the question author is also displayed, as are all student comments written about the question. A simple metric is used to assess whether the selected answer is correct. If the answer selected by the student matches the answer suggested by the question author, and in turn this matches the most popular answer selected by other students, then the answer is deemed to be correct. In other scenarios, different icons are displayed depending on whether the student agreed with the author, or with the most popular answer selected by other students.

At this point, a rating form is displayed, which allows the student to rate the quality and difficulty of the question, and provide their own feedback. The quality rating is on a scale of 0 to 5, and the difficulty can be specified as either "easy", "medium" or "hard". As questions are answered, they are moved from the “Unanswered questions” section to the "Answered questions" section (Fig. 7) where they remain available for review at any time.



PeerWise

COMPSCI 101 - S1 2008

You are logged in as **paul**. Logout

Home | Main menu > Answered questions

Answered questions

You have answered the following questions :

Questions ordered by date

QUESTION	RESULT	DIFFICULTY	ANSWERED	RESPONSES	RATING
1 View	✓	easy	3 minutes ago	42	3.3448
2 View	✗	easy	5 minutes ago	46	2.9091
3 View	✓	easy	8 minutes ago	18	1.7857
4 View	...✓	hard	10 minutes ago	2	5.0000
5 View	✓	easy	59 minutes ago	46	3.5000
6 View	✓	easy	1 hour ago	48	3.7813
7 View	✓	easy	1 hour ago	63	2.8049

<< Prev | 1-7 | Next >>
(Displaying 1 - 7 of 7)

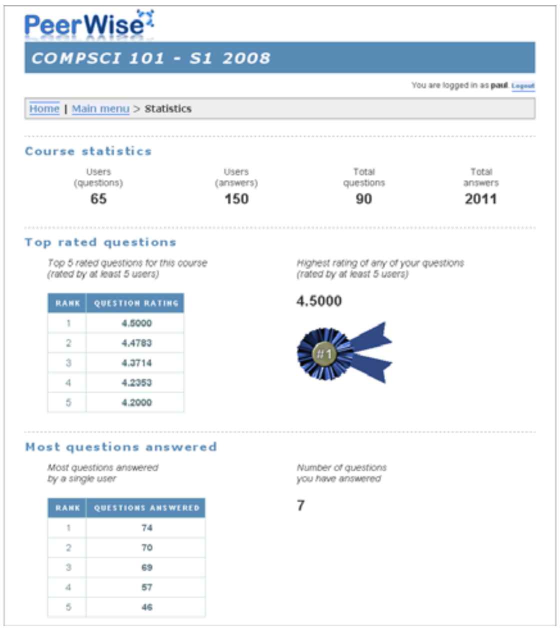
Figure 7: Page showing answered questions

3.1.3 Answered questions

The questions that a student has currently answered are always available for review, and are displayed in a table in the Answered questions section. If a student has provided feedback on a question, they can check to see if the author has responded to the feedback in this section. In addition, the accuracy of the ratings and correctness metric for these questions improves as more students respond.

3.1.4 Leaderboards

Students' contributions are anonymously ranked on a basic leaderboard (Fig. 8). From logs, we see that a number of students regularly check their status in these leaderboards which we believe provides motivation for contributing beyond the minimum requirements for assessment. Specific tables display the top rated questions, and rank students on the number of questions that have been answered, on the popularity of question authors and on the popularity of students who have written feedback on questions.



PeerWise

COMPSCI 101 - S1 2008

You are logged in as **paul**. Logout

Home | Main menu > Statistics

Course statistics

Users (questions)	Users (answers)	Total questions	Total answers
65	150	90	2011

Top rated questions

Top 5 rated questions for this course (rated by at least 5 users)

RANK	QUESTION RATING
1	4.5000
2	4.4783
3	4.3714
4	4.2353
5	4.2000

Highest rating of any of your questions (rated by at least 5 users)

4.5000

Most questions answered

Most questions answered by a single user

RANK	QUESTIONS ANSWERED
1	74
2	70
3	69
4	57
5	46

Number of questions you have answered

7

Figure 8: Page showing leaderboard

3.2 Current usage

PeerWise was first introduced at the start of 2007, and has since been used in 12 courses at the University of Auckland and 4 courses at the University of British Columbia. While still in use at the time of writing, there have currently been over 7000 questions and 170000 answers contributed by 3000 students. While the teaching staff, style of course examinations and requirements for assessment have varied greatly amongst these courses, we can report on several patterns of use that appear to be common to all courses.

3.2.1 Contributing questions

Firstly, most students contribute only the minimum number of questions that are required for assessment. Developing questions can be very time consuming and students may feel they get more benefit from answering many questions in the same time that they could contribute just one, which is supported by the following qualitative data:

> *"I answered lot of questions but only developed 2 questions because I don't enjoy making questions"*

> *"I think I gained more from answering than submitting."*

However, we do see evidence in the qualitative data that at least some students feel they learn more from taking the time to develop new questions:

> *"Thinking about actually forming a question helps more than simply answering them."*

> *"Developing questions makes me read the book and actually thinking deeply about the subject."*

> *"I had to think of the possible wrong solutions students would fall for and required alot of thinking from me, which in the end was a lot of help because i was just about able to answer any question that was on the same topic as my question. That was the biggest learning experience for me!"*

Despite the fact that the majority of students are reluctant to contribute more questions than are required, PeerWise has been used in courses where contributing questions is voluntary. This has proven unsuccessful in small courses (less than 100 students), but remarkably successful in large courses (at least 500 students). The success may be attributed to the fact that the students who are willing to contribute questions voluntarily are motivated and keen, and tend to produce good quality questions. Once there are a certain number of questions in the system, which tends to occur in large classes, it becomes useful as a revision resource for other students.

We have found PeerWise to be most positively received in classes with a conservative requirement on the number of questions that must be contributed per student. Having a high contribution requirement places significant load on the students, and can lead to a reduction in the quality of the questions available. In one 12 week course, students were required to submit two original questions every week, and this appears to have been too heavy a load for most students. Qualitative feedback supports this:

"How about 2 question for every two weeks? Seriously need more time to actually study..."

"1/week instead. Too many questions!"

"Having to contribute 2 unique questions / week is discouraging - I'm only willing to put so much time towards PeerWise, and so far all of that time has been consumed coming up w/questions, leaving no time to answer any."

"Reducing the questions to once a week may help reduce the frequency of redundant questions."

3.2.2 Answering questions

Students answer many more questions than they are required to. In some courses, the number of answers submitted by students was more than 20 times the minimum requirement [19]. It is particularly interesting to note that in almost all courses, there is a significant increase in the number of answers submitted to PeerWise immediately leading up to course tests and exams. Students seem very positive about using the bank of questions developed by their peers for revision purposes. In one large course, 600 students submitted in excess of 40,000 answers in the five days leading up to the course test [Fig. 9].

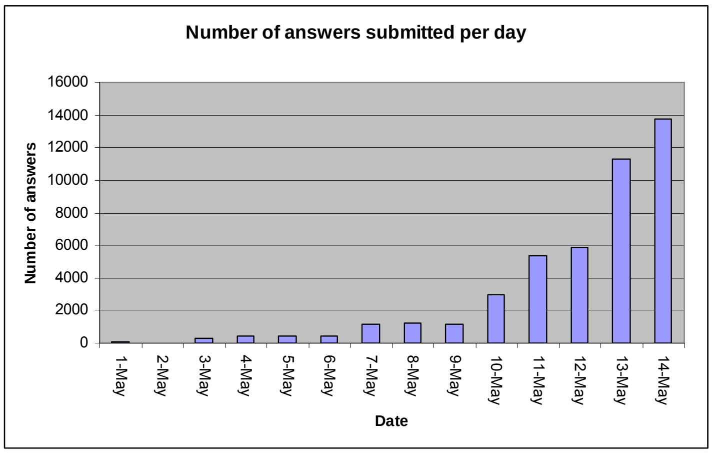


Figure 9: Number of answers submitted per day in the two weeks leading up to a course test, held on the evening of 14th May. There were 600 students answering questions during this period.

3.2.3 Providing feedback

Students in all courses made use of the feedback facilities to provide comments about questions in the system. Although there are typically no marks allocated to this process, in all courses students made a substantial number of comments.

3.3 Efficacy

To investigate the efficacy of the PeerWise tool with regards to student learning, we analysed quantitative data obtained from a large first-year programming course. This course had 460 students whom collectively contributed 1238 questions and 16247 answers to PeerWise.

The top students in the class exhibit different characteristics to the weakest students [19]. In order to understand how the use of PeerWise affected different students, we divided the class into quartiles and for each quartile, asked whether the use of PeerWise provided any measurable benefit. The quartiles, each consisting of 115 students, were formed using the mark obtained in the mid-semester test which was conducted before PeerWise was introduced. Within each quartile, students were ranked on their level of activity with PeerWise, and we compared the most active half with the half that was least active, using several different measures of activity.

Active use of PeerWise improved students' grades in both the multi-choice and written sections of the final examination. The improvements in the written sections seem to imply that the use of PeerWise resulted in deep learning, rather than simply coaching students into better MCQ technique. Our analysis of different measures of PeerWise activity suggests that, in addition to time-on-task, voluntary engagement through the question discussion forum is a strong contributor to this improvement.

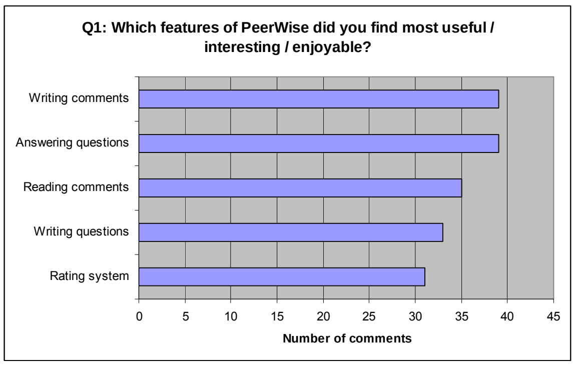
The benefits of PeerWise are not confined to students of just high or just low ability. We see improvements across all performance quartiles, and most consistently in the top and bottom groups. There is some evidence to suggest the benefit to mid-ability students is less significant, which raises the challenge of developing variations to PeerWise which may better engage learners at all ability levels.

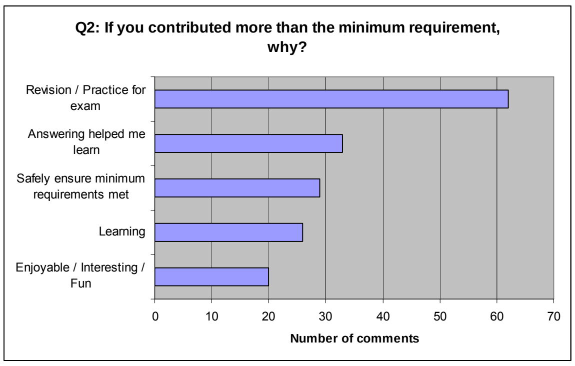
3.4 Qualitative feedback

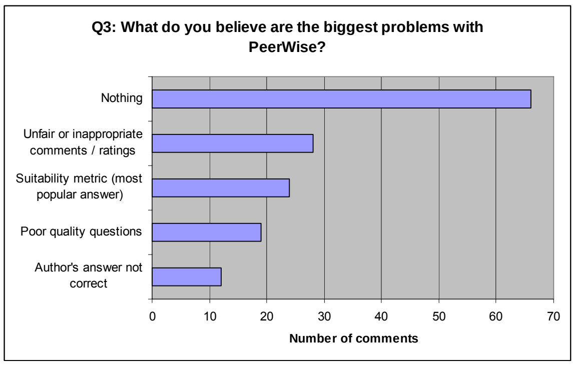
We conducted a survey of students in a first year programming course who had used PeerWise throughout the second half of the semester. There were 439 respondents, and each was asked to provide their thoughts on the following open-ended questions:

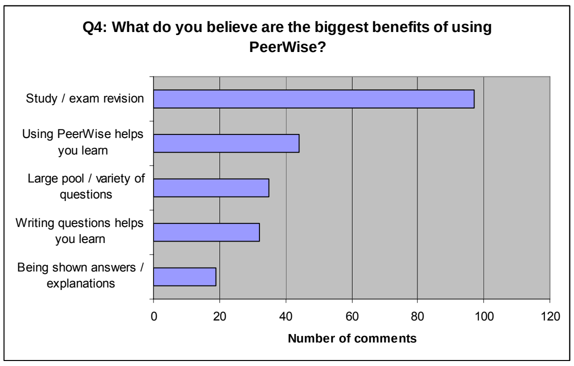
- Q1: Which features of PeerWise did you find most useful/interesting/enjoyable?
- Q2: If you contributed more than the minimum requirement, why?
- Q3: What do you believe are the biggest problems with PeerWise?
- Q4: What do you believe are the biggest benefits of using PeerWise?

We categorised the responses to these questions, and the 5 most common categories of responses are summarised below:









4. Discussion

Students using Aropä are focused on the formal peer review process. They are required to formally assess a number of pieces of work according to a specified marking rubric. The rubric provides guidance for the evaluation activities. Students must interpret and apply the specified criteria, making critical judgements about the quality of work they are reviewing.

PeerWise focuses student attention on assessment and how it connects with the learning outcomes of a course. Students are expected to create questions that probe understanding, requiring the application of higher-order cognitive skills.

Both systems are used to develop appropriate self assessment skills. Using Aropä, students are expected to apply the marking rubric to their own work as well as that of their peers. Students using PeerWise use the question bank to assess their own understanding.

Feedback is a critical component of effective education practice, and one that is difficult to deliver in large classes which often rely on the ability of casual tutors to provide the feedback. PeerWise and Aropä both use the student cohort to provide this feedback. As the feedback is not from an authoritative source, the act of receiving feedback requires further evaluation and judgement to determine its accuracy and what benefits can be obtained from it. These systems encourage the application of knowledge, enquiry and critical thinking skills.

5. Related Work

The design of the PeerWise tool has previously been reported in [18]. An analysis of the common usage patterns over a range of courses has been performed [19], and the efficacy of the tool has been evaluated by considering whether increased activity leads to improved student exam performance [20]. Students of all ability levels appear willing to answer many more questions than they are required, with increased usage occurring during revision periods prior to exams. The most actively engaged students perform significantly better in written exams than students of equivalent ability who are less active on PeerWise.

The Aropä tool has been described previously in [17]. Qualitative data collected from two large courses was analysed to evaluate the effectiveness of the tool. The evidence suggests that Aropä successfully supported peer review activities in large classes and contributed to student learning on many different levels.

Although there are previous reports of systems that support student-developed MCQs [21] [22] [23] [24], and peer review in large classes [25] [26] there are no reports of such systems being implemented so extensively in such a large number of courses. Kern et al. [27] recently noted that while the use of peer review in the classroom has numerous benefits, there has been only one report of large scale education application of peer review, conducted in 2001, and involving 411 students. They report that peer review *should* be used in higher education and *should* become regular educational practice.

6. Conclusions and future work

Web technologies are able to provide a great variety of environments that engage students in different forms of collaborative learning. The two systems described in this paper are excellent examples of this. They are both practical. The systems have been designed to scale to accommodate arbitrarily large class sizes, and they do this with minimal additional load on instructors. The systems also contribute to the dual need for students to not only learn specific course material, but also to develop into self-reliant learners able to collaborate with others. They do this by having students take on multiple roles, as receivers and generators of knowledge, within a learning community.

We now have sufficient experience in using these tools to be able to make recommendations on how they can best be adopted in a range of classes and subject areas. The data we are collecting on student performance and attitudinal change is providing evidence that the contribution-based approach is indeed effective in the ways we have hoped. We are also looking at identifying subgroups of students who do not engage with the activities or show performance gains, in the hope that better understanding these groups will lead to ideas for new tools and refinements for our existing ones.

References

1. Collis B. The contributing student: A blend of pedagogy and technology. In: EDUCAUSE Australasia, Auckland, New Zealand; 2005.

2. Hamer J. Some experiences with the “contributing student approach”. ACM SIGCSE Bulletin 2006;38(3):68-72.

3. Wenger E, McDermott R, Snyder WM. W. M. Cultivating communities of practice: a guide to managing knowledge, Harvard Business School Press; 2002.

4. Brookfield, S. (1985): Self-directed learning: a critical review of research. New Directions for Adult and Continuing Education 1985;25:5-16.

5. Barker LJ, Garvin-Doxas K, Roberts E. What can computer science learn from a fine arts approach to teaching? In: SIGCSE ’05: Proceedings of the 36th SIGCSE technical symposium on Computer science; 2005. p. 421–5.

6. Cohoon JM. Recruiting and retaining women in undergraduate computing majors. ACM SIGCSE Bulletin 2002;34(2): 48–52.

7. Mayes T, Fowler C. Learners, learning, literacy and the pedagogy of e-learning. In: Martin A, Madigan D, editors. Digital Literacies for Learning. Facet Publishing; 2006.

8. Birenbaum M. Assessment 2000: Towards a pluralistic approach to assessment. In: Birenbaum M, Dochy FJRC, editors. Alternatives in assessment of achievements, learning processes and prior knowledge. Boston: Kluwer Academic; 1996. p. 3-30.

9. Biggs J. Teaching for quality learning at university. 2nd ed. Buckingham: The Society for Research into Higher Education and Open University Press; 2003.

10. Boud D, Cohen R, Sampson J. Peer learning and assessment. In: Boud D, Cohen R, Sampson J, editors. Peer learning in higher education. Kogan Page; 2001. p. 67–81.

11. Stefani LAJ. Assessment in partnership with learners. Assessment and Evaluation in Higher Education 1998;23:339-50.
12. Boud D. Enhancing learning through self assessment. Kogan Page; 1995.
13. Boud D. Assessment and learning: contradictory or complementary? In: Knight P, editor. Assessment for Learning in Higher Education. London: Kogan Page; 1995. p. 35-48.
14. Boud D, Falchikov N. Quantitative studies of student self-assessment in higher education: a critical analysis of findings. Higher Education 1989;18(5):529–49.
15. Falchikov N, Boud D. Student self-assessment in higher education: a meta-analysis. Review of Educational Research 1989; 59(4):395–430.
16. Topping K. Peer assessment between students in colleges and universities. Review of Educational Research 1998;68(3):249–76.
17. Hamer J, Kell C, Spence F. Peer assessment using Aropa. In: ACE '07: Proceedings of the ninth Australasian conference on computing education; Darlinghurst, Australia; 2007. p. 43–54.
18. Denny P, Luxton-Reilly A, Hamer J. The PeerWise system of student contributed assessment questions. In Simon, Hamilton M, editors, ACE '08: Proceedings of the tenth Australasian conference on computing education; Wollongong, NSW, Australia; 2008. p. 69–74. (CRPIT; vol 78).
19. Denny P, Luxton-Reilly A, Hamer J. Student use of the PeerWise system. ITiCSE'08, June 30–July 2, 2008, Madrid, Spain; 2008. p. 73-77.
20. Denny P, Hamer J, Luxton-Reilly A, Purchase H. PeerWise: Students Sharing their Multiple Choice Questions. To appear in: ICER'08, 6–7 September, 2008, Sydney, Australia; 2008.
21. Arthur N. Using student-generated assessment items to enhance teamwork, feedback and the learning process. Synergy 2006; 24:21–3.
22. Barak M, Rafaeli S. On-line question-posing and peer-assessment as means for web-based knowledge sharing in learning. International Journal of Human-Computer Studies 2004;61:84–103.
23. Horgen SA. Pedagogical use of multiple choice tests - students create their own tests. In: Kefalas P, Sotiriadou A, Davies G, McGettrick A, editors. Proceedings of the Informatics Education Europe II Conference. SEERC; 2007.
24. Yu F, Liu Y, Chan T. A web-based learning system for question posing and peer assessment. Innovations in Education and Teaching International 2005;42(4):337–48.
25. Lewis S, Davies P. Automated peer-assisted assessment of programming skills. In: Information Technology: Research and Education; 2004. p. 84–6.
26. Sitthiworachart J, Joy M. Effective peer assessment for learning computer programming. In ITiCSE '04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education. New York, NY, USA; 2004. p. 122–6.
27. Kern VM, dos Santos Pacheco RC, Saraiva LM, Pernigotti JM. Peer review in computer sciences: Toward a regular, large scale educational approach. In: Neto FMM, Brasileiro FV, editors. Advances in computer supported learning. Information Science Publishing; 2007. p. 45–65.

Software Development as the Core of Informatics

Tony Cowling

Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello Street, Sheffield, S1 4DP, England, A.Cowling@dcs.shef.ac.uk

This paper argues that the traditional view of programming as the core activity within the study of informatics at undergraduate level needs to be enlarged, to include the whole series of activities that comprise the software development lifecycle. This series of activities is defined precisely, along with the restrictions that should be applied to them to balance the need to keep this core of the curriculum small against the need to cover adequately the whole of the lifecycle. To explain why this core of the curriculum needs to be extended the paper describes how the understanding of software development activities has evolved, both in model curricula and in practice, showing how a gap has emerged between curricula and practice that needs to be filled. Then the material that needs to be brought into the core of informatics to fill this gap is discussed, considering both curriculum content and course structures, and then discussing why this additional material is important. Finally the paper describes how the various disciplines within informatics build on this enlarged core to reflect their distinctive features.

Keywords

Model curricula, programming, progressive development of skills, software engineering.

1. Introduction

If one were to ask a typical educator in informatics – and particularly a typical educator in Computer Science (CS from now on) – what they considered to be the core of the discipline of informatics, it is likely that key words in their answer would be "programming" or "software". Further, they might then describe the relationships between these two essentially in terms of programming being the activity of creating software, as implied by the definition given in Computing Curricula 91 [1] of programming as "*the entire collection of activities that surround the description, development, and effective implementation of algorithmic solutions to well-specified problems*". There are thus good historical reasons for putting this emphasis on programming and also for understanding the relationship between programming and software in this way, and these will be reviewed later in this paper. The main purpose of the paper is, however, to argue that this relationship between programming and software has changed in recent years to such an extent that describing it in this way is now far too simplistic, so that this historical emphasis on programming must therefore be reconsidered.

Specifically, in the few years immediately leading up to the Computing Curricula 2001 project (CC2001 from now on) and the creation of its CS volume [2] (CS2001 from now on), huge changes were taking place in the industrial and commercial practices for creating software, but these were at best only partially reflected in CS2001 itself. The overall effect of these changes has been to emphasise that the process of creating software consists not just of a single activity, namely programming, but of a number of activities, of which programming is just one. An illustration of this is that it is commonplace to refer to the software development lifecycle, where the most important feature that is common to all the different descriptions of this lifecycle is that it covers a wide range of activities. Some of these, such as software design or software testing, do cover various activities that could be described as aspects of programming, but others of them – particularly those that come early in the lifecycle – certainly are not programming in any shape or form, although (as will be discussed later) they are at least as important to the success of the whole lifecycle as the programming-related activities, if not more so. Hence, the basic argument of this paper is that, since the creation of software has now grown as an activity in this way, informatics curricula need to recognise this growth. Specifically, this means that these curricula need to expand their view of the core of the curriculum, so as to recognise that it should now be taken to be this wider activity of software development (SD from now on), rather than just the programming component of it that has traditionally had this role.

Given this aim, the structure of the rest of the paper is as follows. Firstly the scope of SD needs to be defined, which is done in the next section, while section 3 reviews the relevant history, both of the role of programming and the evolution of SD. Section 4 then presents the elements that need to be included in the curriculum if SD is to form its core, and section 5 justifies teaching these elements, while section 6 discusses each of the disciplines within informatics to show how SD forms the core of it. Finally, section 7 summarises the conclusions of the paper, and the further developments that are needed for its ideas.

2. Defining Software Development

The aspect of informatics that is being described here as SD was identified in the course of work [3] concerned with defining the stages through which students develop the practical skills needed to do software engineering (SE from now on). This work regarded the achievement of basic skills in programming as a foundational stage, and it then identified the next stage as achieving the ability to perform SD. This was intended to be the most minimal subset of SE that was meaningful, in the sense of covering in some form the whole of the SD lifecycle. As such it involves far more than just the recognition of the roles of processes and models within programming [4, 5], important though these are. Rather, SD progresses on from programming by dealing with systems instead of just programs, and so shifting focus from the technology of software to the kinds of purposes for which it is applied, where typically the problems are far less well specified than the earlier definition of programming applies. Thus, the skills to be developed in the SD stage were defined as a tightly restricted subset of the more general abilities that would be required for a software engineering graduate (as discussed in [6]).

2.1 Restrictions on Software Development

Specifically, the previous definition of SD as a subset of SE restricted it to the use of basic methods in order to produce software systems that provide feasible solutions to sets of functional requirements. Hence, SD involves some form of each of the main activities of the lifecycle, from requirements elicitation through to deployment. Then, the significance of the restrictions to basic methods, feasible solutions and functional requirements is that, unlike SE, SD is not intended to give any consideration to quality issues, whether of the product being created or the process by which it is created, with two exceptions. One of these is explicit, and is the basic need for the systems that are created to function correctly, meaning that when their functions are executed they should produce results that satisfy their specifications, since if they did not then those systems could not be said to be solutions to their functional requirements. The other exception is implicit, and is that the systems created must provide interfaces for their users that are sufficiently usable to allow a user to actually make the systems perform their required functions without too much difficulty, as otherwise those systems again could not be said to be solutions to their functional requirements. In practice, though, the use of standard techniques for the construction of menu-driven user interfaces, which is implied in the definition of SD, should normally be sufficient to achieve at least this basic level of usability, and this is why this exception is described as implicit.

More recently, work on the significance of application domains for informatics curricula [7] has indicated that another restriction now needs to be added to this previous definition of SD, so that for the purposes of this paper it is now being defined as the use of basic methods to produce software systems that provide feasible solutions to sets of functional requirements, which are typical of those arising in a simple and well-defined application domain. This additional restriction is intended to exclude application domains that require consideration of more than just functional requirements, by eliminating those domains that are characterised either by demands for systems to possess particular non-functional properties (eg safety, security, real-time performance, etc), or by demands for particular process methods to be used in constructing the systems (eg hazard analysis, modelling of performance or power consumption, etc) in order to achieve such properties.

A practical effect of this restriction of SD to simple and well-defined application domains is that it simplifies the processes that need to be used in most stages of the development lifecycle. For instance, typically any such domain will not require all the various kinds of analysis models to be constructed in order to represent the requirements for a system adequately, or to specify them precisely enough. Similarly, for each such domain there will usually be one style of architecture that is most commonly employed for systems, such as three-layer architectures for small business systems or transform centre ones for simple embedded systems. Hence, the adoption of such a style as the standard for its domain will reduce significantly the number of design choices to be made, and so simplify the design and construction activities for the systems.

2.2 The Content of Software Development

The combined effect of these restrictions is that, while there are obvious similarities between SD and SE, there are also significant differences between them. The similarities arise because both are concerned with the complete development lifecycle, from requirements elicitation and analysis through some kind of specification (not necessarily formal), system design, construction and testing to system deployment. Also, both focus on the need for systems that are developed to meet their requirements. The differences arise from the limitation of SD to only considering straightforward functional requirements, and hence only involving very basic development processes. This limitation makes SD a purely qualitative discipline, because the decisions to be made at each stage in the development lifecycle only have to consider a restricted range of options, and so can be framed in qualitative terms. By contrast, SE is in principle a quantitative discipline, even if in practice the amount of qualitative material to be covered within a curriculum to support the quantitative aspects (and particularly the variety of situations where these may be applied) means that undergraduate programmes in SE often have to compromise on the extent to which they can cover its quantitative aspects.

This is therefore why this previous work on the progressive development of students' skills argued that they needed to master SD, with its qualitative approach, before they could move on to studying the quantitative aspects that characterise SE. The core of the argument that this paper makes is then to extend this aspect of the progressive development of skills to the whole of informatics. This is not to suggest that students who have studied SD then need to progress on to studying SE, but rather that if they are going to study any discipline within informatics in depth they need to begin by studying the whole of SD, and not just the small part of it that is represented by programming.

3. A History of Software Development

The history of the ideas being discussed here goes right back to the very first proposals for a model curriculum for informatics, and the early evolution of software development can be traced effectively through the various curriculum models that were produced. Towards the end of the 1980s, however, the evolution of SD began to diverge from its treatment within curriculum models, and so these later developments need to be considered separately.

3.1 Curriculum 68

Curriculum 68 [8] was the first model curriculum for what we now call informatics, but which it then called CS, and it structured it into three divisions: *information structures and processes*, *information processing systems*, and *methodologies*. The first of these then consisted of three subject areas: *data structures*, *programming languages*, and *models of computation*; and so was concerned almost entirely with the theories of both software and programming, where the latter was seen just in terms of the representation of algorithms. The second division, information processing systems, consisted of four subject areas: *computer design and organisation*, *translators and interpreters*, *computer and operating systems*, and *special purpose systems*. Hence, it was concerned partly with software and partly with hardware. The third division, methodologies, was described as being "*derived from broad areas of applications of computing*", and covered ten subject areas, such as *numerical mathematics*, *data processing*, *process control*, etc, which would now usually be described instead as applications areas. Each of these areas then focused primarily on the software that was being applied within it, rather than on the applications themselves.

More importantly, for one of these application areas an advanced course was proposed, on *Large-Scale Information Processing Systems*, where the detailed description in the appendix included this explanation. "*The process of establishing such a large system involves a number of steps: (1) the determination of the processing requirements; (2) the statement of those requirements in a complete and unambiguous form suitable for the next steps; (3) the design of the system, i.e. the specification of computer programs, hardware devices, and procedures which together can "best" accomplish the required processing; (4) the construction of the programs and procedures, and the acquisition of the hardware devices; and (5) the testing and operation of the assembled components in an integrated system.*". Hence, even at this earliest stage it was recognised that there was actually far more to developing a software system than just programming. On the other hand, the content being proposed for this course only covered some of these steps, and the report [9] on the very first conference on software engineering, which took place some six months after publication of this model curriculum, strongly indicates that at that time very little systematic knowledge actually existed as to how they should be carried out.

3.2 Curriculum 78

Ten years later, Curriculum 78 [10] adopted a different fundamental structure, in which the primary feature was the distinction between core and elective courses, where the core courses were then described as covering four sets of topics, namely: *Programming Topics* (of which there were 5), *Software Organization* (8 topics), *Hardware Organization* (7 topics), and *Data Structures and File Processing* (7 topics). Even though these sets of topics were not necessarily intended to each occupy similar amounts of curriculum time, the fact that three out of the four had titles that implied that they were concerned with software or programming continued this emphasis on these two aspects, even though the topics in the software organization set were actually concerned mainly with the structures needed in assembly language programming, rather than with those used in what at that time were commonly called problem-oriented languages. Also, there was a much clearer view of programming as an activity, and hence as part of a process: for instance, the outline of the course *CS1 Computer Programming I* included the statement: "*The emphasis of the course is on the techniques of algorithm development and programming with style*".

Paradoxically, though, there appeared to be less recognition than previously that SD involved more than just programming, for even though there was an advanced course (CS14) on *Software Design and Development*, its content essentially just consisted of *Design Techniques* (defined very much in programming terms), *Organization and Management* (which we would now describe as basic project management methods), and a team project. Similarly, perhaps reflecting the limitations of knowledge suggested above, the advanced course (CS11) on *Database Management Systems Design*, which was the successor to the Curriculum 68 course discussed above, was mainly concerned with models and techniques, rather than with any process for the design or implementation of a database.

3.3 Curriculum 91

By contrast, Curriculum 91 was structured around ten compulsory subject areas: nine that had been defined in an earlier report [11], plus a tenth for social, ethical and professional issues. One of these ten subject areas was *Software Methodology and Engineering*, which mainly covered material from SD, and was allocated a similar number of core hours to each of the other four large areas, so that it had about a sixth of the total time. Of the other nine subject areas, two of the large ones (*Algorithms and Data Structures* and *Programming Languages*) were concerned explicitly with programming, but also an eleventh subject area was suggested, called *Introduction to a Programming Language*. This was intended to be optional, but in practice the material in it was sufficiently fundamental that any real programme would have to include it. On the face of it, therefore, programming only appeared to be allocated just over twice as much time as SD.

Part of the significance of this time allocation is that there had obviously been a major debate about the role of programming between the publication of the earlier report and of this model. The view taken in the earlier report can be summarised by its statement that "*Many activities in computing are not programming ... therefore the notion that "computer science equals programming" is misleading.*". By contrast, the curriculum model did emphasise the significance of programming, as "*Programming occurs in all nine subject areas in the discipline of computing ... programming is an extension of the basic communication skills that students and professionals normally use in day-to-day communication.*".

Reflecting this, the first two knowledge units of *Software Methodology and Engineering*, viz *Fundamental Problem-Solving Concepts* and *The Software Development Process*, which were allocated nearly half of the time for the subject, were actually oriented heavily towards programming, particularly for the suggested laboratory exercises. Hence, material relating to other activities within SD was essentially confined to the other three knowledge units in this area, viz *Software Requirements and Specifications*, *Software Design and Implementation* (which also covered a certain amount of programming material), and *Verification and Validation*. Thus, while the body of knowledge in this curriculum seemed to recognise the importance of SD as a whole, it actually treated programming as much more important, and in particular it defined far more material related to it.

The other key feature of Curriculum 91 is that much of its practical importance came from the complete programme structures (called implementations) that it proposed. In principle some of these were not too heavily oriented towards programming – for instance, there is one (implementation G) that is explicitly defined to have a software engineering emphasis, and which is strongly oriented towards SD – but in practice the only ones that were widely accepted were those that did have a strong emphasis on programming. In part this is probably a consequence of the status of the subject area *Introduction to a Programming*

Language, for in practice most educators would consider that the material in it could not be regarded as optional, and so those implementations which omitted it (the majority of them, including the one focusing on software engineering) were very unlikely to be utilised as models for real programmes.

Within the implementations that did include the subject area *Introduction to a Programming Language*, such as implementation A for Computer Engineering (CE from now on) and implementation D for CS, the main core courses (designated CS101 and CS102, and entitled *Introduction to Computing I* and *II*) were essentially very similar in content to each other, as well as being a natural updating of the courses CS1 and CS2 defined in Curriculum 78, which may explain why educators still commonly refer to these two introductory courses as CS1 and CS2. Hence, the content of these two courses was primarily oriented towards programming and related software concepts, as indicated by the following extract from their descriptions "*In the context of a modern programming language, such topics as problem solving strategies, basic data structures, data and procedural abstraction, and algorithm complexity are discussed.*". Given their role as the primary core courses, this emphasis therefore reinforced strongly the view that programming was synonymous with the creation of software, and was the core of informatics.

By contrast, in each of these two implementations the non-programming elements of the subject *Software Methodology and Engineering* were covered in a pair of courses designated *C302 Software Systems* and *C303 Software Engineering*, with most of the coverage of requirements analysis, specification, software design (particularly architectural design) and software validation coming in CS303, so that even CS302 had quite a strong programming flavour. Hence, these course structures very strongly suggested that these activities were definitely advanced ones, and only to be considered once the core activity of programming had been thoroughly mastered. Furthermore, although the implementations specified these courses as required, their allocation to the third level put them alongside most of the optional topics, which may have led many instructors to draw parallels with the division between core and elective topics in Curriculum 78, and so actually treat these courses as optional too.

3.4 Computing Curricula 2001

With the decision during the CC2001 project to split informatics into its separate disciplines, and create separate model curricula for each, it was perhaps inevitable that the various models should focus as much on the differences between the disciplines as on the elements that were common to them. There were some common elements, and in particular the volume for SE [12] (SE2004 from now on) imported several of the knowledge areas directly from CS2001, although it changed their structure by treating some of these complete areas as forming single knowledge units. CS2001 also appeared to try to integrate its SE knowledge area more with the rest of its core than Curriculum 91 had done, even though material on basic problem-solving was moved from this area to *Programming Fundamentals*, and largely replaced by a new knowledge unit on *Using APIs* (which was similarly oriented towards programming aspects of SD), as well as the addition of other units covering non-programming aspects such as software project management, formal methods, component-based development, etc.

Similarly, in terms of the course structures, while CS2001 defined the same three levels of courses as Curriculum 91, its main successor to the latter's courses CS302 and CS303 was designated as *CS290 Software Development*. Hence, the classification of this as an intermediate course suggests that its content was intended to be closer to the core than had been the case for the old CS302 and CS303, although like the old CS302 this content was actually oriented primarily towards the programming aspects of SD, as indicated by the main topics covered. Five of these are almost completely focused on programming: *event-driven programming*, *using APIs*, *computer graphics*, *introduction to HCI*, and *GUIs*; hence only the other three take a wider view of SD, viz: *human-centred software evaluation*, *human-centred software development*, and *software development techniques*. Thus, while in principle this course may have been intended to bring SD closer to the core of CS, in practice it did not succeed in doing this.

3.5 Modern Software Development

Part of the reason for this is that, in defining its knowledge area and courses relating to SD in this way, CS2001 had actually failed to track major developments in the practice of SD. Whereas it was true for both Curriculum 68 and Curriculum 78 that there was then little in the way of established practice for SD that could sensibly be taught in a curriculum, by the time of Curriculum 91 this had begun to change. The processes of requirements analysis and modelling were better understood and becoming more systematic, as evidenced by the publication of Yourdon's text *Modern Structured Analysis* in 1988 [13], and so too was the activity of software design, as evidenced by the publication of texts in the UK describing the early versions of SSADM (eg [14]).

Over the next ten years these improvements in the practice of SE gathered pace, as competition between alternative approaches led to the so-called "methodology wars", which ended with the mergers during 1994 and 1995 of the three main competitors into a single organisation (Rational Software). Following these mergers a unified modelling language (UML) was created, along with a development method (the unified process) based round its use: some of the first textbooks covering the use of UML appeared in 1997 [15], and the first editions of the three definitive UML texts were published in 1999 [16, 17, 18]. These steps represented enormous improvements in the technology of SD, by making all of the activities within it much more systematic than they had previously been. As a consequence, by 2001 practice in SD was far more disciplined, and hence far more amenable to being taught, than it had been just ten years earlier, and it has continued to develop further since then.

3.6 Curricula versus Practice

Despite the practical importance of these improvements in the methods of SD, and in particular the importance of the concepts and technologies associated with UML, the phrase UML only occurs three times in CS2001. Two of these occurrences are in the knowledge unit *IM3 Data Modelling*, where UML is suggested as an alternative to entity-relationship diagrams, and the other one is in one of the flavours of the course CS102, namely the version called *Objects and Data Abstraction*. Here UML use cases are suggested as one of the modelling tools that might be used in teaching the design aspects of object-oriented programming, along with class diagrams, although UML is not mentioned in relation to these. Similarly, state diagrams are mentioned in the context of operating system scheduling, but again no connection is made between these and UML, even though UML includes such diagrams. As for the unified process, this is not mentioned at all.

Hence, this model curriculum for CS comes very close to ignoring completely these huge steps forward in the practice of SD. One can only speculate on why this was: presumably it was somehow considered that they were really part of SE rather than CS, even though they were not mentioned in the SE knowledge area either. This is a completely erroneous view, though, for while the most obvious aspect of UML may well be its diagrammatic notations, arguably the more important features of it and of the technologies related to it are the models for the software structures that are represented in these diagrams. Hence, while the use of the diagrams may be so fundamental to SE that it is also fundamental to SD, the meanings of the diagrams are equally fundamental to CS, for they relate to the very structures of the software that CS in particular, and informatics in general, claim to study.

Indeed, the only property that differentiates the models of software structures that underpin UML from the models underpinning the study of simple data structures or algorithms is that of scale. Large scale is, however, now an integral feature of nearly all real software, and so it would be completely unrealistic to argue that large-scale software structures might somehow be less deserving of study than small-scale ones: if anything, their potential for greater complexity makes it even more essential that the fundamental features of them are well understood. Hence, a key part of the argument for extending the core of informatics from just programming to the whole of SD is that educators can not afford to ignore these larger-scale CS structures that underpin the whole of SD, at the expense of concentrating on the smaller-scale structures that are characteristic of programming.

4. The Curriculum for Software Development

This description of how SD has evolved, and of some of the reasons for regarding it is important, gives a partial indication of the material that needs to be included in the curriculum for informatics if SD is to be treated as its core, and in particular it has emphasised the need for the knowledge element of SD to include study of the large-scale structures of software as well as the small-scale structures that form part of the topic of programming. The key feature of the curriculum for SD is, though, that it is one of the most practical parts of informatics, and so its focus has to be on the skills in developing software that students acquire, as much as on the knowledge that underpins these skills. Hence, the best way of structuring this description of the curriculum for SD is in terms of the main stages in the SD lifecycle that students need to be able to work through.

4.1 Required Knowledge and Skills

The first of these stages is requirements analysis, and this involves two sets of skills. One set is concerned with eliciting, from the client for a system, information that will contribute to the requirements for it, through activities such as interviews and other structured meetings, reading and analysis of background documentation, observation, etc. The other set is concerned with creating models from the information that has been obtained so as to present it in a structured form, where (as already indicated) the *de facto* format for these models is now to express them as UML diagrams: class diagrams to describe the business data that a system must handle, and use case and activity diagrams to describe the functions that a system must be able to carry out. The skill of constructing these models is therefore a key element for SD, but students also need to be able to create the contextual documentation for these models in some appropriate format, such as traditional requirements documents, story cards as used in extreme programming [19], or any suitable combination of these.

The second stage in the lifecycle is specification, but this does not necessarily mean formal specification. The starting point for it is that, while a requirements document describes the functions that a system should perform, it rarely defines precisely enough the effects that any function should have on the data that the system is handling. Hence, there are three main skills that students need to develop for this stage. One is the ability to analyse a set of requirements, so as to determine what extra information is needed about the effects that functions should have. The second is the ability to elicit the information needed to provide the necessary detail, which may involve explaining to a client the consequence of different possible choices that could be made. The third is the ability to document this information in some format that is at least suitably structured, even if not necessarily fully formal.

The third stage in the lifecycle is software design, where the limitations inherent in SD mean that architectural design is likely to reduce to selecting whatever standard architecture is the normal one in the domain of the systems that are being studied. Hence, the main skill that students need for this stage is the ability to create a detailed design for a system within the framework, and in particular to create relevant models to explain the operation of a design. These again will typically be UML diagrams, such as package diagrams to show the static structure of a system and collaboration diagrams (most commonly sequence diagrams) to show its dynamic operation.

As well as these skills, this stage will typically require students to apply knowledge about various kinds of components that may need to be included in typical software systems. For instance, almost every system will require some kind of user interface, preferably menu-driven and quite possibly graphical, and so students will need to know about the technology and tools typically used to build such interfaces. An embedded system will also have to handle inputs from sensors and outputs to actuators, and so students constructing systems in this application domain will need to know about the characteristics and operation of these. On the other hand, such systems may only have very simple requirements for storing data persistently, whereas systems in other application domains are more likely to require either a relational database or a simple object-oriented database for this purpose, or some kind of knowledge base for artificial intelligence systems. Thus, students will need to know about the appropriate choice of storage technologies, and also about basic aspects of operating systems, networking, data structures, algorithms and APIs for these structures, as systems within the scope of SD may need to use these in their design as well.

The fourth stage in the lifecycle is software construction, which mainly involves the programming activity that is already well covered. Historically system testing then formed the fifth stage in the lifecycle, but modern practice is that system testing and construction are carried out together, with unit testing accompanying the programming of individual units of code such as classes, and integration testing accompanying the assembly of these into larger units of code, so that only final acceptance testing needs to be carried out once construction is complete. Thus, the additional skill that students need for this stage is the ability to design appropriate test cases and then apply them systematically, at each of the levels of detail during the construction and final evaluation of a system.

The final stage in the lifecycle is system deployment, which for the kinds of systems within the scope of SD typically consist of just two technical steps: system installation and system initialisation. Furthermore, the restrictions on the scope of SD mean that typically installation is done just by copying files from one location to another, either manually or by means of a simple script. Similarly, any initialisation needed for a system is done by running a program which forms part of that system, and which creates any initial data that may be needed by the rest of it. Hence, the knowledge and skills required for these technical steps are simply the same as those required for the design and construction stages.

As well as these technical steps, though, system deployment also involves the provision of appropriate documentation for a system, not least to explain to users how to install and initialise it. The creation of documentation that explains what users need to know, and in terms that they can understand, is very definitely a different skill from those required in the earlier stages of the lifecycle. Indeed, it is sometimes left to specialist technical writers, but system developers have to be able to communicate effectively with technical writers, and so some experience of actually creating documentation is necessary to help them understand what is involved in this activity. Of course, it shares some aspects with the communication skills needed for requirements elicitation, in that both involve being able to understand the viewpoint of users who may not have much technical knowledge of computing, but who do understand the tasks that they want a software system to perform. Fundamentally, though, the two differ in that requirements elicitation is concerned with trying to obtain information from users about a system that does not yet exist, whereas documentation is concerned with conveying information to users about a system that has just been created.

4.2 Relationships with Curriculum Models

The important feature about this material is that it is already included in the various CC2001 volumes, so that treating SD as the core of informatics does not involve adding huge amounts of material to the present cores of these curricula. For instance, the basics of software processes and of the various stages in the process are already in the SE area of CS2001, the basics of interface construction are already in its HCI area, and the various kinds of component technologies mentioned above have areas that cover their basics too, such as Information Management, Operating Systems, Net-Centric Computing, etc. Thus, to make SD the core of the discipline just needs extra time to be allocated to three aspects.

Firstly, for some of these topics SD requires students to be able to apply them, rather than just knowing about them, and so they will need to be covered and practiced in more detail. Secondly, since some of these topics are concerned with larger-scale structures in CS, some time may need to be spent on the relationships between the different scales of structure that exist. These two aspects will therefore require some learning outcomes being revised to address higher levels within Bloom's taxonomy [20]. Thirdly, these topics need bringing together into a coherent whole, but this mainly involves adapting the structures of courses rather than the curriculum content, as discussed below.

Similarly, the domain of embedded systems comes within the scope of CE, and here the equivalent material is already included in the corresponding areas of the CC2001 volume for CE [21] (CE2004 from now on), because these areas are taken directly from CS2001, and so the same comment about adaptation applies to them. By contrast, for information systems (IS from now on) the CC2001 volume [22] (IS2002 from now on) has a rather different structure, with three top-level units to its body of knowledge, of which one is entitled *Theory and Development of Systems*, but while this is oriented to a more general view of systems, it does also naturally contain the topics described above as necessary for SD. Hence the changes that would be required here would be similar to those needed for CS and CE.

The only exception to this situation is information technology, where the body of knowledge in the CC2001 volume [23] has a broadly similar structure to those of the CS and CE volumes, but does not have any knowledge area for SE or SD. Hence, a lot of such material would have to be added to this model to provide adequate coverage of SD. By contrast, SE2004 naturally includes all of the material needed for SD, since (as explained earlier) SD is a restricted subset of SE, and so there is no additional content required, although [3] did conclude that some rearrangement of the packaging of material into courses would provide better support for focusing on SD in the early stages of the curriculum.

4.3 Relationships with Course Structures

While providing adequate coverage of SD would only require small amounts of material to be added to the curriculum models for most of the disciplines, if SD is to form a coherent core then (as already indicated) this has to be reflected in the course structures as well, which will require rather more significant changes. Specifically, by comparison with CS2001, it will no longer be appropriate to begin with a sequence of courses such as CS101, CS102 and CS103, and then CS290, all of which are largely oriented towards programming, before beginning to study in detail concepts such as the process of SD, or the way in which it integrates topics from these previous courses. Apart from the rest of CS290, CS2001 defers these to advanced courses, for which it proposes titles (eg *CS390 Advanced Software Development*), but does not specify any details of the expected structure or content. In arguing that these course structures need to change, though, it is not intended to suggest that the material covered in courses such as CS101 to CS103 is unimportant, since that is clearly not the case. Rather, this material needs to be seen by the students in its context as part of SD, and this means that this context needs to be introduced early as well.

One model for how this might be done is provided by SE2004, which defines a sequence of four core courses. This begins with SE101 and SE102, which cover broadly similar material to CS101 and CS102, but where the intention is described as "*... this course would start with the SE material, and teach all the material as a means to the end of solving software engineering problems for customers*". These first two courses do therefore provide quite a lot of the context for the programming-oriented material, and this is then developed to give a more comprehensive treatment of SD in the second half of the sequence, which is the pair of courses SE200 and SE201.

An alternative to this sequential model in SE2004 is the one developed by the author and colleagues in the Department of Computer Science at the University of Sheffield, which is based on the observation that the large-scale structures and models used in SD are sufficiently separate from the smaller-scale models associated with programming that the two can be introduced alongside each other, as illustrated in table 1. Here, the conventional sequence of two courses labelled CS101 and CS102 runs alongside the sequence labelled SD101 and SD102, which starts with a general introduction to SD and then works through the stages of the lifecycle. Thus, CS101 and SD101 are almost completely independent, but SD102 needs to reflect the relationships between programming and software design, and the role of programming in the construction process, so that it must either parallel with CS102 or follow on from it.

Table 1 A core course structure for software development.

Introductory Course Sequences (in parallel)	
CS101 Introduction to Programming →	CS102 Data Structures & Algorithms
SD101 Software Lifecycles & Requirements →	SD102 Software Design & Construction
Intermediate Courses (order not specified)	
Databases (eg CS270),	HCI and User interfaces (eg CS250),
Methods for Software Analysis & Design,	Integrating Project

The intermediate courses in table 1 are then based largely on those in CS2001, but incorporating the kinds of modifications discussed in section 4.2, meaning that the course on methods for software analysis and design would be significantly different from CS290. Also an integrating project course is proposed, which is not meant to detract from the role of capstone projects, but rather to recognise that because they are capstones they come too late in the curriculum for the practical lessons learnt to have any impact on the rest of what is being studied. Hence, the purpose of this integrating project is to give students a coherent view of SD, by putting into practice what they are learning about it in other courses, and also to help prepare students for undertaking their capstone projects.

5. The Importance of Software Development

The main conclusion that is being drawn from the historical evolution of SD is that a gap has emerged between current practices in teaching informatics and actually doing it, as these are represented respectively by model curricula and by commercial and industrial approaches to creating software. The argument so far has then been that this is a gap which needs to be filled, but a sceptical educator might well ask just how important it actually is to fill this gap, and there are three main points that need to be made in response to this.

Firstly, there is an internal issue within informatics concerning the relationships between education and industry. In any discipline, undergraduate degree programmes need to have educational goals that are wider than simply preparing students for employment within that discipline, but equally these programmes must take proper account of those goals that reflect the needs of potential employers of their graduates. For informatics programmes, where a high proportion of graduates do go into the computing industry, this means that it is important to pay attention to the range of skills that employers in this industry require. Essentially the computing industry is concerned with all aspects of SD, and not just with programming, and so this means that the industry requires the full range of SD skills, and not just programming skills. Of course, the range of activities within the industry go far beyond SD as it has been defined here, and so it also requires a huge variety of knowledge and skills that go beyond those of SD, in all sorts of ways, but these are additional to the core of the discipline and so outside the scope of this argument, which is concerned just with this core. In this framework the function of this core is to ensure that students gain a realistic minimal preparation for their future employment, and this need for realism requires the broader range of skills represented by SD to be provided, rather than just a narrow focus on programming. Hence, to achieve this the core of informatics needs to be widened to consist of SD.

Secondly, widening the core of informatics to cover the broader range of knowledge and skills needed for SD is also important for the external view of informatics as a discipline. In recent years falling enrolments for degree programmes in informatics have emphasised that this external view is not healthy, and there are various reasons for this, many of which have little to do with the curriculum content or structure of degree programmes. One reason that is closely related to these features, however, is the “geek” image of computing in general, and the activities of creating software in particular. At the heart of this image is the view that informatics is concerned purely with technology, which has arisen largely because it has been seen to be equated with programming, and with the narrow set of technical skills needed for programming. If the core of informatics is broadened, so as to encompass the much wider range of skills and styles of working that are needed for SD, then the whole of informatics can gain the same benefits in avoiding this “geek” image that have been claimed to arise for SE from its wider scope [24].

Thirdly, for many informatics degree programmes there is an issue which is related to that of enrolments, namely retention, in that worryingly large numbers of the students who start off by studying informatics then switch away from it to studying other subjects instead. Again there are various reasons for this, some of which are related to the technical difficulty of the activity of programming, but one of these reasons is that students do gain the impression, from the nature of the programming tasks that they are asked to carry out, that they are only seeing part of the process. That is, they can see that the programs which they are asked to write will perform some task, but in many cases they will not be able to see that this task is either useful or realistic.

By contrast, if SD is taken as the core, and the treatment of it includes any kind of practical project, then the starting point that the students see is the elicitation of the requirements for some system that clearly is meant to perform a task which should be either realistic or useful, or preferably both, and this immediately addresses one problem. Also, they then see the process right through to the end, where a system has been created and can be deployed, so that as a result of their efforts they can now see this task actually being performed by this system. This therefore provides them with a much more satisfactory overall experience, and if students feel satisfied with what they have learnt of a discipline they are more likely to want to continue to study it than if their experiences have been unsatisfactory. This will, of course, act more effectively if this project comes earlier in the curriculum, which is part of the argument for not leaving it until the capstone of the course. Also, another factor in making such a project act more effectively is the extent to which the system being developed is to carry out a task that is both realistic and useful, where the most satisfactory situation is for the project to be producing a system for a real client. On the other hand, the involvement of real clients raises a variety of other issues, which while relevant to the teaching of SD have to be regarded as outside the scope of this paper, and thus not pursued any further here.

6. Informatics Beyond Software Development

An important feature of the argument that SD should be treated as the core of informatics is that it is not just restricted to one of the disciplines within informatics, such as CS, but applies to all of them. This could be seen as trying to blur the distinctions between the different disciplines within informatics, and while there might be arguments to be made that a greater unity of these different disciplines would be a good thing, they would be a distraction from the main argument of this paper, and so will not be pursued. Rather, what does need to be considered is whether treating SD as a common core will actually blur these distinctions to any significant extent, or whether the disciplines would still retain their individuality.

A key issue here is that there are actually two separate notions of core, and these need to be distinguished. One notion is associated with the individual disciplines within informatics, which each have their own formal cores, which are precisely defined in their model curricula in terms of the knowledge units that form them and the amount of material from each of these units that ought to be covered. The second notion is the one that this paper is mainly concerned with, and is much less formal, in that it applies to the role of larger constructs (eg programming or SD) within the whole of informatics. Hence, the kind of relationship that is being suggested here is that this informal core should largely be contained within the formal core for each discipline, but that the differences between the disciplines will affect to some extent the way in which the larger ideas are interpreted: for instance, by affecting the choice of application domain for SD. Hence, the informal core will not just be the intersection of the various formal cores, or even some subset of this intersection, but may vary to some extent between the disciplines.

Consequently, the formal core of each discipline can be regarded as consisting of two components. One of these components is the material required for SD, and so represents this informal core of informatics, while also reflecting the application domains that are specific to the discipline. The other component for each discipline is the material that is additional to this informal core, and that is included in the formal core of that discipline in order to reflect the nature of the discipline. Of course, this additional material may also be related to SD in some way – indeed, given the efforts that were made during the creation of the various volumes of CC2001 to ensure that their cores were coherent it would be surprising if it were not – but in each case these relationships are not the reasons why this material is in the formal core, since these derive purely from the philosophies of the various disciplines.

On the other hand, the fact that these relationships exist between the informal core of SD and the formal core of each discipline does serve to indicate the importance of SD, and in practice such relationships take two forms. One form is where material extends the scope of SD by widening the range of software systems that students would be capable of developing, such as the units from the *Intelligent Systems* area in CS2001. Similarly, in the case of CE2004 the units from the *Digital Logic* or *Digital Signal Processing* areas would be examples of this, as they would extend the kinds of candidate systems from simple embedded ones towards firmware systems or to ones that form specialised processing units.

The other form of these relationships is where material will strengthen the understanding that students have of the technologies that are applied in SD, by examining them in more detail than is needed for simply using them within SD. Examples of this in CS2001 would be the units from knowledge areas such as *Architecture and Organization*, *Operating Systems* or *Net-Centric Computing*, and similarly in CE2004 the units from the *Circuits and Signals* or *Electronics* areas could be seen in this way, although they could also be viewed as extending the range of candidate systems even further into hardware.

Similar examples to these could also be identified in the other disciplines within informatics. For instance, in IS a key issue is that of how the requirements for software systems arise from problems situations within business organisations, as identified using methodologies such as soft systems analysis [25]. Such activities are outside the scope of SD itself as it has been defined here, not least because sometimes the result of them is to identify that actually the solution to the problem that has been identified is to reorganise some aspects of the business, and that until this has been done the last thing that is needed is to try to create or introduce some new piece of software. Similarly, in SE some of the core topics are concerned specifically with going beyond SD to consider quantitative aspects of products or processes, because the essential difference between SE and SD is that the former involves these quantitative aspects whereas the latter is explicitly restricted to excluding them. Also, in each case both IS and SE include in their formal cores units that provide deeper underpinning to the SD material, such as the IS2002 units on *Computer Architectures*, *Operating Systems* and *Telecommunications*, or the SE2004 topics that import the equivalent units from CS2001.

7. Conclusions

There are therefore seven main conclusions to be drawn, where the most basic is that there is a clearly identifiable set of activities that form SD, of which programming is just one. Furthermore, while this set of activities is common to all of the disciplines within informatics, it can be defined rigorously in a way that reflects for each discipline the nature of that discipline, and the application domains that are specific to it.

Secondly, this whole set of activities has actually existed and been recognised since the first curriculum model was created for what we now call informatics, even if at that time programming was the only activity in the set for which enough systematic material existed that it could be defined in detail in a curriculum. Thus, thirdly the discussion of how SD methods have evolved has shown that this situation has now changed as the technology of SD has become established. Fourthly, the analysis of CS2001 in particular has shown that this evolution has not been adequately reflected in most of the curriculum models, so that there is now a significant gap between these models and the practice of SD.

Fifthly, this gap between curriculum models and commercial or industrial practice is one that needs to be bridged, for several reasons. One is that the technology being ignored concerns models of large-scale software structures, and these are just as important for understanding informatics as the models of small-scale structures that underpin programming. Another reason is that giving the other activities within SD equal important with programming will have great benefits for both the internal and external images of informatics. Internally it will make the overall balance of material that is studied more realistic, and hence prepare students better for future employment. Externally it will widen the range of skills that students must develop, away from the focus on purely technical skills that has led to the damaging “geek” image of informatics, and by giving students a more complete experience of what informatics is about it may help to improve retention within degree programmes.

Sixthly, for most of the disciplines within informatics bridging this gap will not actually require much change in the content defined by their curriculum models. What will require change is the course structures, since these currently leave the introduction of the material relating to activities other than programming until too late. In particular, it has been shown that some of the concepts of large-scale structures in informatics, which underpin SD, are sufficiently different from the smaller-scale structures of programming that the two can be introduced in parallel without causing confusion. Finally, since treating SD as the core of informatics does not require changes to the curriculum content, it also does not have any adverse effect on any of the individual disciplines, because it does not affect those parts of their formal cores that are not part of SD, but that do characterise the various disciplines.

Hence, this paper has been able to make the case that SD should be treated as the core of informatics, but there is still a lot of further work that needs to be done for this to become a common approach. In particular, this paper has only been able to outline the kinds of changes to course structures that would be required, whereas what would be needed would be a set of course structures that were defined in as much detail as the existing proposals in the various CC2001 volumes. This will be a necessary further step to making SD the core of informatics: and then all that will remain will be to convince enough other informatics educators that they should move in this direction. Not an easy task, but certainly a worthwhile one!

References

1 Tucker A B et al. Computing Curricula 1991: Report of the ACM/IEEE-CS Joint Curriculum Task Force. IEEE Computer Society, 1991.

2 ACM/IEEE, The Joint Task Force on Computing Curricula. Computing Curricula 2001 Computer Science. ACM/IEEE, 2001. Also at <http://acm.org/education/curric_vols/cc2001.pdf>.

3 Cowling A J. Stages in Teaching Software Design. Proc. 20th Conf. on Software Engineering Education and Training, IEEE Computer Society Press, 2007; 141–148.

4 Bennedsen J & Caspersen M E. Revealing the Programming Process. Proc. SIGCSE ’05, ACM Press, 2005; 186-190.

5 Bennedsen J & Caspersen M E. Programming in Context – A Model-First Approach to CS1. Proc. SIGCSE ’04, ACM Press, 2004; 477-481.

6 Cowling A J. What Should Graduating Software Engineers Be Able To Do? Proc. 16th Conf. on Software Engineering Education and Training, IEEE Computer Society Press, 2003; 88–98.

7 Cowling AJ. The Role of Application Domains in Informatics Curricula, Proc. 2nd Informatics Education Europe Conf., SEERC, 2007; 166–175.

8 Atchison W F et al. CURRICULUM 68: Recommendations for Academic Programs in Computer Science. CACM 1968; 11.3: 151–197.

9 Naur P & Randell B. Software Engineering: Report on a conference. NATO, 1968. Also at <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF>.

10 Austing R H et al. CURRICULUM '78: Recommendations for the Undergraduate Program in Computer Science. CACM 1979; 22.3: 147–166.

11 Denning P J et al. Computing as a Discipline. CACM 1989; 32.1: 9–23.

12 ACM/IEEE, The Joint Task Force on Computing Curricula. Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. ACM/IEEE, 2004. Also at <http://sites.computer.org/ccse/SE2004Volume.pdf>.

13 Yourdon E. Modern Structured Analysis. Prentice-Hall, 1988.

14 Downs E, Clare P & Coe I. Structured Systems Analysis and Design Method: Application and Context. Prentice-Hall, 1988.

15 Fowler M with Scott K. UML Distilled: Applying the Standard Object Modelling Language. Addison Wesley, 1997.

16 Booch G, Rumbaugh J & Jacobson I. The Unified Modelling Language User Guide. Addison Wesley, 1999.

17 Rumbaugh J, Jacobson I & Booch G. The Unified Modelling Language Reference Manual. Addison Wesley, 1999.

18 Jacobson I, Booch G & Rumbaugh J. The Unified Software Development Process. Addison Wesley, 1999.

19 Beck K with Andres C. Extreme programming explained: embrace change. Addison Wesley, 2005.

20 Bloom B S (ed). Taxonomy of Educational Objectives: The Classification of Educational Goals, Handbook 1: Cognitive Domain. McKay, 1956.

21 ACM/IEEE, The Joint Task Force on Computing Curricula. Computer Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering, ACM/IEEE, 2004. Also at <http://www.acm.org/education/CE-Final%20Report.pdf>.

22 Gorgone J T et al. IS 2002: Model Curriculum and Guidelines for Undergraduate Degree Programs in Information Systems, ACM/AIS/AITP, 2002. Also at <http://www.acm.org/education/is2002.pdf>.

23 ACM SIGITE, Curriculum Committee, Computing Curricula: Information Technology Volume (Draft dated October 20, 2005), ACM, 2005. Also at <http://www.acm.org/education/curric_vols/IT_October_2005.pdf>.

24 Williams L. Debunking the Geek Stereotype with Software Engineering Education. Proc. 18th Conf. on Software Engineering Education and Training, IEEE Computer Society Press, 2005; 4.

25 Checkland P. Soft systems methodology : a 30-year retrospective. Wiley, 1999.

Bridging Classroom Heterogeneity: A Software Engineering Course and Projects

Ani Nahapetian[1]

[1]*California State University, 1000 E. Victoria St., Carson, CA 90747, ani@csudh.edu*

In this paper, the process of bridging classroom heterogeneity in a Software Engineering course is discussed, using a real course as a framework for analysis. Specifically, this paper addresses issues when disparities exist in the same classroom, 1) between graduate and undergraduate students, 2) among students with a variety of programming skills and programming language familiarities, and 3) among student experience levels in software development. Additionally, the struggles of aiming for substance, while dealing with the perils of group work, are addressed.

This paper presents real and practical solutions to these challenges, including addressing issues with course content presentation, textbook selection, course projects, and graduate research opportunities.

Keywords
Software Engineering, Computer Science Pedagogy.

1. Introduction

Software Engineering courses exhibit a large degree of heterogeneity in terms of student abilities, skills, and motivation. This is especially true in the following situations:

1) the undergraduate course is co-located with a graduate version of the course;

2) some students have years of industry experience while others have learned to program only recently and have no programming experience outside of the classroom;

3) students approach the course with different expectations about what they will learn and the work they will do for the class;

4) students have different familiarities with programming languages and programming environments.

In this paper, I discuss the various approaches used to develop a Software Engineering course co-located with a graduate level advanced Software Engineering course.

Various conflicting interests presented themselves in developing the curriculum, and specifically the projects for these two courses. First, of course there was an issue with having a graduate level course and an undergraduate level course co-located. Secondly, I felt that a group project was a critical component to the course, but I wanted to avoid the hurdles faced when students worked in dysfunctional teams. The "I did all the work" syndrome could have a significant effect on students' impressions of the course material. Thirdly, the students needed to be exposed to the basic concepts of Software Engineering, so they would appreciate that software development is much more than just writing code in a programming language.

In the rest of this paper, I present the challenges faced when preparing the course curriculum and syllabus for a diverse group of Software Engineering students in a single classroom. I present a survey of Software Engineering course syllabi and their use of textbooks and other reading material. I then present the various new and interesting approaches I employed in my classroom to address these issues.

2. Inherent Hurdles

2.1 Co-location of Graduate and Undergraduate Classes

For the first time in my department, a co-located Graduate Advanced Software Engineering course was offered at the same time and location as the undergraduate course. Though the undergraduate course had been offered previously the graduate course was entirely new, along with the brand new Masters degree program.

As would be expected this arrangement presented several hurdles. The first of which was the fact that the graduate students were expected to have previously taken an undergraduate Software Engineering course.

2.2 Variety of Programming Skills

There were a variety of programming skills in the classroom. My university carries out its introductory programming course mostly in Java. Though, one elective course on Programming in C in the UNIX environment is also offered. Additionally, there is a course that acts as a preparatory course to the undergraduate curriculum. This course has been taught in several languages including Visual Basic, Matlab, and Alice.

Despite this emphasis on Java, there are large numbers of transfer students who have studied C++ at their original college or university. Additionally, there are students who have a great deal of industry experience, who are most familiar with the programming skills acquired at their workplace. For example, in a classroom it is possible to find students well versed in SQL, but weary of their background in Java.

This presented several distinct options for the projects:

1) allow students to work in any platform they are comfortable, and then deal with compliance issues within groups as they arise;

2) impose one programming language on all the students;

3) offer a small set of popular programming languages, and compose the groups accordingly;

4) use a less known or proprietary programming language to level the playing field.

2.3 Dealing with Not Having Been in the Trenches

Software Engineering unlike other Computer Science fields is based on a large volume of empirical knowledge. Also to those with experience, critical issues and their solutions sometimes seem obvious. Students generally enroll in the course, once they complete their introductory programming series, and so they may take the course as early as their sophomore year.

In my Software Engineering course, there were students who had been working in industry for many years and were able to gleam quite a bit of benefit from discussions regarding practical aspects of Software Engineering, as compared with the students who had followed a direct path from high school to this course offering.

This presented a challenging dichotomy in the class. Some students, especially those with years of government industry experience, were very well versed in the techniques presented in the class, while others, including some of the graduate students, had no experience outside of the classroom and hence were learning the material from scratch.

Dealing with this issue was the most challenging hurdle faced in the course, as it spanned all the other disparities, including class rank and programming skills.

2.4 Avoiding Perils of Group Work while Learning to Work in a Team

There is research regarding the perils of group work [Waite04]. Invariably, when a large disparity between programming ability within a group exists, less experienced students defer an uneven amount of work to their more advanced programming classmates. Though opportunities to learn from each other are presented in these situations, students are not always the most adept at apportioning the work and aiding each other. As a result, learning opportunities are unevenly divided among the students, which can also lead to resentment.

On the other hand, working in a team is critical for a Software Engineering course. Industry demands teamwork and Software Engineering curriculum specifically addresses issues related to group software development. Students, during the job hiring process, are evaluated on their ability to work in or lead large and small teams. This is especially highlighted by our university's close proximity to large aerospace corporations. Note that software products in the aerospace industry are highly regulated and as a result use Software Engineering practices to a greater extent.

The key challenge here becomes creating opportunities for group work, while still overcoming the programming experience disparities among students.

2.5 Aiming for Substance

Software Engineering is often condescendingly considered a “pseudoscience,” and Software Engineering courses have a reputation for being the least technical of the Computer Science department’s offerings. E. W. Dijkstra is famous for saying “The required techniques of effective reasoning are pretty formal, but as long as programming is done by people that don't master them, the software crisis will remain with us and will be considered an incurable disease. And you know what incurable diseases do: they invite the quacks and charlatans in, who in this case take the form of Software Engineering gurus.” [Dijkstra]

The field of Software Engineering has advanced significantly since the time when this statement was made. Corporate hiring managers and university industry advisors strongly encourage and even demand that students complete Software Engineering courses at the undergraduate level.

Thus a key challenge for the course is to make clear to students that the seemingly basic topics that are the basis of a solid Software Engineering practice are truly valuable and have been developed at a great cost. Student resistance to such basic concepts as documentation and configuration management is prevalent and considerable. A key challenge is presenting the material so that students do not view them as unnecessary and time-consuming prescriptions.

2.6 Learning to Present Technical Material

Both industry and academia demand that students have strong presentation skills, including clarity, ease, and effectiveness. Software engineers are expected to engage in public speaking to handle code reviews, design reviews, requirement reviews, demonstrations, and more. Thus incorporating presentation opportunities became an important consideration in my Software Engineering course development.

3. Textbooks

3.1 Not for Inexperienced Software Developers

Software Engineering literature is often written for engineers with industry and large scale software development experience. For a large majority of the undergraduate and even graduate students in my class, this is not a correct assumption. Students often take Software Engineering right along with their Computer Architecture and Computer Networking courses. There is no large time gap between the students’ initial introduction to programming and their discussion of Software Engineering. Additionally, Software Engineering curriculum has a focus on managerial duties that may be new to even experienced programmers.

The search for a textbook or other reference material is a challenge that needs to consider student experience levels, while still aiming for the best and the latest empirical approaches that Software Engineering has to offer.

3.2 Survey of Online Courses

Software Engineering courses are more likely to not follow a single text, than any other core undergraduate Computer Science course. Table 1 summarizes the textbook information found in six syllabi found on the web after some reasonable amount of web searching. 2 out of 6 online course syllabi did not have a course textbook. None of the online course syllabi used a single textbook. Additionally, 3 out of 6 used published and web articles as part of their curriculum.

Table 1. Software Engineering Textbooks for Various Universities

Reference	**University**	**Offering**	**Text Type**
Text(s) Used			
[CSUN]	CSUN	Fall 2007	Multiple texts
1. Ian Sommerville, *Software Engineering, 8th Edition*, Addison-Wesley Longman Publishing Co., Inc (2007). 2. Dan Pilone and Neil Pitman. *UML 2.0 in a Nutshell, 2nd Edition*, O'Reilly Media, Inc., (2005).			
[CalPoly]	Caly Poly San Luis Obispo	Spring 2007	Articles
1. Supplemental reading			
[Pomona]	Pomona College	Fall 2007	Single text, supplemented with articles
1. Steve McConnell, *Code Complete, Second Edition*, Microsoft Press (2004). 2. Supplemental reading			
[UCLA]	UCLA	Winter 2008	Multiple texts
1. Steve McConnell, *Code Complete, Second Edition*, Microsoft Press (2004). 2. Roger S. Pressman, *Software Engineering: A Practitioner's Approach, 6th edition*, McGraw Hill (2005).			
[Washington]	University of Washington	Spring 2007	Multiple texts
1. Steve McConnell, Software Project Survival Guide. Microsoft Press (1997). 2. Andrew Hunt and David Thomas, *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley Professional (1999).			
[Cornell]	Cornell	Spring 2008	None
1. None, only recommended texts			
[Berkeley]	UC Berkeley	Spring 2008	Articles
1. None, only recommended texts			

4. Project Framework

After considering the course challenges presented in the earlier sections, I chose to address them with the following approaches.

4.1 Course Content Presentation

In an effort to bridge the disparities that existed among the students, I took several different approaches to course content presentation. The various forms of imparting information not only reinforced concepts, but also enlarged the group that was able to follow the material.

Specifically, I employed a comprehensive Software Engineering text [Sommerville07] and also incorporated articles, some from a classic Software Engineering book [Brooks95]. Additionally, I used palatable, but also very powerful, articles on various important coding topics. I emphasized a project development atmosphere in the course, so students would have the opportunities to flex their software development muscles. Finally, I incorporated student presentations into the mix, so that students would be able learn from each others experiences and activities, as well as from the text and me.

Another essential component became classroom discussion. I found that students, even despite their inexperience, still had some basis on which to make valuable comments and well posed arguments. Additionally, the more experience students were able to provide an invaluable perspective about the field and the practice of Software Engineering, that all the students found beneficial. For example, the class was able to hear about how several different local companies' approach the code review process, with experienced students taking on a role similar to that of a guest industry speaker.

4.2 Split Project Nature

I divided the course project into four segments: requirements, design, code, and testing. The first two segments, requirements and design, were carried out as a team of three or four students. The students developed a software project idea and prepared a requirements document for their project.

The next segment of the project was to develop a design for the project. As a group, a high level component based design was developed, where each team member eventually worked to develop a component of the design.

The novelty of my course syllabus lies in the splitting the team for the code development segment. To address the heterogeneity challenges presented in the earlier sections, students wrote their own code without being able to rely on their partners for help. Individual code development with the aim of combining the code components into a larger project is common in industry. It also clarifies the need for practices such as documentation, proper interfacing, component testing, integration, and more. It also addresses the inherent differences between student programming skills which lead to the perils of group work discussed earlier.

In an effort to give students a technical software development project which they would work on individually, I considered the following options before settling on the split project idea. I considered having students 1) edit legacy code, 2) develop a small pieces of open source software, and 3) complete each others code.

4.3 Requirements, Design, and Code Reviews

In an effort to provide technical presentation practice, the first three segments of the project culminated with presentations, namely requirements review, design review, and code review. Students presented as a group, or individually in the case of the code review, according to the format of industry review meetings. All members of the class were expected to comment during the review, and students had to implement the instructor's comments in the next phase of the project.

The reviews provide a forum for students to learn technical presentation skills, to engage in the common practice of software reviews, and to learn to handle questions and comments on their work. Additionally, it is an opportunity for students to learn their role as a member of a review committee.

4.4 Reading Research Material

To encourage students to approach Software Engineering not only as a topic to learn, but also as an accessible field about which to do research, I decided to expose graduate and undergraduate students to Software Engineering research literature. The topics in Software Engineering literature tend to be more palatable to wider audiences than other Computer Science fields. Additionally, contributions from software engineers in the field are both practical and informative.

Students read and analyzed several articles throughout the semester. The first article was used as part of a take home midterm, to ensure that students read the article fully and were given an opportunity to express their ideas on the material. The article was a publication on code reviews, thoughtfully written by an industry software developer in very plain language [Wiegers98]. It was the first Computer Science research article that many students had ever read, and since the material presented in the article was relevant to the students' lives and also easy to understand, I believe that it helped form a positive impression of Computer Science research and became a feasible entry point into the world of research. This exercise was the best received assignment during the semester.

Throughout the course, students were asked to read selected Software Engineering excerpts from F.P. Brooks' famous text The Mythical Man Month [Brooks02]. Though the text is a classic and an essential part of Software Engineering, it did not give the students a sense of the current interests in the field, the way the research articles did.

4.5 Graduate Research Project

The standard approach for handling graduate course work, in a largely undergraduate class is to give extra work to the graduate students. However, I had reservations about that approach, since I felt that the graduate students might get bored, that it was the not be the best use of their time, and that in turn might detract from the undergraduate learning experience.

In the end, I gave the graduate students a semester long research project assignment in place of the final exam. The graduate students participated in all other parts of the course, including the projects and the midterm. Outside of the classroom on an individual basis, the graduate students and I discussed the research project, and the work was presented to the entire class at the end of the semester. The work carried out for the research project involved choosing a research idea in Software Engineering and working towards and preparing a conference level research article on the idea.

4.6 Scheduling Graduate Work

I grappled with the idea of ending the undergraduate class session early to introduce and engage in discussions about graduate concepts. Though, this presents an opportunity to present graduate level work, it does cut the undergraduate lecture time short. Having a small number of graduate students in the course permitted some of the graduate needs to be addressed outside the classroom. In the end, I decided to hold combined undergraduate and graduate lecture sessions, but to have additional weekly meetings with the graduate students.

Through the process of preparing a graduate research paper, a great deal of interaction occurred between the graduate students and me. I believe this greatly enhanced their course experience. As the graduate program becomes more popular, this may become more difficult to do.

I had one request from a graduate student to do the course software development project individually. I did not agree to this, as the graduate students need opportunities to be involved in a software development group, both for their benefit and that of the undergraduates.

4.7 Final Opt-Out

Due to the split project nature of the course, students do not necessarily need to consolidate their components into the final software project as they envisioned. However, since they should each have a working component of the system, the option to integrate the components is presented to the students. As a reward, the students are allowed to opt out of the final, upon successful completion of their original project proposal.

Some students were highly motivated to take this option, as it provides the satisfaction of completing a large software engineering project. Additionally, it allows the motivated groups to obtain a deeper perspective about the software development process.

Though making the integration part of the project mandatory is enticing, it presents a problem when some group members do not fully accomplish their component requirements. This puts the other group member in a difficult situation. I find that the optional, final opt-out approach is able to handle this situation more fairly, by giving dysfunctional teams a break by taking the final exam instead.

5. Conclusion

In this paper, the process of bridging classroom heterogeneity in a Software Engineering course is discussed. A real course is used as the framework for addressing classroom disparities, in terms of class ranks, programming skills, and industry experience levels. Additionally, approaches to avoid the perils of group work are presented, given this level of classroom heterogeneity.

6. Acknowledgements

I would like to thank Professor Miodrag Potkonjak of UCLA for an interesting discussion regarding Software Engineering textbooks, which greatly contributed to development of my ideas on the subject. Also, I would like to thank Theodor Soneriu for collecting some of the data used in Table 1.

References

[Berkeley] Berkeley Software Engineering Course Website. http://inst.eecs.berkeley.edu/~cs169/sp08/doku.php?id=info

[Brooks95] Brooks, F. P. The Mythical Man-Month (Anniversary Ed.). Addison-Wesley Longman Publishing Co., Inc (1995).

[CalPoly] Cal Poly Software Engineering Course Website. http://www.csc.calpoly.edu/~djanzen/courses/307S07/

[CSUDH] California State University, Dominguez Hills Software Engineering Course Website. http://www.csc.csudh.edu/ani/courses/2007Fall/csc481-581/csc481-581.html

[CSUN] CSUN Software Engineering Course Website. http://www.csun.edu/~twang/380/

[Cornell] Cornel Software Engineering Course Website. http://www.cs.cornell.edu/courses/cs501/2008sp/

[Dijkstra] EWD 1305: Answers to questions from students of Software Engineering. http://www.cs.utexas.edu/users/EWD/transcriptions/EWD13xx/EWD1305.html.

[Pomona] Pomona College Software Engineering Course Website. http://www.cs.pomona.edu/classes/cs121/

[Sommerville07] Sommerville, I.. Software Engineering, 8th Edition, Addison-Wesley Longman Publishing Co., Inc (2007).

[UCLA] UCLA Software Engineering Course Website. http://www.cs.ucla.edu/classes/winter06/cs130/syllabus.html

[Washington] University of Washington Software Engineering Course Website. http://www.cs.washington.edu/education/courses/403/07sp/syllabus403.html

[Waite04] Waite, W. M., Jackson, M. H., Diwan, A., and Leonardi, P. M. Student culture vs group work in computer science. SIGCSE Bull. 36, 1 (Mar. 2004), 12-16.

[Wiegers98] Wiegers, K. 1998. The seven deadly sins of software reviews. Softw. Dev. 6, 3 (Mar. 1998), 44-47.

What Every Software Developer Should Know about Human-Computer Interaction – A Curriculum for a Basic Module in HCI in Informatics Education

Andreas M. Heinecke[1], Friedrich Strauß[2], Astrid Beck[3], Markus Dahm[4], Kai-Christoph Hamborg[5], Rainer Heers[6]

[1]FH Gelsenkirchen, 45877 Gelsenkirchen, Germany, andreas.heinecke@informatik.fh-gelsenkirchen.de

[2]sd&m AG, software design & management, Carl-Wery-Str. 42, 81739 München, Germany, friedrich.strauss@sdm.de

[3]HS Esslingen, Fakultät Informationstechnik, Flandernstr. 101, 73732 Esslingen, Germany, Astrid.Beck@hs-esslingen.de

[4]FH Düsseldorf, Fachbereich Medien, Josef-Gockeln-Str. 9, 40474 Düsseldorf, Germany, markus.dahm@fh-duesseldorf.de

[5]Universität Osnabrück, Universität Osnabrück, Fachbereich Humanwissenschaften, Institut für Psychologie, Arbeits- und Organisationspsychologie, Seminarstr.20, 49069 Osnabrück, Germany, khamborg@uos.de

[6]Visteon Deutschland GmbH, Visteonstr. 4 – 10, 50169 Kerpen, Germany, rheers@visteon.com

Due to legal issues and economic reasons knowledge of human-computer interaction is of growing importance to all people who develop interactive software. Many current study programmes in informatics don't teach HCI sufficiently. The task working group on software ergonomics of the German informatics society (Gesellschaft für Informatik e.V. – GI) has developed a curriculum for a basic module in HCI in order to teach students and others who will design interactive software the most important fundamentals of HCI. This module is intended to give an introduction to software ergonomics and to teach the fundamentals of usability and of user-centred design processes. Its contents and objectives are meant to be an obligatory part of informatics education.

Keywords
Curriculum, Human-Computer Interaction, Learning Objectives, Legal Issues, Standards

1 Introduction

Human-computer interaction is an interdisciplinary field of study and research combining informatics, behavioural sciences, design, and others. Its importance is growing due to legislative and economic reasons. Therefore every software developer should have a basic understanding of human-computer interaction.

1.1 Human-Computer Interaction as a Field of Study

"Human-computer interaction is a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them." [1] It uses methods and results from different fields like informatics, cognitive psychology, linguistics, social sciences, graphic and industrial design, and others.

As far as design of interactive computing systems is concerned the main goal of HCI is to improve usability. Usability denotes the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use [2]. Thus usability is a question both of functionality of an interactive system and of design of its user interface. As in common applications the hardware for interaction is given more or less, improvement of usability focuses on software functionality, on user interface design, and on task allocation between human being and computer. These three topics are often summarized under the term of software ergonomics.

Courses on HCI can be found in all the disciplines that contribute to this field. In most cases these are optional and meant for students who want to specialize in HCI. According to each discipline they may focus on different subjects. Thus not every student specialized in HCI is an expert in developing usable software. As the number of usability experts is limited, we assume that in the foreseeable future there will be still a large number of projects developing interactive software in which no usability experts will participate.

1.2 The Importance of HCI for Software Developers

HCI is of growing importance for software developers who have to fulfill ergonomic requirements which have been set up by legislation during the last two decades in order to improve health protection at work or to prevent discrimination of disabled persons.

In 1990 the European Economic Community passed the Council Directive on the minimum safety and health requirements for work with display screen equipment [3]. It states that "in designing, selecting, commissioning and modifying software, and in designing tasks using display screen equipment, the employer shall take into account" certain principles which can be seen as basic ergonomic requirements. Demanding that "the principles of software ergonomics must be applied" the Directive refers to European and international standards like ISO EN 9241 [4] which define those principles. In the 1990s the Member States of the EEC have brought "into force the laws, regulations and administrative provisions necessary to comply with this Directive". In Germany, e.g., this has been done by the Bildschirmarbeitsverordnung [5].

While the Council Directive deals with general usability of applications for work, there is a second field of applied HCI covered by legislation, namely accessibility. Accessibility addresses the usability of software for people with the widest range of physical, sensory and cognitive abilities, including those who are temporarily disabled, and the elderly. In several countries accessibility of software applications is required by acts against discrimination of disabled persons. Sometimes this applies for web applications only, for applications provided by governmental institutions only, or both. Section 508 of the US Rehabilitation Act [6] defines rules for accessibility of web sites referring to the Web Content Accessibility Guidelines (WCAG) [7]. In Germany, the Barrierefreie Informationstechnik-Verordnung [8] closely follows the WCAG, too. Additionally there are internationals standards [9] or drafts of international standards [10] for accessibility of a wide range of software.

Besides requirements by law there is an economic reason why software developers should take care of usability. If there are different software products for the same tasks the one with better usability obviously will sell better.

2 Goals of a Curriculum for a Basic Module in HCI

There are several curricula for HCI which may be applied in teaching students of informatics who focus on HCI. A curriculum which defines the basic knowledge on HCI for every software developer has been missed so far.

2.1 Learning Outcomes of a Basic Module

In Germany, already in the early 1990s there have been complaints about teaching system developers mainly technical know-how neglecting the aspects of ergonomical design of software [11]. Since then, there have been many demands for the integration of HCI matters into education of software developers. In 1997 Krasemann [12] stated a need for "project management and design methods which are much more end-user oriented". Two years later a task working group of the Gesellschaft für Informatik (GI, the German informatics society) developed recommendations for informatics education at universities to become more application-oriented. They claimed that the students should acquire basic competences in work sciences, above all in analysis and design of working devices and especially in software ergonomics including computer graphics, virtual reality, and multimedia systems [13].

According to the GI recommendations for bachelor and master studies in informatics [14] shall acquire a practical design competence in man-machine interaction. Accreditation procedures of study programmes in informatics should examine the achievement of this learning outcome. One of the German accreditation agencies explicitly adopts this demand claiming bachelors in informatics "are able to model man-machine interfaces adequate to the application and ergonomical" [15]. Although there may be different ways to reach this goal, we assume that a dedicated basic module in HCI will be best. It enables presenting the subject in context and avoids scattering it to many different courses which often leads to neglect.

The module aims at a basic understanding of findings, methods and processes in developing usable software. Students shall have a basic knowledge of software ergonomics, especially of process models and methods for user-centred software development. As software developers they shall be able to avoid serious offences against usability and to fulfil basic requirements of usability. On the other hand, in more complex software projects they shall be able to realize whether participation of usability experts is necessary.

2.2 Target Groups for a Basic Module

The curriculum of the basic module aims primarily at informatics education at universities. Thus students in study programmes with 55% to 70% of informatics (type 1 according to the GI classification [14]) or 40% to 55% of informatics (type 2) make up the main target group. Additionally, all other students who will develop interactive software belong to the target group. This includes students from disciplines like communications design, media design, information technology, psychology, cognitive science and so on.

The basic module in HCI applies not only to universities and universities of applied sciences in campus and distance learning but to colleges and other institutions of education and training as well. It may be used as guidance for further training on the job. The curriculum defines basic knowledge of usability for software developers in software companies or software departments of companies and institutions.

2.3 Differences to Current HCI Curricula

In the USA a Special Interest Group of the ACM started work on HCI curricula in the 1980s [1]. The IFIP Working Group 13.1 "Education in HCI and HCI Curriculum" maintains a website on HCI education resources with an overview of HCI education programmes in several countries [16]. In Germany a GI task working group defined a curriculum in software ergonomics which became an official recommendation of GI in 1993 [17]. This curriculum defines optional courses for students who want to specialize in HCI.

There are two substantial differences between the 1993 recommendation and the curriculum we suggest here. On the one hand, new developments in the field of interactive software have been taken into consideration. This includes new fields of applications (e-business, e-learning, edutainment or computer games, e.g.), new user interfaces (PDAs, wearables or augmented reality, e.g.) and new models and methods of usability engineering. On the other hand, the course aims at an overview about HCI for every software developer and thus has been limited to the necessary basic knowledge.

3 Overview of the Curriculum

The curriculum presented here has been developed by the task working group "Software Ergonomics" of GI. It has been discussed in workshops at several German HCI conferences. After having been reviewed by German experts who are teaching HCI and usability issues it has become an official recommendation of GI [18]. The following chapter presents an overview of its structure and contents and discusses organizational questions of teaching it.

3.1 Structure

Figure 1 gives an overview of how the contents of the basic module have been structured. There are three main blocks. The first one gives an introduction to software ergonomics as an interdisciplinary field of study and research. The second one deals with foundations of design for usability. The third one focuses on models and methods of a user-centred process of software development.

Preface Motivation – Definition – Target Group – Basic Module Specification		
Introduction to Software Ergonomics • Human – Task – Software • History of Software Ergonomics • Standards and Legal Issues	Usability of Software • Human Information Processing and Behaviour • Input- / Output Devices and Interaction Techniques • Design of Work and Tasks	User-centred Design Process • Models of User-centred Development • Requirements Analysis • Specification and Prototyping, Evaluation
References General – Textbooks – Legislation and Guidelines – Special Topics		

Figure 1 Structure of the Curriculum for a Basic Module in HCI.

All three blocks follow the same structure. First there is an introduction to it with an overview of its chapters. Each chapter contains a short introduction, a list of learning objectives, a list of learning contents, and a list of exercises the students may work on.

References primarily refer to books presenting the contents of the curriculum in a way which is adequate for teaching and learning. Furthermore, there is a list of standards and laws a software developer should know. An additional list contains references for those who want to focus on special topics or to broaden their knowledge.

3.2 Workload

The curriculum presented here requires students of informatics in bachelor programmes of type 1 or type 2 to work at least 120 hours on a basic module in HCI. That means, a workload of 4 ECTS credit points should be the minimum for HCI subjects. In study programmes with a focus on design of interactive software like media informatics or communications design, e.g., all chapters of the curriculum should become extended and more detailed requiring students working a longer time. In these cases it may be better to split the subject into several modules.

If a study programme centres on software development of interactive systems, tools and methods for implementation and design of user interfaces and interactive software have to be taught additionally. According to the focus of a study programme, other subjects of HCI like computer supported cooperative work, adaptivity and user modelling, or hardware ergonomics, e.g., have to be added. Then a higher workload has to be allocated.

3.3 Lectures and Exercises

Teaching of the basic module requires at least 30 hours at 45 minutes each. In university teaching this would correspond to a term having 15 weeks with a lecture of 90 minutes each week. In order to acquire competences in designing usable software students should do some practical work like exercises on paper or on screen. Such practical work should focus on ergonomics. That means, necessary skills in software engineering and programming like GUI programming with Java Swing, e.g., have to be taught in other courses.

As a course in continuing education for software developers the basic module can be taught in four to five days.

4 Learning Objectives of the Curriculum

The following chapter explains the learning objectives of the basic module in HCI. For each topic there are a short description of contents and the list of objectives. By reaching the learning outcomes students shall get the necessary knowledge and skills to produce software which fulfils the basic requirements of usability.

4.1 Part 1: Introduction to Software Ergonomics

The first part of the curriculum gives in introduction into HCI and its methods. It shows that the design of usable software has to focus on the demands of the users relating to their tasks within the context of use (technical, organizational and social environment). Here we define fundamental terms like usability, satisfaction, or suitability for the task. Part 1 gives an overview of all the topics covered by this curriculum, too.

Human – Task – Software

As a basic introduction and a first overview of software ergonomics the first chapter of part 1 deals with layers of human-computer interaction, with relations between the design of software and the design of work flows, with different types of applications like office systems, web sites and embedded systems, with the different roles of people involved in software development, and with the opportunities of optimizing user interfaces by software ergonomics.

Students shall

- understand how important an ergonomical user interface is for the quality of a software product,

- realize the influence of software design on the tasks the user has to fulfil using the software,
- know some fields in which software ergonomics are applied,
- know the benefits of a user-centred design process,
- understand that software ergonomics are a necessary part of the software engineering process and thus a part of software quality,
- understand that interdisciplinary knowledge is required for software development.

History of Software Ergonomics

There are two main topics in this chapter, the evolution of computer technology and its influence on human-computer interaction, and the contributions of different disciplines to software ergonomics. Different technologies like terminal systems, personal computers, client-server systems and embedded and mobile systems with different output capabilities led to different user interfaces and different ways of interaction. These, in turn, raised different ergonomic questions. Several disciplines like work sciences, psychology and physiology have contributed their special views and methods to these topics.

Students shall

- know the history of software ergonomics in context of the history of computing,
- know how different disciplines contribute to software ergonomics.

Standards and Legal Issues

The last chapter of the introduction deals with standards which define properties of the software product (ISO 9241 [4], ISO 14915 [19]) or the software development process (ISO 13407 [20]) that are necessary to ensure usability. These standards define essential terms and goals of software ergonomics and contain practical examples for the design of usable software. In many countries there are laws and regulations demanding usability and / or accessibility of software in certain fields of application by providing checklists or referring to special standards like ISO/TS 16071 [9].

Students shall

- be able to quote definitions which are central to human-computer interaction,
- know the standards of software ergonomics and be able to use them for their own work,
- know the legal issues and the fields in which they have to be met,
- know the fields of application in which accessibility is required,
- know about advantages and disadvantages of standardization.

4.2 Part 2: Usability of Software

Part 2 teaches the fundamentals of human-computer interaction. It starts from the human and his or her physiological and psychological properties. The user interacts with the computer via input and output devices. These devices are used to present information and to execute a dialogue. Dialogues are executed in order to use software applications for certain tasks. Thus part 2 ends with a discussion of the design of work and tasks.

Human Information Processing and Behaviour

In order to design adequate means of interaction software developers have to know how humans perceive and process information. Therefore we start with important facts from physiology and cognitive psychology about capabilities and limits of human perception, memory, learning, and acting. This includes topics like colour perception, Gestalt theory, selective attention, mental models, strain and stress, error handling, and cultural and individual differences, for example.

Students shall

- know about performance and limits of human perception and be able to apply such knowledge to the presentation of information,
- be able to apply knowledge of human information processing and behaviour when designing means of interaction,
- know how system design can ease learning the use of interactive systems,
- be able to design systems according to concepts of error avoidance and error management,
- know about the significant differences between users which have to be taken into account in design of interactive systems,
- know about the factors of strain and stress on users of software applications.

Input and Output Devices

Input and output devices determine how users can interact with the computer. Although standard interaction devices like two-dimensional colour display, keyboard and mouse are very common a lesson about their ergonomic requirements is necessary as in practical use there are often ergonomic deficits. Besides standard I/O devices this chapter deals with devices for people with special needs (assistive technologies, e.g.) and with devices for special applications (three-dimensional I/O by immersive and non-immersive devices, e.g.). Depending on the local focus of research and teaching some devices may get more attention than others.

Students shall

- know the important input and output devices, their technical properties and their ergonomic advantages and disadvantages,
- be able to state the ergonomic requirements that input and output devices have to fulfil,
- know the input and output devices for persons with special needs and for non-standard applications,
- be able to choose adequate input and output devices for a given context of use.

Interaction Techniques

This chapter deals with principles and criteria for presentation of information and design of dialogues. As graphical user interfaces with windows, menus and pointing devices (WIMP) are common, an important topic is how to use of widgets, fonts, colours and highlighting. Choosing the adequate type of dialogue (command dialogue, menu dialogue, multimedia dialogue, e.g.), arranging and grouping its elements (menu items, data entry fields, buttons on touch screens, e.g.) and finding adequate structures of information and navigation are crucial points for usability.

Students shall

- know the basic principles and criteria for presentation of information and design of dialogues and be able to apply them to screen design of software applications,
- know the different ways of interaction and the different types of dialogues and be able to choose the adequate ones for a given context of use,
- know the essential rules of design for each type of dialogue and be able to apply them,
- be able to establish usable structures of contents and navigation in information systems,
- be able to choose media and combinations of media which are appropriate to the prospective users,
- know the different means of user guidance and be able to implement them.

Design of Work and Tasks

As software is commonly used in order to fulfil certain tasks, the more software developers know about the tasks of the users the better will be the usability of the software. On the other hand, new software or a new release of existing software often leads to changes of the work-flow or the organisational structures in a company. Therefore this chapter provides the essential knowledge about the design of work and tasks.

Students shall

- know the interdependencies between design of work, tasks and software and be able to explain them,
- know the influence of software design on the design of tasks and work of the users.

4.3 Part 3: User-centred Design Process

The third part deals with usability engineering. It shows how software ergonomics can be integrated into software engineering following a model of user-centred design processes. Special emphasis lies on methods for analyzing user and task requirements. In a next step, there is a need to specify ergonomic goals of a project and to discuss them with prospective users. In many cases this can be done by different ways of prototyping in order to establish an iterative development process with user participation. Both prototypes and the completed software system have to be evaluated whether they fulfil the ergonomic requirements of users and tasks.

In a basic module the relations between software engineering and usability engineering can only be outlined roughly. In order to understand the models of user-centred development the students should have a basic knowledge in software engineering, especially in the area of requirements analysis and software quality management.

Models of User-centred Development

Part 3 is mainly based on ISO 12407 [20]. The first chapter gives an overview of the process model for user-centred design. Its main activities are requirements analysis, prototyping and evaluation which are covered in-depth in the following three chapters. Other important topics are the different roles of people involved in the project and the question how users can participate in the process of software development.

Students shall

- know the benefits of integrating software ergonomics into the process of software engineering,
- be able to explain models of user-centred design processes and to illustrate them by examples,
- be able to explain which phases of system development are influenced by user-centred requirements,
- be able to explain means of user participation in the design process,
- know the necessity of analysis-design-evaluation cycles,
- be able to name supports and obstacles of user-centred process models.

Requirements Analysis

User-centred design requires analysis of all aspects of the context of use. Therefore we present methods and techniques for analyzing work flows and tasks, conditions of the environment and properties of the users and the organisation like personas, contextual enquiry, and scenario based development.

Students shall

- be able to explain how methods for analyzing the context of use support the development of usable software,
- know methods of user-centred requirements analysis,
- be able to name different means of user participation and explain their advantages and disadvantages,

Specification and Prototyping

In a user-centred design process communication between the developers and the users is crucial and needs special means like scribbles, mock-ups, story boards and others. Different ways of prototyping (vertical or horizontal, low-fidelity or high-fidelity, e.g.) are taught as means to communicate interaction design to the users as early as possible. Design decisions may be influenced by style guides or in turn lead to a special style guide of a project.

Students shall

- be able to rate methods of software specification in relation to their comprehensibility to the users,
- know methods of prototyping and of evaluation of prototypes and be able to explain their value for a user-centred design process,
- be able to turn results of a user and task analysis into a design concept of a software,
- know about differences between style guides of software companies, users and projects.

Evaluation

In all phases of the development process the current design has to be evaluated. Topics of this chapter are methods of evaluation like inspections, walkthroughs, questionnaires and usability test, for example.

Students shall

- know methods of evaluation and how they can be used within the software life cycle,
- be able to name advantages and disadvantages of different methods of evaluation,
- be able to carry out simple evaluations,
- know the differences between software tests and usability tests.

5 Conclusions

Currently it is still possible to graduate in informatics without having learned the fundamentals of HCI. One the other hand, competences in HCI are of growing importance in the workplace of software developers. The curriculum for a basic module in HCI aims at filling the gap between practical requirements and current study programmes. Its objectives and contents should be an obligatory part of informatics education. If there is no special module in HCI, they must be found within other modules like software engineering or programming of user interfaces. Nevertheless, teaching of HCI will be easier in the context of a dedicated module.

The curriculum for a basic module is a first step in teaching HCI. While this module is to be a part of any study programme in informatics there should be additional modules for students who focus on HCI and for study programmes which centre on the design of interactive software. In future we will develop additional modules for further studies in HCI dealing with subjects like computer-supported cooperative work, speech processing, or usability of machines and plants, for example.

As there are rather few study programmes in Germany in which all the necessary fundamentals of HCI are taught [21], first of all the above curriculum has to be integrated into informatics education in order to assure that future software developers are able to produce software that fulfils the legal and economic requirements of usability.

References

1 Hewett T T, Baecker R, Card S, Carey T, Gasen J, Mantei M, Perlman G, Strong G, Verplank W. ACM SIGCHI curricula for human-computer interaction. New York, ACM: 1992
Web version http://www.sigchi.org/cdg/ 2008 April

2 ISO 9241 Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability. ISO: 1998.

3 Council Directive 90/270/EEC of 29 May 1990 on the minimum safety and health requirements for work with display screen equipment. EEC Official Journal L 156, 21/06/1990 P. 0014 – 0018.

4 ISO 9241 Ergonomic requirements for office work with visual display terminals (VDTs) (older parts) / ISO 9241 Ergonomics of human-system interaction (newer parts). ISO: different years.

5 Verordnung über Sicherheit und Gesundheitsschutz bei der Arbeit an Bildschirmgeräten (Bildschirmarbeitsverordnung – BildscharbV). BGBl I 1996, 1843.

6 Section 508 of the Rehabilitation Act (29 U.S.C. 794d), as amended by the Workforce Investment Act of 1998 (P.L. 105-220), August 7, 1998. http://www.section508.gov 2008 May.

7 W3C. Web Content Accessibility Guidelines 1.0 - W3C Recommendation 5-May-1999. http://www.w3.org/TR/WCAG10/ 2008 May.

8 Verordnung zur Schaffung barrierefreier Informationstechnik nach dem Behindertengleichstellungsgesetz (Barrierefreie Informationstechnik-Verordnung – BITV), BGBl I 2002, 49.

9 ISO/TS 16071 Ergonomics of human-system interaction – Guidance on accessibility for human-computer interfaces. ISO: 2003-02.

10 ISO 9241 Ergonomics of human-system interaction – Part 171: Guidance on software accessibility. ISO/DIS 9241-171: 2006.

11 Maaß S, Ackermann D, Dzida W, Gorny P, Oberquelle H, Rödiger K-H, Rupietta W, Streitz N A. Software-Ergonomie-Ausbildung in Informatik-Studiengängen bundesdeutscher Universitäten. Informatik-Spektrum, 16 (1993) 1, 25-30

12 Krasemann H. Welche Ausbildung brauchen Informatiker? Informatik-Spektrum 20 (1997) 6, 328–334.

13 Stärkung der Anwendungsorientierung in Diplom- Studiengängen der Informatik an Universitäten. Empfehlungen der Gesellschaft für Informatik e.V. (GI) No. 42, 1999.

14 Bachelor- und Masterprogramme im Studienfach Informatik an Hochschulen. Empfehlungen der Gesellschaft für Informatik e.V. (GI) No. 48, 2005.

15 Fachausschuss Informatik. Fachspezifisch ergänzende Hinweise zur Akkreditierung von Bachelor- und Masterstudiengängen der Informatik (Stand 08. Dezember 2006). Düsseldorf, ASIIN: 2006.

16 IFIP Working Group 13.1 - Education in HCI and HCI Curriculum: http://www.hcieducation.org/ 2008 May.

17 Software-Ergonomie-Ausbildung in Informatik-Studiengängen bundesdeutscher Universitäten, Empfehlungen der Gesellschaft für Informatik e.V. (GI) No. 33, 1993.

18 Curriculum für ein Basismodul zur Mensch-Computer-Interaktion. Empfehlungen der Gesellschaft für Informatik e.V. (GI) No. 49, 2006.

19 ISO 14915 Software ergonomics for multimedia user interfaces. ISO: 2002 / 2003.

20 ISO 13407 Human-centred design processes for interactive systems. ISO: 1999.

21 Dahm M, Latzina M, Stroick, R. Software-Ergonomie in der Lehre – Praxisanforderungen und Lehrangebot. In: Hassenzahl M, Peissner, M (Hrsg.). Usability Professionals 2005 – Jahrestagung der gc-UPA 2005, 58-61.

Recursive Thinking in CS1

Tamar Vilner[1]*, Ela Zur*[2]*, Judith Gal-Ezer*[3]*,*

[1] *The Open University of Israel, 108 Ravutzki St., Raanana, Israel, tami@openu.ac.il*

[2] *The Open University of Israel, 108 Ravutzki St., Raanana, Israel, ela@openu.ac.il*

[3] *The Open University of Israel, 108 Ravutzki St., Raanana, Israel, glaezer@openu.ac.il*

All agree on the importance of teaching recursion in the early stages of an undergraduate program in computer science. Nonetheless, there is a consensus among researchers about the difficulties involved in teaching recursion to both high school and university students, even in advanced courses such as data structures. This paper describes a study we conducted in order to examine the difficulties our students encounter in a CS1 course – the introductory course to computer science. We examined whether CS1 students, in the final exam of the course use a recursive approach to solving problems which are inherently recursive. We found that the majority of the students solved problems iteratively, even though the problems were recursive in nature. Based on the results, we provide some recommendations oriented to placing more emphasis on recursive thinking in a CS1 course, using meaningful examples which are inherently recursive, and continuing to exercise recursive algorithms after the concept has been introduced in the specific chapter on recursion.

Keywords

CS1, recursion, backtracking

1. Introduction

1.1 The Role of Recursion in Computer Science

Recursion is an important technique for solving various kinds of problems in computer science (CS), for example problems on graphs and trees, artificial intelligence problems, etc. As Ginat points out, "recursion is not just a concept-construct element, it also is a problem-solving heuristic, which encapsulates backward reasoning and a reverse train of thought" [1]. In addition to its importance in problem-solving, recursion is also an excellent tool for developing skills of abstract thinking, one of the biggest challenges in CS education [2].

1.2 Difficulties in Understanding Recursion

Though recursion is an important issue, and has many uses in CS, it is considered a very difficult subject to learn and understand, especially when compared to iteration. There is a consensus among researchers concerning the difficulties involved in teaching recursion to both high school and university students, even in advanced courses such as data structures [3, 2]. Many researchers and CS educators have tried to analyze the reasons for these difficulties. Valazquez-Iturbide notes that the difficulties are caused by learning recursion through a procedural (imperative) paradigm (such as in teaching Pascal or C), and are not found when teaching it via a functional paradigm (such as in teaching Scheme) [4].

Sanders et al. believe that an understanding of the mental models of recursion that students develop will assist us in teaching them more effectively [5]. They found that although many students develop the "correct" copies mental model of recursion, there are still many that develop non-viable models [5].

Ginat and Shifroni [6] found that problems in understanding recursion occur when students try to understand it based on previous knowledge in programming (loops, conditional statements, etc.). They claim that it is difficult for students to understand that recursion is a new approach, and not only an expansion of iterative structures. They suggest focusing, not on the implementation of a recursive program, but on a perceptual explanation of the concept.

In [7] Levy and Lapidot state that every program involves three components: a *procedure* that runs a *process* which in the end produces a *product*. One of the difficulties in understanding recursion stems from the discrepancy between the level of complexity of recursive procedures (which look simple) and that of the processes that they produce (which are complex). The reason for this discrepancy is that the procedure presents a static situation, while the process is dynamic. Levi and Lapidot claim that in most courses that teach recursion, too much emphasis is placed on the process and too little on the product. In addition, other aspects of recursion get practically no attention. One of these is recursion as a technique for solving backwards problems. Cognitive theories of problem-solving relate to backwards problem solutions as advanced and efficient; solutions that often distinguish between experts in the field and novices who are incapable of utilizing them.

Some studies show that even when students understand the concept of recursion, and are able to use it in solving problems, they usually do so only (a) when they are specifically told that the topic discussed is recursion; (b) when they are asked to solve the problem using recursion; or (c) when the problem is expressed by a recursive definition. When there is no indication that the solution is recursive, few students think of it themselves and try to solve the problem using recursion [6, 8].

1.3 Problems in Teaching Recursion in CS1

Usually, the topic of recursion is taught quite late in the introductory course CS1. Often the teacher tries to base students' understanding of the concept on the Basic Computing Model. The Basic Computing Model is the mental model that the students build when they learn basic mechanisms of programming, like variables, assignment statements, conditional statements, loops, etc. It is a concrete model that helps students to understand and trace what is happening in the computer when it gets an instruction to execute the recursive function. The problem is that focusing on this model comes at the expense of emphasizing the declarative and abstract characteristics of recursion. The result found by many researchers is that emphasizing the concrete aspect of the recursion mechanism creates very limited understanding and causes confusion, while relating to the declarative abstract aspect improves understanding and the ability to use recursion [6].

The chapters introducing recursion in most textbooks used in CS1 courses include very few, and only standard, examples of recursion. Many of them give the factorial function as the first example, or something similar like computing the sum of the numbers between 1 to n, or computing 2^n, etc. While this is a very simple example and thus perhaps a good one to start with, it has a serious disadvantage when presenting the principle of recursion: such problems are easily solved by iteration. The students are already familiar with iteration and they feel that they already have the tools to cope with these problems - they don't need another technique, especially when the technique is not easy to understand. In addition, when the teacher presents the complicated runtime stack chart of recursion in contrast to the simple stack chart of iteration, the student decides that the recursive solution is inefficient. Using such examples, it is very difficult to convince students that recursion is an important technique for solving problems, and that it will be relevant to them in their advanced studies. As a result, students' motivation to learn this new topic decreases, and since motivation is very important in succeeding in learning a new concept, the study of recursion is negatively affected.

Beyond the confusion created because iteration is studied before recursion, there is the additional factor of the time spent teaching the concept of iteration as opposed to the time spent on recursion. Since iteration is studied in the early stages of the course, students gain a great deal of experience and confidence in working with this process. In contrast, because recursion is studied only in the advanced stages of the course, they lack the time to become thoroughly familiar with it. Iterative solutions to problems are more familiar to students, and it is hard for them to understand and use recursion [9]. Students often write an incorrect iterative solution instead of using recursion due to lack of familiarity and confidence [6].

In addition, many CS1 textbooks include only one chapter on recursion, and after teaching it, there is no emphasis on the topic. The message passed on to students as a result is that recursion is not an essential subject. The CS education community, however, thinks otherwise, and believes that it is important to spend much more time on teaching recursion.

Researchers claim that when teaching recursion, one needs to keep in mind two overall goals: reducing the complexity of the topic by emphasizing abstraction, and controlling the complexity by using modularity.

Abstraction: Abstraction is a cognitive means that enables us to concentrate on the essential features of a topic, and ignore details that are not relevant at a specific stage of problem-solving situations. Abstraction is essential in solving complex problems, as it enables the problem solver to think in terms of conceptual ideas, rather than details. Abstraction can be expressed in different ways. We will not review all these aspects of abstraction. Relevant literature is Hoare [10] and Abelson and Sussman [11].

Abstraction is the cornerstone of designing any good program. The essence of abstract-functional thinking is feeling comfortable with considering what the function does, rather than how it does it. This approach is necessary in order to understand recursion. The main idea in abstraction is that the way something functions does not necessarily need to be the way to work with it. Just as it is possible to use an electrical appliance without understanding how it works, we can use recursion without understanding how things happen [2].

Modularity: This idea is expressed in the ancient Roman concept of "divide and conquer." All the students need to know is that they have to divide a big problem into a number of small problems, to solve these, and to combine the solutions [2, 12].

Scholars suggest various solutions for overcoming the difficulty of understanding recursion. One of these solutions is to use visual and concrete/perceptible models when teaching recursion, like the models suggested by Haynes [13] and Wu et al. [9]. Another solution suggested by Wu et al. [14] is to use software to simulate recursion. This helps the students to understand the process which occurs in the computer during execution of the recursion; however, it does not focus on solving problems using recursive thinking, which is the goal of teaching recursion.

2. Teaching Recursion in Our CS1 Course

Our CS1 syllabus includes topics recommended in *Computing Curricula 2001*: basic logic; algorithms and problem solving; fundamental data structures; fundamental programming constructs; recursion; fundamental computing algorithms; basic computability; linked lists; binary trees, etc. At the time we conducted our research the language introduced in our course is C++, but mainly the procedural facet of the language, with very little time devoted to the object-oriented facet.

After teaching the basic elements of programming - variables and assignment statement, conditional and loops statements, functions and arrays in C++ - we introduce the concept of recursion. We present some simple examples like computing 2^n, sum of 1 to n, the factorial function, computing a^b. Then we continued to some more complicated problems like the Fibonacci sequence and computing the Greatest Common Divisor (GCD) using Euclid's algorithm. We then presented some examples that use the stack which is built during the recursion process, with the aim of solving the problem; for instance, reading a sentence from the input, and printing it in reverse, without using any data structure (such as array). Another example is converting a number from the decimal base to its binary base. The Hanoi towers problem was introduced to demonstrate how recursive thinking helps to solve a problem which is almost impossible to solve otherwise. Another example is counting how many paths there are in a grid, from point (0,0) to point (x, y), when you can move only North or East. Here again, the backward recursive method is the simplest way to solve the problem. We ended by teaching backtracking, using exercises such as escape from a maze or showing the knight's moves on a chess board to demonstrate the backtracking process.

After teaching recursion, we continue to search and sort methods. Here we present algorithms like binary search and merge-sort, and we give their recursive versions. In the final part of the course we teach linked lists and binary trees. Because binary trees are inherently recursive, all the examples given are recursive.

The learning process is accompanied by home assignments. The students are required to hand in assignments, exercises or other types of tasks during the semester according to a predefined schedule. At least one of the assignments deals with recursion.

3. The Study

To explore how our students perceive the concept of recursion, we considered checking whether students use the recursive approach when given a problem which is inherently recursive. Because students usually use recursion only when they learn the concept, and avoid using it otherwise, we decided to test this in the final exam (the first and the second sitting) of the CS1 course; that is, after studying the entire course and not just the chapter related to recursion. We adapted two interesting and suitable problems from Ginat [1] without suggesting to students that recursion is the preferable approach to use in solving the problem.

3.1 Research Population

Our study was carried out on 252 CS1 students. The students could take one of two final exam sittings. Their choice is made arbitrarily according to their own schedule. Only 124 students passed (49%) the exams. This low success rate is due to the university's open admissions policy, which means that there are no entry prerequisites.

3.2 Research Instruments

The research instruments were the two following questions related to recursion, included in the two different sittings of the final exam.

Question 1 (First sitting)

Write a program for which the input is the positive integers X and Y, X<Y, and its output is the *minimal number* of invocations of the operations *+1* and *×2* that are required to obtain Y from X.

Example. For the input 10 17, the output will be 7 (due to 7 invocations of +1). For the input 10 21, the output will be 2 (due to *×2* and then *+1*).

The function can be a recursive one or an iterative one, as you like.

Question 2 (Second sitting)

Write an algorithm for which the input is a positive integer N, indicating the number of points a team accumulated in a basketball game, and its output is the number of different ways that these points could have been accumulated. Recall that in basketball the number of points can increase by 1, 2, or 3 at a time.

Example. For the input 3, the output will be 4, since there are 4 different ways to accumulate 3: 1+1+1, 1+2, 2+1, 3.

The function can be a recursive one or an iterative one, as you like. Please note that you do not have to write the possibilities themselves but only how many there are.

4. Results

4.1 Question 1

At the first sitting of the exam, students were given Question 1. Of 121 students who chose to take the first sitting, 61 passed. The correct solution is based on backward reduction, which may be recursive or iterative. Here is the code for the recursive backward solution:

```
int minOps (int x, int y)
{
  if (2 * x > y)
        return y-x;
  else
        if (y%2 == 1)
              return (minOps (x, y-1) + 1);
        else
              return (minOps (x, y/2) + 1);
}
```

Only 6 of the 61 students who passed the exam (10%) gave this solution. The same algorithm could be approached by iteration. Here is this solution:

```
int iterOps (int x, int y)
{
  int numOfOps = 0;
  while (x < y)
  {
        if ((y%2 == 0) && (y/2 >=x))
              y = y/2;
        else
              y = y-1;
        numOfOps++;
  }
  return numOfOps;
}
```

Of the 61 students, eight (13%) gave the iterative version of the backward solution.

Many of the students who passed the exam answered this question incorrectly. Their solution was based on forward thinking, which led to an incorrect solution. Here again we found two kinds of solutions, iterative and recursive. Below is a typical iterative solution proposed by students:

```
int wrongIteration (int x, int y)
   {
      int numOfOps = 0;
      while (x < y)
      {
            if (2 * x <= y)
                  x = x*2;
            else
                  x = x+1;
            numOfOps++;
      }
      return numOfOps;
   }
```

57% of the students (35) gave this kind of iterative solution.

Surprisingly, we found that 20% of the students (12) did write a recursive solution, but used only a recursive technique without using recursive thinking. Here is a typical recursive function written by students:

```
   int wrongRecursion (int x, int y)
   {
       if (x == y)
            return 0;
       else
       {
            if (2 * x <= y)
                  return wrongRecursion (2*x, y)+ 1;
            else
                  return wrongRecursion (x+1, y)+ 1;
       }
   }
```

Figure 1 shows the distribution of answers to question 1.

From analyzing the results, we see that 70% of the students gave an iterative function, while only 30% gave a recursive solution. In the question we specifically noted that they could do either, and indeed, as we expected, the students felt more comfortable using the iterative than the recursive approach. Of the 70% who gave an iterative function, only 19% gave a correct solution, while of the 30% who gave a recursive function, one-third answered correctly. We can infer that because the problem was inherently recursive, students who thought recursively were more successful in writing a correct solution.

Although it is disappointing to see that only 23% answered the question correctly either through recursion or iteration, there may be an explanation for this. The examples in the question were not entirely representative. In both examples the correct answer can be obtained by using a greedy algorithm which tries to multiply by 2 first, and then to add 1. If we had given an example like x = 10 and y = 22, the correct answer (+1 ×2) would have been a hint that a greedy algorithm is not the right one.

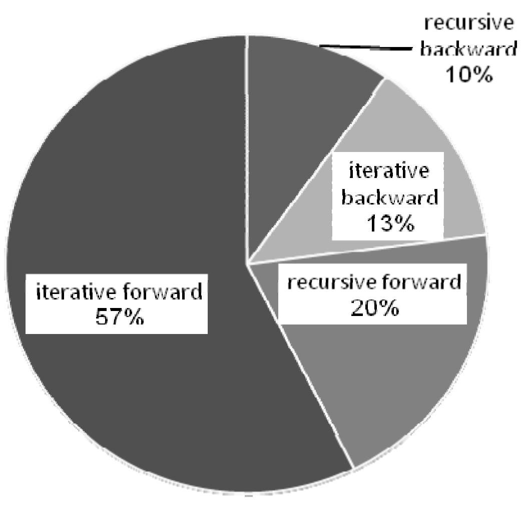


Figure 1 Distribution of answers to question 1.

4.2 Question 2

The second sitting of the exam included question 2. Of 131 students who chose to take the second sitting, 63 passed. The correct solution is based on backward reduction, which may be recursive or iterative. Here is the code for the recursive backward solution:

```
int noOfPath (int n)
{
  if (n==1)
        return 1;
  if (n==2)
        return 2;
  if (n==3)
        return 4;
  return noOfPath(n-1) + noOfPath(n-2) + noOfPath(n-3);
}
```

Most of the students, 36 out of 63 (57%) were able to solve this problem, and gave this recursive solution.

The same algorithmic thinking could be iterative (much more efficient). Here is the solution:

```
int iterPath (int n)
{
   int first = 1, second = 2, third = 4;
   int newNum = first + second + third;
   for (int i=1 ; i < n-3; i++)
   {
         first = second;
         second = third;
         third = newNum;
         newNum = first + second + third;
   }
   return newNum;
}
```

Only 3 excellent students (5%) chose this approach, which combines backward thinking and considering the efficiency of the algorithm. 24 of the students (38%) wrote various wrong solutions (iterative or recursive ones).

The answers to this question were better. 62% of the students gave a correct solution. 92% of them wrote a recursive function and 8% of them wrote an iterative one.

5. Discussion and Recommendations

Our results show that for the first question, which was inherently recursive, the majority (70%) of the students gave an iterative solution. Ginat's results showed the same phenomenon [1]. Ginat conducted his study with 23 CS college-level students who were studying towards a CS teaching certificate. His students completed several courses that involved recursion (like CS1, CS2, Introduction to Algorithms etc.), and the study was conducted some time later during a didactic course.

We thought that the results of Ginat's study showing that the majority of the students did not invoke or follow backward reasoning, was due to the fact that this didactic course took place some time after the students took the courses that introduced recursion. Surprisingly, we anticipated the same phenomenon, despite the fact that our exam took place just after the CS1 course, which introduces and practices recursive thinking.

When we re-examined the results of the second question, we observed that many of the students (62%) succeeded in solving the question. Since there were no differences between the students who took either of the two sittings of the exam, we realized that there must be some reason which led to this success. Indeed going through the examples given in the course, we saw that two examples were quite similar to the second question given in the second sitting of the exam. One was the Fibonacci sequence, and the other was the counting of the North-East paths between points (0, 0) to (x, y) on a grid. We can see that the students used algorithmic patterns. Though the best solution to this problem is to use iteration in backward thinking specifically, most of the students used a recursive algorithm similar to the ones with which they were already familiar.

To conclude, we recommend presenting different kinds of examples when teaching recursion, starting with visual examples, such as fractal figures or Russian dolls. Then giving the students some exercises on algorithmic recursive thinking, without concentrating on the language syntax of recursion. We think that giving simple examples like factorial computation, at this point, causes difficulties in convincing students that recursion is an important technique for solving problems.

A good candidate to present as a non-trivial problem is the problem of question 1 – a question which can be better solved by using a recursive algorithm. The teacher can demonstrate the solution by giving each student his/her task to remember his/her move and "sending" the next call to another student.

We believe that such problems will increase students' motivation to learn and use recursion.

After the students understand the recursive way of thinking, one can proceed to the syntax, and start writing recursive functions. Now, of course, is the time for simple examples like factorial or Fibonnacci sequence, so the students can follow the function flow.

We suggest providing some simple examples first, and then proceeding to more complicated ones. Here one can present examples like Hanoi towers. Later, the subject of backtracking has to be presented, and one can present examples like solving a maze or the 8-Queens problem. In our course, we adopted these recommendations and changed out method of teaching recursion accordingly.

In the following we show a nice and non-trivial example which uses backtracking. It is a variation of the famous "knapsack problem": Write a recursive Boolean function which gets as parameters a one-dimension array full of integers, and an integer number x. The function has to return true if there are some numbers in the array, whose sum is equal to x. For example, if the array is {5, 22, 13, 5, 7, -4}, and x = 42, the function returns *true* since 22+13+7 = 42. If x = 7, the function returns *false*.

The solution for this problem is:

```
int cover (int [] a, int i, int amount)
{
      if (amount == 0)
         return 1;
      if (i == n)
         return 0;
      return (cover (a, i+1, amount-a[i]) || cover (a, i+1, amount));
 }
```

Finally, as we all know, sometimes quantity is as important as quality. In the case of recursion, it is important to provide as many different problems as possible so that the students are convinced of the importance of recursion in solving problems even though it is not always the most efficient solution. Also, it is important to continue and practice recursive algorithms after the concept has been introduced in the chapter on recursion. The teacher can show recursion when teaching linked lists, arrays, searching and sorting algorithms, and of course, in binary trees.

References

1 Ginat, D., Do Senior CS Students Capitalize on Recursion?, *Proceedings of ITiCSE 2004*, Leeds, UK, 2004, pp. 82-86.

2 Sooriamurthi, R., Problems in Comprehending Recursion and Suggested Solutions. *SIGCSE Bulletin*, 2001, 33, 3, pp. 25-28.

3 Levy, D., Lapidot, T. & Paz, T., 'It's Just Like the Whole Picture, but Smaller': Expressions of Gradualism, Self Similarity and Other Pre-Conceptions while Classifying Recursion Phenomena. *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group*, 2001, pp. 249-262.

4 Velazquez-Iturbide, J. A., A Progressive Approach to Recursion, *Proceedings of the 29th ASEE/IEEE Frontiers in Education Conference*, San Juan, 1999, Session 12a9, pp.34-38.

5. Sanders, I., Galpin, V. & Gotschi, T., Mental Models of Recursion Revisted, *SIGCSE Bulletin*, 2006, 38, 3, pp. 138-142.

6 Ginat, D. & Shifroni, E., Teaching Recursion in a Procedural Environment - How Much Should We Emphasize the Computing Model? *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education*, New-Orleans, LA, 1999, pp.127-131.

7 Levy, D. & Lapidot T., Recursively Speaking: Analyzing Students' Discourse of Recursive Phenomena, *SIGCSE Bulletin*, 2000, 32, 1, pp. 315-319.

8 Benander, A. C., Benander, B. A. & Pu, H., Recursion vs. Iteration: An Empirical Study of Comprehension, *Journal of Systems Software*, 1996, 32, pp. 73-82.

9 Wu, C-C., Dale, N. B. & Bethel, L. J., Conceptual Models and Cognitive Learning Styles in Teaching Recursion, *SIGCSE Bulletin*, 1998, 30, 1, pp. 292-296.

10 Hoare, C. A. R., Mathematics of programming, *Byte*, 1986, pp. 115-124, 148-150.

11 Abelson, H. and Sussman, G. J., *Structure and interpretation of computer programs*, MIT Press and McGraw-Hill, 1986.

12 Turbak, F., Royden, C., Stephan, J. & Herbst, J., Teaching Recursion before Loops in CS1, *Journal of Computing in Small Colleges*, 14, 4, pp. 86-101.

13 Haynes, S. M., Explaining Recursion to the Unsophisticated, *SIGCSE Bulletin*, 1995, 27, 3, pp. 3-6.

14 Wu, C-C., Lin, M. & Hsu, I. Y-W., Closed Laboratories using SimLIST and SimRECUR, *Computers & Education*, 28, 2, 1997, pp. 55-64.

When does Algorithm Visualization Improve Algorithm Learning? – Reviewing and Refining an Evaluation Framework

Tobias Lauer

Albert-Ludwigs-Universität, D-79098 Freiburg, Germany, lauer@informatik.uni-freiburg.de

We present a review of recent evaluations of the pedagogical value of algorithm visualizations in informatics education, most of them carried out within a common research framework known as the *engagement taxonomy*. In the light of seemingly contradictory results from otherwise similar experiments, we propose refined categories for the taxonomy and modified hypotheses to be tested within the framework in future research.

Keywords
Algorithm visualization, animation, engagement taxonomy, evaluation

1. Introduction

Algorithm visualizations such as interactive animations have been popular learning aids in informatics education since at least 1980, when Ron Baecker created his famous animation film *Sorting out Sorting* [1]. Almost three decades later, it is still far from clear how effective such visualizations actually are for improving learning. While both instructors and learners are often intuitively convinced of the value of algorithm visualizations when asked in evaluations, this is of course no objective measure of the effectiveness of these learning aids. Most evaluations of algorithm visualizations unanimously report that students were “excited”, “enthusiastic”, or “motivated”, and that they “enjoyed” working with the visualizations, and often the learners express their firm belief that they have learned better because of the visualizations [1, 2]. However, as has been said in [3], while such tools may *enhance* the learning experience, they do not necessarily *improve* learning. On the contrary, even a reduction of students' performance has been observed if presentations are laden with too much (and possibly irrelevant) multimedia materials [4].

Since the late 1990s, an increasing number of empirical studies have been carried out in order to assess the pedagogical value of algorithm visualizations; an excellent overview is given in [5]. However, the outcomes of these evaluations were very mixed. While some studies reported significantly improved learning, others could not detect any difference to traditional teaching without visualizations or even indicated a negative effect. This dissatisfying result may be attributed to the fact that the settings and designs of the studies were, in most cases, not comparable and hence a great variety of influencing factors might be responsible for the discrepancies.

However, an extensive meta-review of former studies by Hundhausen *et al.* suggested that there was indeed one factor which could – at least in part – explain the differences of 21 earlier evaluations under consideration [5]. The authors found that out of the 9 experiments which focused on purely *representational* aspects of the visualizations such as sophisticated graphics and animation, only 3 produced significant results, while 10 of the remaining 12 evaluations found significant effects. These latter 12 experiments were different from the former 9 in that they all engaged the students in *activities* beyond (passive) watching. The obvious conclusion was that it is not so much important what learners *see* but what they *do* with visualizations [6].

Hundhausen's work was groundbreaking for the research on the effectiveness of algorithm visualizations in the sense that it set a new trend. In the light of the above meta-study, the focus of research has shifted from studying representational aspects of visual tools towards examining the level of engagement that learners exert when learning with visualizations.

2. The Engagement Taxonomy

In 2002, a working group at the ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE) put forward a research framework including a *taxonomy of learner engagement*, together with a number of hypotheses and testing methods for further evaluations [7]. We will describe the framework in some detail, as it is important for our work described in the following sections.

2.1 Levels of learner engagement

In a learning scenario, students exert a certain level of engagement with the visualization of an algorithm. The actual degree of engagement certainly depends a great deal on the students' attitudes towards the learning contents and methods and their willingness to engage. However, the visualization itself and the context in which it is used also allow or even enforce a certain level of engagement. A rough division into several categories can be made. The six levels of learner engagement as proposed in [7] are:

(1) NO VIEWING: This category describes the situation where learners are not provided with any visualization and is thus the default case.

(2) VIEWING: This is the most basic form of engagement with visualization which includes all the following categories. When purely viewing, learners watch an algorithm visualization more or less passively, i.e. without any interaction other than navigational controls of the execution or changing between different views. In particular, there is no interaction with the algorithm under consideration.

(3) RESPONDING: At this level, learners still cannot manipulate the visualization, but at certain points, the visualization is interrupted and learners have to answer questions or quizzes before the visualization proceeds. Examples of questions could be predictions ("What will happen next?"), assigning a segment of the code of the algorithm to the currently visualized part, assessing the correctness of the algorithm, or efficiency analysis.

(4) CHANGING: Students can modify the visualization. For instance, they have to select the input to the algorithm so they can compare the behaviour in different cases. Another example would be for learners to choose a sequence of operations carried out on a visualized data structure. Depending on the algorithm or data structure, changing can be done either offline (i.e. before the algorithm visualization starts) or online, in the course of the visualized algorithm.

(5) CONSTRUCTING: In this category students are expected to create their own visualizations of an algorithm. This is probably the most diverse category in the taxonomy, as there is a multitude of options for constructing visualizations. *Hand-constructed* visualizations are not connected to the algorithm; they can be created as a movie with any given animation editor [8] or even without computer assistance using art supplies [9]. This type has sometimes been referred to as "low fidelity" or "low tech" algorithm visualization and has been extensively studied by Hundhausen. Obvious advantages are very short production times and concentration on the workings of the algorithm rather than on implementation

details. At the other end of the spectrum, *direct construction* builds on an implementation of the algorithm. Students either map a given algorithm to a visualization or annotate the code of the algorithm with visualization commands, or they completely program the algorithm from scratch together with a visualization. Yet another construction scenario has the learners work on a given graphical representation, on which they are expected to simulate the steps of the algorithm. The graphical representation is linked to an actual implementation of the data structure and can be manipulated via the graphical user interface. This approach is suitable for assessing students' knowledge and has been used for exploratory learning [10] and automatic assessment [11], and also within an evaluation carried out by ourselves [12].

Note that in [7] the heterogeneity of this category is not further addressed when hypotheses about the effects on learning are proposed. We will discuss later why an explicit subdivision should be part of both the taxonomy and the hypotheses.

(6) PRESENTING: Learners themselves present the algorithm or data structure to an audience using visualizations – either created by themselves, or existing ones that they find helpful.

Note that although these categories are, in a certain sense, ordered by an increasing level of engagement with visualizations, the list is not to be understood as a hierarchical scale. With the exception of categories (1) and (2), no category necessarily includes or excludes any of the others. Instead, the last four categories can occur in any combination in a specific learning scenario. Also note that only engagement levels (3), (5) and (6) allow students a degree of freedom where they can actually make mistakes during their activities (and possibly get automatic feedback).

2.2 Hypotheses

The major hypothesis of the framework in [7] is that each level of engagement will result in significantly better learning than the previous ones. For instance, one sub-hypothesis claims that RESPONDING should result in significantly better learning than VIEWING.

Note that there is one exception to this: VIEWING is not hypothesized to result in better learning than NO VIEWING. In fact, one of the hypotheses given in [7] is that passively viewing an algorithm visualization will not improve learning when compared to no visualization. This claim may seem rather surprising at first, but the hypothesis – like all others – is borne out of and consistent with the results of the majority of former evaluations, most notably the ones reviewed in the meta-study by [5].

In addition, when several levels are combined, one additional hypothesis can be paraphrased as "more is better" [7]. This means that scenarios including more than one of the levels (3) to (6) in the taxonomy will result in better learning than those with only a single level of engagement.

2.3 Methodology

In addition to the taxonomy and hypotheses, the research framework provides guidelines for the practical realization of future experiments. These guidelines include examples of test scenarios, each comparing two or more engagement levels. They also contain possible test questions for assessing different *types* of learning and understanding according to the well-known taxonomy proposed by Bloom and Krathwohl, who distinguish six different levels of educational objectives: *knowledge*, *comprehension*, *application*, *analysis*, *synthesis*, and *evaluation* [13].

Moreover, other aspects to be measured and covariant factors which might influence the results are proposed. For example, a study conducted by Naps and Grissom [14] found that the experience of students with the visualization tool, as well as the fact whether the outcome of a test counts toward the final grade or course credit, can also influence the outcome of an experiment.

3. Subsequent research

The engagement taxonomy has been a great success in that it had an enormous impact on the subsequent research conducted to assess the pedagogical value of algorithm visualization. A considerable number of studies have been carried out within the framework since it was first proposed. It should be noted that some of these studies did not strictly adhere to all guidelines suggested in [7]. Nevertheless, the framework has provided a common language which makes it much easier to put different experiments in relation to each other, even though the results may not always be directly comparable. In this section, we give an overview of the picture that has emerged from recent empirical studies.

3.1 Recent experiments

Goldstein *et al.* Conducted an experiment to test the effectiveness of an interactive simulation tool in the computer networks domain [3]. Students were able to manipulate a virtual network that was visualized; hence the engagement level in the study was CHANGING. While the evaluation found significantly improved understanding of the learners after the self-study session with the visualization tool (as compared to before the session), the results of a control group who had a traditional tutor-led session on the same topic could not be used for comparison (for statistical reasons). Hence, unfortunately, this experiment shows only that an additional practical session with a simulation tool can foster learning. However, we do not know whether it is better than a traditional session, nor can we conclude that the simulation itself or the engagement with it was actually responsible for the improvement. It would be possible that reviewing the same topic from a textbook for the same amount of time could have led to the same improved understanding.

On the other hand, an experiment carried out at the same institution [15] and on a similar topic found evidence for the hypothesis that mere VIEWING of an animation will not improve learning. The study was originally designed to compare verbally narrated animations with and without an accompanying on-screen text of the narration. No differences in the learning outcome could be detected between the two conditions. Moreover, apparently no learning at all was going on; students did not do significantly better in a post-test than in the pre-test conducted before the visualization session. Naturally, while other influences on the result, such as the introductory lecture on the topic, can never be excluded, it appears that simply watching movie-style animations as accompanying materials does not result in improved learning.

Grissom *et al.* compared the effects of the levels NO VIEWING, VIEWING and RESPONDING [16]. They found no significant differences between NO VIEWING and VIEWING (as predicted by the hypothesis), but did not find any significant difference between VIEWING and RESPONDING either (contrary to the hypothesis). However, a significant improvement was measured between NO VIEWING and RESPONDING. Hence, while the overall claim that more engagement is better was supported, not all of the individual hypotheses were substantiated. The differences in the learning outcome between increasing levels of engagement may not be as discrete as the levels proposed in the taxonomy, but might be rather gradual. Moreover, as we shall point out later, there can be differences regarding the degree of engagement within the VIEWING category.

The hypothesis that a simple “movie-style” animation is no better than no visualization at all is apparently challenged in the light of a study by Ahoniemi and Lahtinen [17]. They studied the effect of visualizations when students prepared for a programming course session. In addition to printed course material, one group received simple visualizations of the new contents before a homework assignment. The authors observed a significant difference in the test grades between the two treatments when only the “novices and strugglers” of each group were considered (unfortunately, without detailing how they arrived at this separation of the learners). They conclude that the visualizations did help the weaker students but not the stronger ones. However, this result could not be replicated in a second run of the experiment carried out one week later. Another methodical problem of that study is that the two treatments used completely different tools to accomplish the assignments. While the students in the control group used paper and pen, the VIEWING group had a specialized tool which allowed them to verify and visualize their code. Hence, the effect might be caused by the students’ method of coding and feedback rather than the movie-style visualizations.

Our own experiment reported in [12] was set up to compare the levels of VIEWING, CHANGING and CONSTRUCTING, using a fairly complex data structure (Fibonacci heap) as the learning content. In this experiment, the VIEWING condition involved a very ‘active’ kind of watching the animations, allowing students to freely navigate backwards and forwards in small algorithmic steps and change the speed of the animation. On the other hand, the CONSTRUCTING condition used a rather ‘weak’ type of construction, namely the simulation of the algorithms on a given visualization by assembling operations out of smaller algorithmic building blocks (with instant feedback on the correctness). Although students in all three groups clearly acquired new knowledge, no significant differences in the learning outcome between any of the three levels of engagement were found.

The study by Urquiza-Fuentes and Velázquez-Iturbide [2] is the only other experiment known to us comparing the VIEWING and CONSTRUCTING levels and is probably the one that can best be compared to the above evaluation [12]. However, their two treatments were more different from each other than the VIEWING and CONSTRUCTING groups in our experiment: while their VIEWING group also only watched a pre-fabricated animation, the CONSTRUCTING group received the source code of the algorithm and had to create an animation from it with the authors’ visualization system WinHIPE. Hence, their type of construction was different from ours, since students also had to deal with the code of the algorithm rather than just a visual representation. The authors found a significantly better learning outcome on the application level of Bloom’s taxonomy, but also regarding the comprehension level (for one out of four questions). However, this result must be treated with caution. The difference might also be attributed to the time the students spent with the visualizations: while both groups were allowed to take as much time as they needed, the students in the CONSTRUCTING group took almost three times as long as those in the VIEWING group on average. This is not surprising, as the latter had to actually create an animation and not just watch one. However, it is unclear whether the longer time of exposure to the problem or the level of engagement was responsible for the different learning outcome. Nevertheless and despite the low number of only 15 participants, the significant results indicate that this type of construction involving the code of the algorithm, can actually lead to improved learning.

Rhodes *et al.* present a study on the effect of interactive pop-up questions built into algorithm animations [18]. In addition to testing the hypothesis claiming that RESPONDING leads to improved learning compared to just VIEWING an animation, the authors were also interested in differences regarding the type of questions and whether or not immediate feedback was provided to the students’ answers. As in our own experiment, great care was taken in order to eliminate any additional factors that might influence the result. However, one severe weakness is the relatively low number of participants (N = 29 distributed between six treatments). Interestingly, the students who had to answer pop-up questions performed worse than those who simply viewed the animation (although the difference was not statistically significant). This may be surprising, but the result is in line with that of an earlier experiment on the same aspect reported in [19]. Despite this finding, those students who received immediate feedback on their pop-up questions performed significantly better on the pop-up questions than those who got no feedback. The difference was not significant for the regular (non pop-up) questions in the post-test. Also, there was no significant difference between students who answered predictive questions ("What will happen next?") and those who had to answer questions about previous steps ("What did you just see?"). The authors assume that the overall negative impact of the pop-up questions is due to the interruption of the higher-level visualization of the whole algorithm by lower-level questions on small details.

Myller *et al.* [20] conducted a study comparing VIEWING and CHANGING (or CONSTRUCTING, cf. the discussion in section 5) in a collaborative algorithm visualization scenario. Although their results showed a tendency towards better learning in the CHANGING group, the difference was not significant. However, in an additional run of the same experiment, in which some methodical shortcomings were removed, the difference between the treatments turned out to be statistically significant [21].

3.2 Representational aspects of animations

In addition to studies relating directly to the framework and engagement taxonomy proposed in [7], there have also been recent experiments studying the effects of representational aspects of algorithm visualizations on the learning outcome. They are interesting in this context because they specifically focus on algorithm animation, rather than attempting to answer the questions for animations in general, i.e. independently of the discipline and subject domain.

Reed *et al.* report on an experiment evaluating the specific representational aspects of visual cueing and exchange motions in a Quicksort animation [22]. Visual cueing is the attempt to attract the viewer's attention to specific objects, for instance by highlighting or flashing two objects in an array in order to signal that they are being compared. Also, an exchange of two objects (such as swapping two elements in an array) can be animated in different ways. A very common method is to have the visual representations of the objects trade places, i.e. one is moving to the location of the other and vice versa. Another way would be to leave the objects at their original positions but change their shape in such a way that at the end, each one looks like the other one did before. The experiment compared CUEING vs. NO CUEING as well as MOVE vs. CHANGE SHAPE for element exchanges in the Quicksort animation.

No significant effects were discovered for questions asking for the overall comprehension of the algorithm. However, for two specific subsets of pop-up questions classified as cue-specific (e.g. "Which two elements were just compared?") and exchange-specific (e.g. "Which two elements were just exchanged?"), a significant benefit of CUEING over NO CUEING and MOVE over CHANGE SHAPE, respectively, was detected.

In summary, these variables show some influence on learning the low-level behaviour of an algorithm when asked immediately after these low-level steps occur, but not on the much more important overall comprehension of an algorithm. This finding is in line with the meta-study by Hundhausen *et al.*, which has observed that representational details seem to have no significant effects on learning [5].

3.3 Visualizations as programming aids

The focus of the framework described in [7] is on students' understanding of algorithms and data structures. This understanding is usually tested with the help of questions in a (written) post-test. What is often neglected in these tests is the learners' ability to actually code the respective algorithms. This aspect has not been evaluated in most of the above evaluations – and in fact, none of the six levels in the engagement taxonomy necessarily involves active coding.

One can argue that although the ability to code an algorithm is very strong evidence for its understanding, the opposite might not necessarily be true: for the comprehension of the abstract concept of an algorithm, it may not be required to be able to program it. However, a survey reported by Jain *et al.* [23] indicates that one major reason for the high drop-out rate of computer science students is the missing bridge between understanding fundamental principles and their implementation. Even students with appropriate expertise in a programming language apparently lack the skills to code the concepts they have understood. This is also supported by our own practical experience in advising students' projects where they create algorithm visualizations; a student's own successful implementation of an algorithm is very important and often takes as much or even more time than augmenting it with a suitable visualization.

Jain *et al.* argue that the gap between understanding a concept and the ability to implement it as an algorithm may be bridged by programming environments which include automatically produced visual representations of the algorithms and data structures that students are coding. They conducted two experiments to test whether such views help students produce and debug code more efficiently and more accurately. While the time taken by the test subjects with and without visualizations was nearly identical, there were significant differences both for the correctness of written programs and the number of errors found in debugging tasks. Students who had the visualizations performed significantly better than those who used the same programming and debugging environment without visualizations. It seems as if visualizations can be particularly helpful for those tasks.

4. Critical discussion

Given all the results we have compiled here, it would seem that the pedagogical value of algorithm visualizations is not too overwhelming, even if they are engaging and require interaction from the users. However, it also seems as if certain navigational features can enhance the benefit of visualizations. In addition, our own experiment has led us to the conclusion that other learning materials accompanying the visualizations such as introductory lectures may play an important role and can even be stronger than possible effects of the engagement level of visualizations. Rather, we tend to claim that higher engagement is effective only if it involves more than just a visual representation of the data structures and algorithms. Most experiments reporting significantly better results for the CONSTRUCTING level involved students' working with the code of the algorithm, not just constructing visual representations [2]. Similarly, more recent results such as those reported in [23] confirm that the importance of actually coding the algorithms must not be underestimated for the students' success. Indeed, this study found significant improvement when programming tasks are supported by visualizations that are part of the developing environment. More research is required to verify this hypothesis.

This issue points to another weakness – or, vagueness – in the engagement taxonomy. Coding as one important form of engagement is not explicitly included there and only *may* be part of the CONSTRUCTING level. It would certainly be beneficial to subdivide that level into forms of construction that involve programming and those that do not.

The findings of experiments conducted within the framework of the engagement taxonomy have only partially confirmed the proposed hypotheses. In general, the results are still inconsistent and often even contradict each other. This may be due to the fact that a number of the experiments did not strictly adhere to the proposed methods and procedures of the framework, often because of factors that could not be modified by the experimenters or that are not explicitly covered by or described in the research framework.

When looking at the contradicting results of seemingly similar evaluations, it is striking that almost all of these studies involve the VIEWING or the CONSTRUCTING levels. For example, both [2] and [12] compare those two categories and arrive at opposite conclusions. A close look revealed that what was regarded as "constructing" in both studies was quite different:

while in the former the students had to code the algorithms using a visual tool, the latter had them construct an animation by simulating the algorithm on a pre-defined visualization. Despite the discrepancy, the categorisation of the tasks as CONSTRUCTING is correct for both experiments, according to [7]. A similar point can be made for the VIEWING level when comparing experiments with different conclusions, for instance [12] and [18].

5. Refining the engagement taxonomy

In the light of our own results and those of other studies, we propose refinements to the engagement taxonomy, including an explicit subdivision of two of the engagement levels, VIEWING and CONSTRUCTING.

Instead of a single VIEWING category, in [24] we suggested a distinction between PASSIVE VIEWING and ACTIVE VIEWING. Recently we have been informed that Myller and colleagues also propose extensions to the engagement taxonomy which have been submitted for publication.[1] While one main goal of their modification is to adapt the taxonomy to collaborative learning, it also involves essentially the same refinement of the VIEWING level as our own, only with different category names. In order to avoid confusion, we adopt their terminology and suggest the distinction between VIEWING and CONTROLLED VIEWING.

The former describes those scenarios where viewing is unidirectional and uninterrupted. This means that learners cannot go back and there are no fixed break points after intermediate steps (even though users may be able to pause the animation manually). Note that a large number of existing algorithm animations belong to this category. In most cases, the missing rewind function is due to the tight coupling of the animations actual algorithm. Since the algorithm cannot be rewound, neither can the visualization. Conversely, the category of CONTROLLED VIEWING includes visualizations which allow users to go back to earlier stages of the algorithm and/or provide break points at which the animation stops to better highlight intermediate steps. Previous results have suggested that CONTROLLED VIEWING leads to better learning than (passive) VIEWING. This hypothesis should be tested in future experiments.

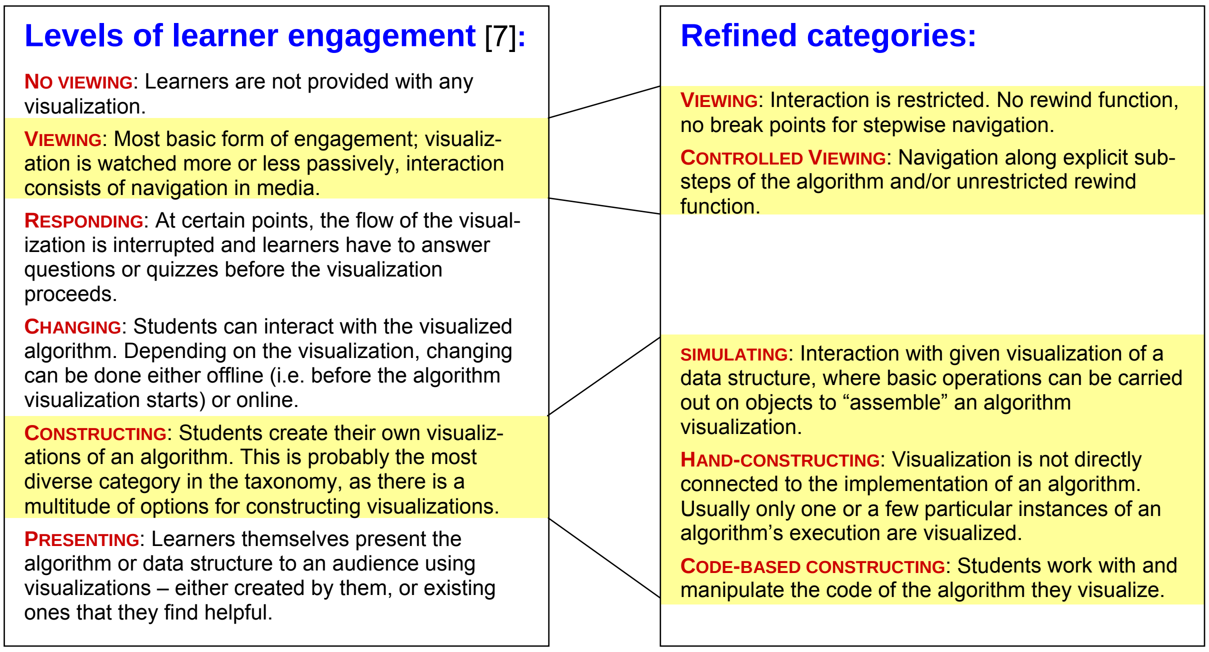
We have already pointed to the great heterogeneity among possible learning scenarios belonging to the CONSTRUCTING level and the resulting problems. We therefore propose an explicit subdivision of that category into what could be labelled SIMULATING (which we referred to as “constructive simulation” in [24]), HAND-CONSTRUCTING, and CODE-BASED CONSTRUCTING. Only the third of these categories involves working with actual code. This means that learners either program an algorithm themselves or augment the given source code in order to visualize it. Hence, the outcome of such a scenario is always a visualization directly connected with the respective algorithm. In contrast, HAND-CONSTRUCTING includes those settings where learners work, for instance, with graphical editors or use art supplies to create visualizations. Thus, Hundhausen’s low-fidelity type of visualization would belong to that category [6].

By SIMULATING, we understand settings where learners carry out individual steps of an algorithm in a predefined visualization environment, thus “assembling” a visualized algorithm out of small building blocks. Examples include the MA&DA system used for the study in [12], as well as the approaches described in [10] and [25]. This is the weakest – or most passive – form of construction, since students neither code the algorithm nor do they have to come up with their own ideas for a suitable visualization. One might even consider this engagement level as being closer to CHANGING than to CONSTRUCTING. In fact, Myller *et al.* have classified this engagement level in their TRAKLA2 simulation of binary heaps as CHANGING [20], whereas the same simulation tasks are listed as an example for CONSTRUCTING in the original definition of the taxonomy [7]. Similarly, Furcy *et al.* seem to mix up the same two categories

[1] Niko Myller, personal communication, 30th June 2008.

in their report on *Sorting out Sorting – The Sequel* (a recent state-of-the-art interactive "remake" of Baecker's film), when they classify the possibility for the user to select one out of several sorting algorithms and its input – clearly an instance of CHANGING according to [7] – as belonging to the CONSTRUCTING level [26]. (Ironically, both these papers were co-authored by members of the very same working group who came up with the engagement taxonomy.)

Apparently, a refined hierarchy, also including a separate SIMULATING category, will be helpful to resolve such and other confusions. The refined taxonomy is shown in Figure 1.



Levels of learner engagement [7]:

NO VIEWING: Learners are not provided with any visualization.

VIEWING: Most basic form of engagement; visualization is watched more or less passively, interaction consists of navigation in media.

RESPONDING: At certain points, the flow of the visualization is interrupted and learners have to answer questions or quizzes before the visualization proceeds.

CHANGING: Students can interact with the visualized algorithm. Depending on the visualization, changing can be done either offline (i.e. before the algorithm visualization starts) or online.

CONSTRUCTING: Students create their own visualizations of an algorithm. This is probably the most diverse category in the taxonomy, as there is a multitude of options for constructing visualizations.

PRESENTING: Learners themselves present the algorithm or data structure to an audience using visualizations – either created by them, or existing ones that they find helpful.

Refined categories:

VIEWING: Interaction is restricted. No rewind function, no break points for stepwise navigation.

CONTROLLED VIEWING: Navigation along explicit substeps of the algorithm and/or unrestricted rewind function.

SIMULATING: Interaction with given visualization of a data structure, where basic operations can be carried out on objects to "assemble" an algorithm visualization.

HAND-CONSTRUCTING: Visualization is not directly connected to the implementation of an algorithm. Usually only one or a few particular instances of an algorithm's execution are visualized.

CODE-BASED CONSTRUCTING: Students work with and manipulate the code of the algorithm they visualize.

Figure 1 The engagement taxonomy (left) and the proposed refinements (right).

The proposed subdivisions allow us to also refine the hypotheses associated with the taxonomy. For example, the original general hypothesis "CONSTRUCTING is better than VIEWING" [7] may have been confirmed for CODE-BASED CONSTRUCTING vs. (passive) VIEWING, but it would also predict that mere SIMULATING results in better learning than CONTROLLED VIEWING, which is rather doubtful in the light of our results. Similarly, an earlier study by Hundhausen suggested that HAND-CONSTRUCTING is probably not better than CONTROLLED VIEWING [9]. Judging from the other studies reviewed above, another hypothesis we propose for evaluation is that CODE-BASED CONSTRUCTING leads to better learning than HAND-CONSTRUCTING or SIMULATING.

6. Conclusions

We have proposed refinements to the engagement taxonomy by diving two of its six categories into further subcategories and adjusting the corresponding hypotheses accordingly. Our refinements resolve most seemingly contradicting results of different experiments conducted within the framework.

We note that these are not the only possible subdivisions within the framework. However, we are also aware that refinements to any taxonomy are useful only if the resulting categories are still general enough to group entities in a reasonable way. With our refined taxonomy, results of previous studies can still be compared; moreover, the comparison is more accurate and accounts for differences that we consider important.

Finally, we hope that future research on the effectiveness of algorithm visualization will benefit from more specific categories describing different levels of learner engagement and that a clearer picture of the influential factors in this area will emerge.

6. Acknowledgements

The author would like to thank participants of the ITiCSE 2008 conference for enlightening talks and conversations. Special thanks go to Niko Myller for information on research results which had not been published at the time of completion of this paper.

References

1 Baecker R. 'Sorting out sorting': a case study of software visualization for teaching computer science. In: Stasko J T, Domingue J, Brown M H, Price B A (eds). Software Visualization: Programming as a Multimedia Experience, Cambridge: MIT Press, 1998, 369-381.
2 Urquiza-Fuentes J, Velázquez-Iturbide J Á. An evaluation of the effortless approach to build algorithm animations with WinHIPE. Proceedings of the 4th Program Visualization Workshop, Florence, Italy, June 2006.
3 Goldstein C, Leisten S, Stark K, Tickle A. Using a network simulation tool to engage students in active learning enhances their understanding of complex data communications concepts. Proceedings of the 7th Australasian Conference on Computing Education, Newcastle, Australia, January 2005, 223-228.
4 Mayer R E, Heiser J, Lonn S. Cognitive constraints on multimedia learning: when presenting more materials results in less understanding. Journal of Educational Psychology, 93, 2001: 187-188.
5 Hundhausen C D, Douglas S A, Stasko J T. A meta-study of algorithm visualization effectiveness. Journal of Visual Languages and Computing 13 (3), 2002: 259-290.
6 Hundhausen C D. Toward effective algorithm visualization artifacts: designing for participation and negotiation in an undergraduate algorithms course. Proceedings of CHI '98, 1998, 54-55.
7 Naps T L, Rößling G, Almstrum V, Dann W, Fleischer R, Hundhausen C, Korhonen A, Malmi L, McNally M, Rodger S, Velázquez-Iturbide J A. Exploring the role of visualization and engagement in computer science education. ACM SIGCSE Bulletin 35 (2), June 2003.
8 Rößling G. The ANIMAL algorithm animation tool. Proceedings of the 5th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2000), Helsinki, Finland, 2000.
9 Hundhausen C D, Douglas S A. Using visualization to learn algorithms: Should students construct their own, or view an expert's? Proceedings of the IEEE International Symposium on Visual Languages (VL '00), September 2000.
10 Faltin N. Structure and constraints in interactive exploratory algorithm learning. In Diehl S (ed.), Software visualization. Berlin: Springer, 2001.
11 Malmi L, Korhonen A, Saikkonen R. Experiences in automatic assessment on mass courses and issues for designing virtual courses. Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2002), Aarhus, Denmark, June 2002.
12 Lauer T. Learner interaction with algorithm visualizations: viewing vs. changing vs. constructing. Proceedings of the 11th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2006), Bologna, Italy, June 2006, 202-206.
13 Bloom B S, Krathwohl D R. Taxonomy of Educational Objectives; the Classification of Educational Goals, Handbook I: Cognitive Domain. Addison-Wesley, 1956.
14 Naps T L, Grissom S. The effective use of Quicksort visualizations in the classroom, Journal of Computing Sciences in Colleges, 18 (1), October 2002: 88-96.
15 Dowling G, Tickle A, Stark K, Rowe J, Godat M. Animation of complex data communications concepts may not always yield improved learning outcomes. Proceedings of the 7th Australasian Conference on Computing Education, January 2005, Newcastle, Australia: 151-154.
16 Grissom S, McNally M, Naps T. Algorithm visualization in CS education: comparing levels of student engagement. Proceedings of the ACM Symposium on Software Visualization, San Diego, CA, USA, 2003.
17 Ahoniemi T, Lahtinen E. Visualizations in preparing for programming exercise sessions. Proceedings of the 4th Program Visualization Workshop, Florence, Italy, June 2006.

18 Rhodes P, Kraemer E, Reed B. The importance of interactive questioning techniques in the comprehension of algorithm animations. Proceedings of the ACM Symposium on Software Visualization (SOFTVIS 2006), Brighton, UK, September 2006, 183-184.

19 Jarc D J, Feldman M B, Heller R S. Assessing the benefits of interactive prediction using web-based algorithm animation courseware. Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education, Austin, Texas, USA, March 2000, 377-381.

20 Myller N, Laakso M, Korhonen A. Analyzing engagement taxonomy in collaborative algorithm visualization. Proceedings of the 12th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2007), Dundee, UK, June 2007.

21 Laakso M J, Myller N, Korhonen A. Analyzing the extended engagement taxonomy in collaborative algorithm visualization. *To appear* in Journal of Educational Technology & Society, 2008.

22 Reed B, Rhodes P, Kraemer E, Davis E T, Hailston K. The effect of comparison cueing and exchange motion on comprehension of program visualizations. Proceedings of the ACM Symposium on Software Visualization (SOFTVIS 2006), Brighton, UK, September 2006, 181-182.

23 Jain J, Cross J H II, Hendrix D, Barowski L. Experimental evaluation of animated-verifying object viewers for Java. Proceedings of the ACM Symposium on Software Visualization, Brighton, UK, September 2006, 27-36.

24 Lauer T. Reevaluating and refining the engagement taxonomy. Proceedings of the 13th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2008), Madrid, Spain, June 2008.

25 Korhonen A, Malmi L. Matrix – concept animation and algorithm simulation system. Proceedings of the Working Conference on Advanced Visual Interfaces, Trento, Italy, May 2002, 109-114.

26 Furcy D, Naps T, Wentworth J. Sorting out Sorting – the Sequel. Proceedings of the 13th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2008), Madrid, Spain, June 2008, 174-178.

Teaching Intelligent Agents using NetLogo

Ilias Sakellariou[1], Petros Kefalas[2], Ioanna Stamatopoulou[2]

[1]Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece, iliass@uom.gr

[2]Department of Computer Science, CITY College, Thessaloniki, Greece, kefalas@city.academic.gr, istamatopoulou@seerc.org

In the context of an Agent and Multi-Agent Systems course, satisfying the students demands for hands-on practice presents an interesting challenge. Educators have reported a variety of environments and techniques they use in order to increase active learning. In this paper we record our experience using NetLogo as part of the practical coursework that students need to carry out within an Intelligent Agents course. We argue that NetLogo meets most of the requirements that suit our criteria. In addition, we describe two extra NetLogo libraries provided to students, one for BDI-like agents (goal-oriented agents) and one for ACL-like (Agent Communication Language) communication. We present a few scenarios that we use in coursework handouts and how the partially developed environment we provide for each scenario facilitates practical agent design and simulation, thus satisfying the learning outcomes of the practical work and the course as a whole.

Keywords
Artificial Intelligence, Intelligent Agents, Practical Assessment, Agent Simulation Platforms

1. Introduction

Courses on Agents and Multi-Agent Systems (AMAS) have started to complement Computer Science and other related curricula during the last decade. AMAS is listed in the ACM/IEEE Computing Curricula [12] as part of Intelligent Systems and University Departments have chosen to offer a course on AMAS (under a wide variety of titles) either as a core or an optional course during undergraduate and/or postgraduate studies. Some chose to integrate AMAS principles into other courses. Due to the wide foundations and applicability of AMAS, it is expected that there is also a lot of diversity with respect to the learning outcomes and content (focus on theory or applications) as well as the context (Artificial Intelligence related or mainstream Computer Science related) in which such course is offered. This also explains the variety of valid options (teaching and assessment methods, practical work, tools, demonstrations etc.) when designing the syllabus as well as the wide variety of experiences reported in teaching.

It is therefore important to briefly define first the context to which this paper refers to. We introduced a course entitled "Intelligent Agents" (IA for short) 7 years ago in our 3 year Computer Science undergraduate curriculum. This is a final year, final semester course obligatory for all students, ranging from 25 to 50 since 2001. It covers a wide range of standard topics in AMAS (mixture of theory and practice as shown below) with no particular emphasis on any, and is assessed through coursework (practical work) and unseen final examinations. We felt that exposing the students to advanced technologies like those involved in AMAS would significantly broaden their horizons on cutting-edge information technologies.

During the first couple of years, we have intensively tried to enrich the lectures with software and video demonstrations, but the feedback from the students reported that, although they enjoyed the concept and score highly the overall teaching, they felt that the course was too theoretical with no hands-on experience. On the other hand, we knew that any attempt to assess them through some kind of IA program development would add significantly to the existing heavy load of the last semester and their effort to complete a parallel individual third year project, which is worth 4 times as much as a single course and 20 times as much a single coursework for any course in that semester. It was a challenge for us to solve the obvious problem, that is, keep the course but allow space for some practical program development. The requirements we set for driving our final decision were the following:

- have a simple environment that presents the minimum installation problems,
- provide easy visualization in order to better view of the agent behaviour and increase student interest,
- choose an easy to learn and use language thus keeping a small learning curve,
- the environment should clearly demonstrate the difficulties in AMAS programming,
- it should have support for at least reactive architectures,
- it should preferably support BDI-like and hybrid architectures,
- it should provide means for, at least limited, communication, message exchange and interaction.

Apparently, the final decision was not easy, since all these are not fully met by existing development environments. We came up with NetLogo [18] which at that time was at early stages of development but offered the minimum required to our needs. We dealt with the rest in a way that is described later on in this paper.

In section 2, we discuss in more detail the requirements and the choices available in order to integrate some practical experience in a IA course. Section 3 provides an analytic description of the IA course we offer at the Computer Science Department of CITY College. NetLogo is briefly presented in Section 4 and the extensions we suggested and implemented are listed in Section 5. In the following section 6, we present sample coursework assignments given to students. Finally, we conclude with discussion and future work.

2. Practical Work in AMAS Courses

It is widely recognised that some kind of practical program development related to AMAS courses is needed, in order for the students to understand the concepts and meet the learning outcomes of the course. It is also reported that there are several choices educators may follow, depending on the emphasis and focus they give to certain aspects of AMAS, e.g. architectures, communication and interaction protocols, applications etc. For example, various tools and environments for AMAS have been reported to assist the educational process, like RoboCup, NetLogo, TAC, FIPA-OS, JADE, JadeX, Jason, Protege etc. [2,10,19,3,5,6]. All aim to improve students' active learning in the context of AMAS, others by engaging students in writing code (e.g. Java), others by allowing development of peripheral to agents structures (e.g. Ontologies).

Obviously, educators feel happy about the value of integrating tools into their courses, but admittedly, most of the times express concerns about their complexity and the time spent by the students to reach the required level of skills in order to produce something useful to their eyes. Given the short period of a semester course, normally ranging from 10 to 15 weeks maximum, this is always going to be an issue.

On the other hand, students would like to see a more realistic outcome of their work. If this is the process of a competition like game (e.g. Trading Agent Competition) or a visualisation of the agents' environment (e.g. RoboCup), the understanding and satisfaction seems to be increasing. One could also argue that a simple robotic platform (e.g. Lego Mindstorms, RoboSapiens, i-SOFT etc.) [4,7] could also serve the purpose, since in the students perception a robot (from a science fiction perspective) matches with their perception of an agent. Although, such an approach is feasible and sometime desirable in more engineering-oriented courses, it still suffers from complexity issues due to the fact that students need to take into account non-symbolic percepts and additional hardware devices and protocols.

3. An Intelligent Agents Course

The IA course offered at CITY College is designed as a natural sequel of three other AI related courses taught in the previous three semesters, namely "Logic Programming", using Prolog, "Artificial Intelligence Techniques", introducing students to AI covering search and knowledge representation issues, and "Artificial Intelligence", which exposes students to more advanced AI issues such as genetic algorithms, fuzzy rule-based systems, planning, knowledge-based systems, machine learning etc. The aims of our IA course are to:

- introduce the student to the notions of intelligent agents and provide an introductory study of the various types of intelligent agents, their architecture, strengths and limitations;
- introduce multi-agent systems and the various issues involved in agent communication and interaction;
- discuss possible application areas of the intelligent agent technology through examples and case studies as well as demonstrate how agents can revolutionize human-computer interaction;
- present the advantages of the agent-based approach to engineering complex software systems.

By the end of the course we expect a student to be able to (learning outcomes):

- understand the basic notions of agent systems;
- explain the difference between agents and other programs;
- understand the key concepts involved with modelling agents and multi-agent systems;
- discuss and synthesise agent solutions;
- sensibly design multi-agent systems;
- cope with key issues in implementing agent-based and multi-agent systems;
- identify tasks in information systems that present possible applications areas of the technology;
- critically analyse the expected benefits of using AMAS technology.

More particularly, the topics covered include, among others, agent architectures (logic, reactive, BDI, hybrid), communication and interaction protocols (speech acts theory, agent communication languages, knowledge communication, blackboards, Contract Net protocol, auctions, negotiation), biology-inspired agents (Ant Colony Optimization, Bio-Networking, Artificial Life), planning, learning and mobile agents, agent theories (intentional notions: information, motivation and social attitudes), multi-agent system software engineering methodologies (AAII methodology, Cassiopeia, Agent UML), Semantic Web basics etc. The recommended textbook is [20], with additional readings from [17,13] and custom made slides.

The material of the course is delivered through a series of formal lectures, summing a total of 30 hours over ten of the weeks of the semester (three hours/week). The idea is that roughly one hour per week is a demonstration session during which students may, for example, watch educational videos, tool demonstrations and be exposed to applications of AMAS, while the other two hours per week is an interactive lecture. Finally, there is also a two-hour consolidation class, in the middle of the semester and another two-hour revision class at the end of it. The students' performance is assessed through two assignments, each contributing 15% of the total mark, and a two-hour formal examination at the end of the semester, for 70% of the total mark.

As in any typical IA course, it was very important for us that through the assignments during the semester we manage to assess a number of different aspects involved with the design and development of agent-based and multi-agent systems, such as agent architectures, communication protocols and languages, and interaction issues including coordination, negotiation, auctions etc. Most importantly, however, what we aimed for is that the students gain hands-on experience and that they are being assessed from a very practical perspective (even more so since assessment of the theoretical aspects is being achieved through the formal examination). With this motive, which had been further reinforced by our previous experience and students' feedback, we decided that the assignments should involve the development of multi-agent systems in an appropriate environment that would be both easy to use, not imposing an extra burden to the students, as well as maximally rewarding.

We chose NetLogo as an environment to assign assessed coursework to our students. Although it is not designed so as to target all the aspects involved in AMAS, it meets several of the requirements listed section 1. It is easy to install and provides with a plethora of interesting examples ready to run. The language is functional and although learning another programming paradigm might put some burden, our students have been acquainted with declarative programming through their previous courses and students appear to be less "intimidated" by it, in comparison to Java or Prolog for example. Therefore, the learning curve seems to be small and students are ready to produce a decent program in a short time. Additionally, it allows the visualisation of the developed systems; this is valuable to students as it helps them both to gain a better understanding as well as get immediate feedback for their efforts.

On the other hand, since NetLogo is not specifically designed for AMAS, it suffers from not being able to provide ready-made constructs for symbolic perception, goal-oriented agents, communication and coordination. In order to compensate for the features NetLogo lacks, we decided to work towards these issues in advance and give the students an appropriate small set of extensions that have been particularly developed to facilitate a BDI-like agent architecture and inter-agent communication issues.

4. NetLogo as an Educational Tool

NetLogo is a modelling environment targeted to the simulation of multi-agent systems that involve a large number of agents. The platform aims to provide "a cross-platform multi-agent programmable modelling environment" [18].

The system offers a simple and expressive programming language and facilities for GUI creation on which custom visualizations of the studied multi-agent systems can be created with particular ease. There is an extensive set of primitives, good support for floating point mathematics, random numbers and plotting capabilities. The environment is an excellent tool for rapid prototyping and initial testing of multi-agent systems, particularly suited to systems with agents situated and operating in a restricted space, as well as an excellent animation tool of the modelled system. It also proved to be an excellent educational platform for teaching IA.

The main entities of NetLogo are the patches, the turtles and the observer[1]. The observer simply controls the experiment, in which turtles and patches actively participate. *Patches* are stationary "agents", i.e. components of a grid on which *turtles* exist, i.e. agents that are able to move, "live" and interact. Both patches and turtles can inspect the environment around

[1] The recent version (4.0) also offers links, however this work does not involve these new entities.

them, for example detect the existence of other agents, view the state of their surrounding patches/turtles, and modify the environment. Probably the feature that most greatly enhances the modelling expressiveness of the platform is the fact that each patch and turtle can have its own user-defined variables: in the case of patches this allows modelling complex environments by including an adequate number of variables that describe it sufficiently and in the case of turtles it simply means that each agent can carry its own state.

The programming language allows the specification of the behaviour of each patch and turtle, and of the control of the execution. Monitoring and execution of the agents is controlled by the *observer* entity that "asks" each agent to perform a specific computational task.

As it has been argued elsewhere [16], we found NetLogo to be suitable for modelling multi-agent systems that we were dealing with in our course, since each NetLogo agent[2]:

- perceives on its environment and acts upon it,
- carries its own thread of control,
- is autonomous.

However, although NetLogo is an excellent tool that exposes the students to the difficulties of MAS development, it lacks build-in support for implementing communicating agents with intentions and beliefs.

5. Extending NetLogo with Libraries for Intelligent Agents

Being a platform that is primarily targeted to modelling social and natural phenomena, NetLogo supported fully the creation and study of reactive agent systems. Indeed, one of the first models that we studied was the Luc Steels Mars Explorer experiment [15] that presents many similarities to the ant colony foraging behaviour experiment included in the library models (ants can be modelled and studied as reactive agents [9,14]). However, the study of BDI-like agents (those that exhibit goal-oriented behaviour through Beliefs-Desires-Intensions) that are able to communicate with explicit symbolic message exchange was not supported. Thus, taken into consideration the fact that NetLogo fulfilled the majority of our requirements we decided to extend the platform by providing the students with one library for building simple BDI-like agents and one for FIPA-like message exchange, with their accompanying manuals.

5.1 BDI-like agents in NetLogo

Although, we could have adopted as an option to link through the JAVA interface of NetLogo an already existing BDI development platform, as for example JAM [11], this would have made the installation of the coursework platform a lot more difficult for students and would have increased significantly the learning curve, due to the complexity of such a fully fledged development environment. Thus, we decided to provide a simpler alternative to develop limited BDI-agents by providing the necessary primitives through a NetLogo library[3].

The simple BDI architecture that we have followed, follows a PRS-like [8] model, i.e. there is a set of intentions (goals) that are pushed into a stack; of course the implementation is far from delivering all the features of systems like JAM, but still can be used in implementing simple BDI agents in the NetLogo simulation platform.

[2] Actually, the excellent page that Jose Vidal (http://jmvidal.cse.sc.edu/) maintains for NetLogo models, was a starting point for our work.

[3] NetLogo being a simple platform, does not support libraries in the classic sense found in other programming environments; by the term library we refer to a set of procedures and functions (reporters) that the students are given and include in their own code

An intention consists of two parts: the *intention name* and a condition that we call *intention-done*. The former maps to a NetLogo procedure (possibly user defined) while the latter maps to boolean NetLogo reporter (function) (again possibly user defined). The semantics are the standard followed by other architectures: an agent must pursue an intention until the condition described in the intention-done part evaluates to true. For example the following intention of a luggage carrier agent working at an airport:

```
["move [23 20]" "at-gate 3"]
```

states that the agent is currently committed to moving towards the point `(23, 20)` and it will retain the intention until the reporter function "`at-gate 3`" evaluates to true. Note that the user has to specify both the procedure and the reporter, that map to the two parts of the intention.

The main concept behind the present implementation is the *intention stack* where all the intentions of the agent are stored. Agents follow the execution cycle shown below:

```
IF the intention stack is not empty THEN do:
  1. Get the top intention I from the stack
  2. Execute I
  3. If intention-done evaluates to true then remove I from stack
ELSE do nothing
```

The NetLogo implementation is rather straightforward: each intention is represented by a list of two elements, one for each part. The intention stack is a list stored in a specific turtle-own variable, which simply means that each agent has its own stack. The execution cycle is encoded in the procedure `execute-intentions`, that is called to invoke the agent's behaviour. The library also provides a set of reporters and procedures to the user (student) for adding and removing intentions on the stack, inspecting the current intentions, set time-outs as intention-done conditions, etc. For example, the following line:

```
add-intention "move [23 20]" "at-gate 3"
```

will add the corresponding intention to the stack of the calling agent.

To further support the BDI architecture, facilities for managing beliefs were also created. Although, the latter was not really necessary, since it is rather simple to store any information on a related turtle variable, we have designed a set of procedures and reporters that would form an abstraction layer that facilitates the students to manage agent beliefs, without getting into too many details about how to program in the NetLogo language.

A belief consists of two elements: the *type* and the *content*. The former declares the type of the belief, i.e. indicates a "class" that the belief belongs to. Examples could include any string, e.g. "`position`" "`agent`" etc. Types facilitate belief management, since they allow to check for example whether a belief of a specific type exists or the removal of multiple beliefs at once. The content on the other hand, is the specific information stored in the belief. It can be any NetLogo structure (integer, string, list, etc.). Obviously, there might be multiple beliefs of the same type with a different content, however two beliefs of the same type and content cannot be added. For instance, `["agent" 5]` and `["location" [3 7]]` are examples of beliefs that the agent can have.

Belief management is done through a set of reporters and procedures that allow the creation, removal, checking of the agent's beliefs. For example, the following line:

```
add-belief create-belief "plane-at" [23 15]
```

will include a belief of type "`plane-at`" with content "`[23 15]`" in the agent's beliefs. In the current implementation, all agent beliefs are stored in an agent own variable named `beliefs`.

5.2 FIPA-like Message Passing

The ability to exchange symbolic messages is rather important in a course that includes agent communication and interaction protocols, such as the contract net protocol. Thus, it was necessary for us to somehow enhance NetLogo with explicit message communication primitives. Messages closely follow the FIPA ACL (the Agent Communication Language of the Foundation for Intelligent Physical Agents) message format, i.e. are lists of the form:

```
[<performative> sender:<sender> receiver:<receiver>
   content: <content>..]
```

For example, the following message was send by agent (turtle) `5` to agent `3`, its content is "`plane-at 23 15`" and the message performative (FIPA) is "inform".

```
["inform" "sender:5" "receiver:3" "content:" "plane-at 23 15"]
```

A message may only include the above fields (performative, sender, receiver, content), omitting others such as the ontology field, for example, used by FIPA, thus assuming that all agents use the same ontology. The library, however, allows the creation and addition of any custom field that may be considered necessary.

As can be seen from the above, agents are uniquely characterized by an ID (number) that is in fact an integer value automatically assigned at the time of their creation by NetLogo ("`who`" NetLogo variable). We have adopted this naming scheme since it greatly facilitates the development of the message passing facilities. Message passing is asynchronous. A set of reporters and providers allow easily creating/sending/receiving/processing messages between NetLogo agents. For example the code below (assuming that the calling agent is 8):

```
let  somemsg create-message "inform"
set  somemsg add-receiver 5 somemsg
set  somemsg add-content "plane-at 23 15" somemsg
send somemsg
```

will send to agent `5` the message that follows:

```
["inform" "sender:8" "receiver:5" "content:" "plane-at 23 15"]
```

Of course, both libraries (BDI and FIPA) take advantage of functional features of the NetLogo programming language. The exact message will be send by simply issuing the following command:

```
send add-content "plane-at 23 15"
     add-receiver 5 create-message "inform"
```

It should be noted that there are also primitives that allow broadcasting a message to a "class" (breed) of NetLogo agents.

Incoming messages for each agent are stored in a variable named `incoming-queue`, which is in fact a list. This is a "user-defined" variable that each agent must have in order to be able to communicate. Sending a message to an agent simply means adding the message to its `incoming-queue` list; it does not require an explicit receive command to be invoked on the receivers side. At any time the agent has the ability to obtain the messages from its queue using the reporters and procedures provided.

Both libraries, FIPA and BDI, were fully implemented in the NetLogo language: thus, their "installation" was trivial, since students had only to include the given library code in their models. Error checking and debugging facilities were kept to a minimum to avoid having efficiency issues.

We have used both libraries for a number of years in our classes: students have found them easy to use and did manage to implement multi agent systems that involve communicating BDI agents under an interaction protocol. The interested reader may find the libraries, brief manuals and examples, at [1].

6. Practical Assignments using NetLogo

Using the programming facilities described in the previous sections, we have designed a number of practical assignments for the IA course. Each year, there are two practicals handed to the students: the first aims to allow students to have a gentle introduction to NetLogo and involves developing a reactive agent system. The second involves BDI agents that cooperate via message exchange and under the Contract Net interaction protocol. We have always tried to have close to the real world scenarios, so that the students' interest and motivation is higher. In one academic semester, both practical handouts refer to the same scenario, so that students see different aspects of applying AMAS technology to the same problem.

6.1 Practical #1: rescue units scenario

One of the scenarios that we have used, is the *rescue units* scenario, in which agents operate in a disaster area to efficiently locate and rescue victims. The rescue procedure is rather simple: a rescue unit locates the civilian in need (victim) and provides oxygen and water to the victims, so that they can be sustained in life until transportation arrives. Thus the rescue team consists of rescue-unit agents, i.e. autonomous vehicles that can move around the disaster area, locate (detect) any civilians in danger (victims) and temporarily rescue them. In the scenario there also exists a base station, installed in a central location, that provides refuelling and renewal of medical supplies (oxygen, water, etc.) services.

Rescue units have a very limited set of sensors, including a sensor for detecting civilians in danger, which operates at a very close range, an obstacle detection sensor (to avoid obstacles and other rescue units), sensors detecting low fuel levels etc. Agents also have limited abilities; they can move around in the disaster area, provide immediate attention to the victims, move towards the base, since the latter transmits a signal they can follow, etc. Agents do not have message exchange capabilities and are to follow the reactive architecture. The similarities of the above scenario with Luc Steels Mars explorer are obvious. This scenario is found in the first handout of the semester and upon completion of the practical we expect the students to:

- understand in depth the reactive agent architecture, its advantages and disadvantages,
- design a simple reactive agent to perform a task,
- build a simple prototype of a reactive agent system in NetLogo,
- evaluate the design choices made based on simulation results.

6.2 Requirements and assessment

In order to allow students to concentrate on the above, we provide all the sensors and agent abilities implemented in NetLogo, as well as an initial setup of the experiment environment; what we ask is the agent architecture, any enhancement that could help to increase the efficiency of the system and experimental results. Students are assessed according to the following criteria:

- Correctness, originality and justification of the proposed agent architectures;
- Implementation and code documentation;
- Analysis and presentation of experimental results;
- Presentation of the report (clarity, structure).

6.3 Practical #2: extended rescue units scenario

The same scenario, augmented appropriately, acts as the basis of the second practical, which is far more demanding. In this case, there are also ambulance agents that are autonomous transportation units, able to collect the rescued civilians and bring them to the base. However, an ambulance can save a victim only if it has been discovered by a rescue unit. In this extended scenario, all agents have the ability to exchange explicit symbolic messages and follow BDI or hybrid architectures. Overall, agents are far more complex but since students have already been exposed to the NetLogo platform, they are now asked to contribute a bit more code. For this second assignment students are asked to implement a cooperation protocol between rescue-units and ambulances, specify the FIPA-ACL messages needed to be exchanged under the cooperation protocol and implement everything they propose in NetLogo, using the libraries mentioned in the previous sections.

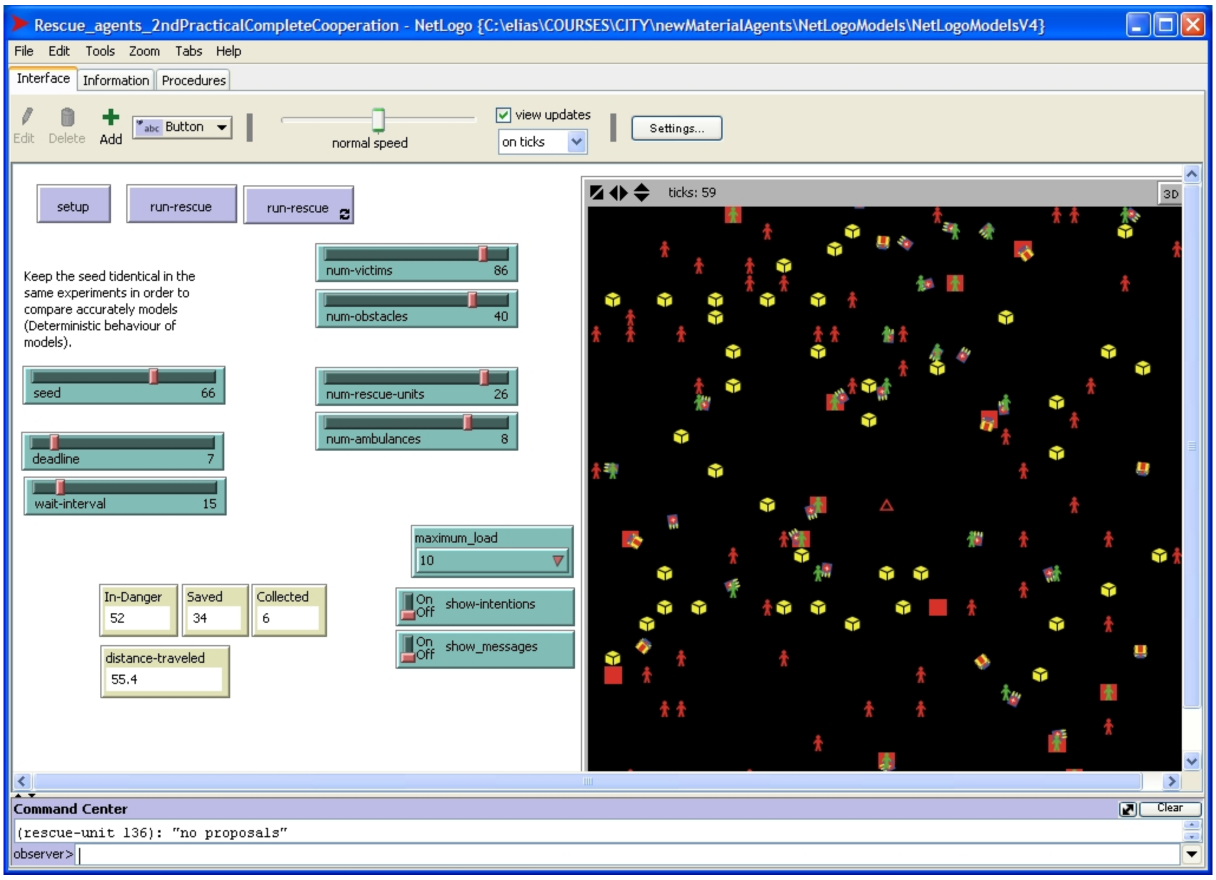
The expected learning outcomes of the second practical are that the students:

- understand in depth the issues and the difficulties involved when building a multi-agent system, such as agent communication languages, interactions protocols, language used etc.,
- use an existing library to construct FIPA ACL-like messages and implement an interaction protocol,
- propose a suitable agent architecture to perform a problem solving task,
- build a simple prototype of a multi-agent system in NetLogo,
- evaluate the design choices made based on simulation results.

Students are assessed according to the following criteria:

- Correctness, originality and justification of the proposed agent architectures;
- Correctness and justification of the cooperation protocols proposed;
- Implementation and code documentation;
- Analysis and presentation of experimental results;
- Presentation of the report (clarity, structure).

It should be noted that the code provided to the students, allows them to modify various parameters of the experiment, i.e. the number of rescue units, the number of ambulances, initial fuel, number of obstacles, as well as measure various efficiency criteria, such as total time, total distance travelled by all agents, etc. Figure 1 shows the complete environment of the rescue unit scenario.



Rescue_agents_2ndPracticalCompleteCooperation - NetLogo {C:\elias\COURSES\CITY\newMaterialAgents\NetLogoModels\NetLogoModelsV4}

Keep the seed tidentical in the same experiments in order to compare accurately models (Deterministic behaviour of models).

```
num-victims        86
num-obstacles      40
seed               66
num-rescue-units   26
num-ambulances      8
deadline            7
wait-interval      15
maximum_load       10
In-Danger          52
Saved              34
Collected           6
distance-traveled  55.4
ticks: 59
```

```
(rescue-unit 136): "no proposals"
observer>
```

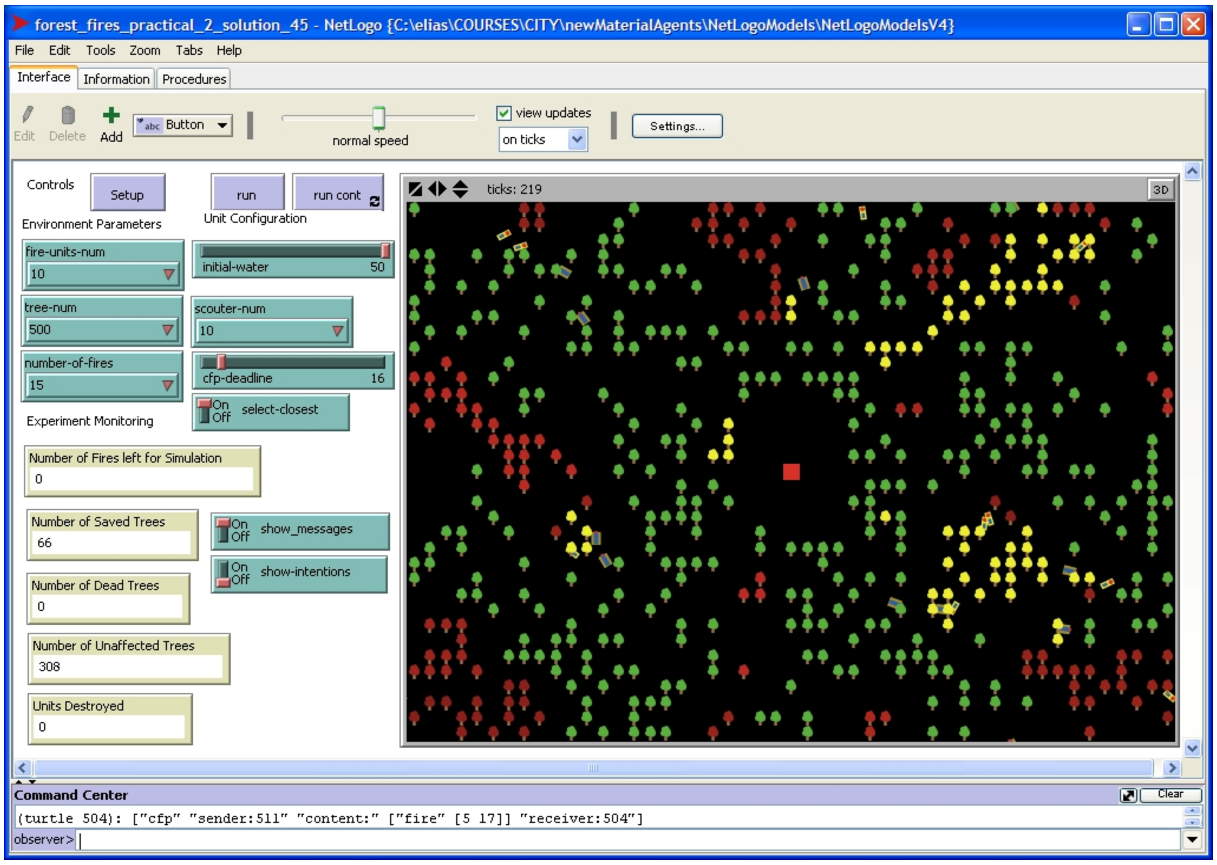
Figure 1 NetLogo ScreenShot of the Rescue Unit Scenario

6.4 Other similar scenarios

A scenario similar to the above one that has been presented to students in another academic year, involved *forest fires*, keeping the same spirit and structure as in the rescue scenario, but in a different setting: for the first practical, students had to implement reactive agents that patrol a forest and extinguish small spots of fire before they spread. For the second practical, there are scouters that detect fire spots in the forest and ground units that move to the specific location and extinguish the fire, after a fire has been detected by a scouter.

In this model, fire spots (i) appear randomly during the execution of the experiment, and (ii) are spreading in adjacent trees over time, if they are not put out, i.e. there is a close to reality fire spread model against which the agents were competing. Various parameters can be set as, for example, the number of fire spots that randomly appear, the density of the forest, the number of scouter and ground units, the water supply of each unit, etc. The learning outcomes and assessment criteria have been as stated above and the students were again given the environment setup, sensors and actions of the participating agents. Figure 2 shows a screenshot of the forest fires environment.

Naturally, the same assignment "template" can be used in a variety of other scenarios such as taxis transporting passengers in a city, vacuum cleaning a floor (which has been actually used in another academic year), logistics problems, etc. The above presented coursework handouts may also be found at [1].


forest_fires_practical_2_solution_45 - NetLogo {C:\elias\COURSES\CITY\newMaterialAgents\NetLogoModels\NetLogoModelsV4}

fire-units-num: 10
initial-water: 50
tree-num: 500
scouter-num: 10
number-of-fires: 15
cfp-deadline: 16
select-closest: On
view updates: on ticks
ticks: 219

Number of Fires left for Simulation: 0
Number of Saved Trees: 66
Number of Dead Trees: 0
Number of Unaffected Trees: 308
Units Destroyed: 0
show_messages: On
show-intentions: On

```
(turtle 504): ["cfp" "sender:511" "content:" ["fire" [5 17]] "receiver:504"]
observer>
```

Figure 2 NetLogo ScreenShot of the Forest Fires Scenario.

7. Discussion and Conclusions

The overall impression that we gain from students is that they enjoy the practical aspect of the course. This is reported in the formal evaluation of the course at the end of the semester. The overall student satisfaction increased in comparison to the early years of the introduction of the course in which the practical component was restricted only to design issues. The interest factor of the course has also increased. In addition to student formal feedback, it is also clear to us that the overall student performance in final examinations is increased due to better and deeper understanding of the issues around AMAS. The final examination contains only theoretical questions and design exercises (no programming involved) but the students present better solutions to the problems posed. Formal analysis of the data is an issue which we need to pursue in the coming years.

We have distributed a questionnaire to our final year students asking about their opinion on NetLogo and its use in the coursework assessment as well as whether the platform facilitated better understanding of the theoretical issues. The results which are listed in Table 1 demonstrate that our initial objectives about using NetLogo as a tool for teaching AMAS were met. In particular, the vast majority of students agrees that use of another programming language would not be preferable. In addition, students realise that the libraries provided for BDI and FIPA-ACL facilitated the development of a fully functional NetLogo code for the two assignments. Further collection of data is necessary in order to have a full validation of our approach, since our research involves only one academic year.

In informal discussions we had with students, they are happy with the coursework assignments. They appreciate the fact that a partial program and an initial setup of the environment is given to them in advance. Although, at first they express their concern about yet another programming language to deal with, their worries were relaxed when they had their first encounter with NetLogo. They really liked the sense of seeing and experimenting with the virtual world they developed with minimum programming effort. Thus, they had more time to design and implement the "real thing", i.e. the agent systems requested from them. The NetLogo libraries and the associated manuals we provided, facilitate their concentration on developing the systems for the given scenarios to a great extent. Finally, they appreciated the fact that the scenarios given seemed realistic enough to attract their interest.

Table 1 Students opinion on NetLogo.

	Strongly Disagree or Disagree (%)	**Neutral (%)**	**Strongly Agree or Agree (%)**
NetLogo was easy to install	0	0	**100**
NetLogo's visual environment helped to better understand agents behaviour	0	6.2	**93.8**
NetLogo demos helped to better understand the agent theory	6.2	31.3	**62.5**
NetLogo practicals increased my interest in the unit	0	18.8	**81.2**
NetLogo practicals helped me to better understand the agent theory	0	18.8	**81.2**
NetLogo language was easy to learn	0	12.5	**87.5**
Previous encounter with logic prog. helped me in learning NetLogo language	12.5	**56.2**	31.3
Practical#1: Initial code given helped me in developing the solution	0	6.2	**93.8**
Practical#2: Libraries for BDI and FIPA-ACL helped me in developing the solution	7.1	21.4	**71.4**
I would prefer to use another language (e.g. Java) for the practicals.	**62.5**	25	12.5
I would prefer to use another language (e.g. Prolog) for the practicals.	**68.7**	31.3	0
I would prefer to use actual robots for the practicals.	25	18.8	**56.2**
NetLogo practicals were fun!!!	12.5	31.3	**56.2**

We believe that the teaching objectives for this course are met through the proposed approach. All learning outcomes are assessed and in particular we have managed to accomplish the learning outcomes related to hands-on experience. It is also somehow important to us that a relatively good percentage of our graduates seek a postgraduate programme in the area of AI and AMAS.

Through the two libraries described in this paper, we managed to enable students to develop something more than a reactive agent. The libraries are by no means complete or fully fledged to meet the complete BDI and FIPA-ACL requirements. Future work is directed towards such extensions. For example, NetLogo offers a JAVA interface, through which a link to complete BDI packages, such as JAM, is possible. Although we consider it not suitable for educational purposes, as argued above, the idea could offer new opportunities for using the platform for research purposes. In addition, we are considering a rather ambitious development of an extendible/customisable game platform in NetLogo, in which educators will be able to setup game environment and rules, such as tank battles, RoboCup, etc., that will offer the possibility to apply the platform in an even simpler manner in the context of a course. Towards this direction, we have already implemented a prototype, but the work is not fully completed yet.

References

1 'Extending NetLogo to support BDI-like architecture and FIPA ACL-like message passing: Libraries' manuals and examples'.http://eos.uom.gr/~iliass/projects/NetLogo.

2 M. D. Beer and R. Hill, 'Teaching multi-agent systems in a UK new university', in Proceedings of 1st AAMAS Workshop on Teaching Multi-AgentSystems, (2004).

3 M. D. Beer and R. Hill, 'Multi-agent systems and the wider artificial intelligence computing curriculum', in Proceedings of the 1st UK Workshop on Artificial Intelligence in Education, (2005).

4 S. Behnke, J. Muller, and M. Schreiber, 'Playing Soccer with RoboSapien', in RoboCup-2005: Robot Soccer World Cup IX, volume 4020 of Lecture Notes in Artificial Intelligence, 36–48, Springer, (2006).

5 R. H. Bordini, 'A recent experience in teachingmulti-agent systems using Jason', in Proceedings of the 2nd AAMASWorkshop on Teaching Multi-Agent Systems, (2005).

6 M. Fasli and M. Michalakopoulos, 'Designing and implementing e-market games', in Proceedings of the IEEE Symposium on Computational Intelligence in Games, pp. 44–50. IEEE Press, (2005).

7 E. Ferme and L. Gaspar, 'RCX+PROLOG: A platform to use Lego Mindstorms^{TM} robots in artificial intelligence courses', In Proceedings of the 3rd UK Workshop on AI in Education, (2007).

8 M. P. Georgeff and A. L. Lansky, 'Reactive reasoning and planning', in Proceedings of the AAAI Conference on Artificial Intelligence (AAAI'87), pp. 677–682, (1987).

9 M. Gheorghe, I. Stamatopoulou,M. Holcombe, and P. Kefalas, 'Modelling dynamically organised colonies of bio-entities', in Unconventional Programming Paradigms: International Workshop, (UPP'04), Le Mont Saint Michel, France, September 15-17, 2004, Revised Selected and Invited Papers, eds., J.-P. Banatre, P. Fradet, J.-L. Giavitto, and O. Michel, volume 3566 of Lecture Notes in Computer Science, 207–224, Springer-Verlag, (2005).

10 H. Hara, K. Sugawara, and T. Kinoshita, 'Design of TAF for training agent-based framework', in Proceedings of 1st AAMAS Workshop on Teaching Multi-Agent Systems, (2004).

11 M. J. Huber, 'JAM: a BDI-theoretic mobile agent architecture', in Proceedings of the 3rd Annual Conference on Autonomous Agents, New York, NY, USA, (1999). ACM.

12 Joint ACM/IEEE Task Force on Computing Curricula, 'Computing curricula 2001', ACM Journal of Educational Resources in Computing, 1(3), (2001).

13 S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, 2002.

14 I. Stamatopoulou, I. Sakellariou, P. Kefalas, and G. Eleftherakis, 'Formal modelling for in-silico experiments with social insect colonies', in Current Trends in Informatics, eds., T. Papatheodorou, D. Christodoulakis, and N. Karanikolas, volume B of Proceedings of the 11th Panhellenic Conference in Informatics (PCI'07), May 18-20, Patras, Greece, pp. 79–89, (2007).

15 L. Steels, 'Cooperation between distributed agents through self-organisation', in Towards a New Frontier of Applications, Proceedings of the IEEE International Workshop on Intelligent Robots and Systems (IROS'90), pp. 8–14, (1990).

16 J. M. Vidal, P. Buhler, and H. Goradia, 'The past and future of multiagent systems', Iin Proceedings of 1st AAMAS Workshop on Teaching Multi-Agent Systems, (2004).

17 G.Weiss,ed. Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. MITPress, 1999.

18 U.Wilensky. Netlogo. http://ccl.northwestern.edu/netlogo. Center for Connected Learning and Computer- based Modelling. Northwestern University, Evanston, IL.,1999.

19 A.B.Williams, 'Teaching multi-agent systems using AI and software technology', in Proceedings of the 1st AAMAS Workshop on Teaching Multi-Agent Systems, (2004).

20 M.Wooldridge, An Introduction to Multi Agent Systems, John Wiley & Sons, 2002.

Author Index