A Teamwork-based Approach to Programming Fundamentals with Scheme, Smalltalk & Java

Michele Lanza
Faculty of Informatics,
University of Lugano
Switzerland
michele.lanza@unisi.ch

Amy L. Murphy
FBK-IRST
Italy
murphy@fbk.eu

Romain Robbes, Mircea Lungu, Paolo Bonzini, Marco D'Ambros, Richard Wettel
Faculty of Informatics,
University of Lugano
first.last@lu.unisi.ch

ABSTRACT

In October 2004 the University of Lugano in southern Switzerland established a new faculty of informatics. Its founding principles are innovation in teaching and faculty participation in the research community. With respect to teaching, students spend mornings attending lectures and afternoons in an *Atelier* designed to support interaction both among students and with the instructors. In teaching the first year "Programming Fundamentals" courses, we took advantage of the clean slate nature of the faculty to introduce innovative teaching elements. The novel aspects include our use of Scheme, Smalltalk, and Java, our combination of individual, pair and group projects and the integration of expert lectures to introduce useful, but slightly orthogonal elements at key points in the semester. Our very positive experience is reported along with a discussion of our observations.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer Science Education, Curriculum—*Programming*

1. INTRODUCTION

In 2004 a new faculty of informatics was founded in Lugano [5], a city located in the southern, Italian-speaking part of Switzerland. The faculty is remarkable in many ways. It features a very low professor-student ratio (1:6). English is the exclusive teaching language, targeting an international student body. A typical week is intentionally structured with all classroom lectures in the mornings and afternoons devoted to a graded *Atelier*, which is a flexible mixture of lectures on dedicated technologies (CVS, LaTeX, HTML, etc.), hands-on assignments, and, to a large extent, group project work. Moreover, we have disposed of all traditional computing labs, instead providing each student with a portable computer for use throughout the three-year bachelor studies. We took advantage of these features and the flexibility and freedom offered by a new faculty in the context of introducing programming concepts to the first year students. While remaining within the guidelines of the ACM curriculum, we introduced into our "Programming Fundamentals" courses a number of innovations described in this article. The main cornerstones are:

- *Multiple programming languages (Scheme, Smalltalk, and Java),* using each to teach specific concepts.
- *Group projects,* allowing groups of students to design and deliver self-contained systems, as early as the first semester.
- *Independent exercises plus single and pair projects,* emphasizing individual mastery of topics.
- *Design fests*, in which teaching staff works with students on the design of a system, thus enabling guided learning.
- Occasional *expert lectures*, orthogonal to the syllabus, serving to boost knowledge necessary for project completion.

The first year of the curriculum includes two semesters of programming fundamentals (PF). Figure 1 shows how the above teaching elements fit into the semester structure of each of the two courses.

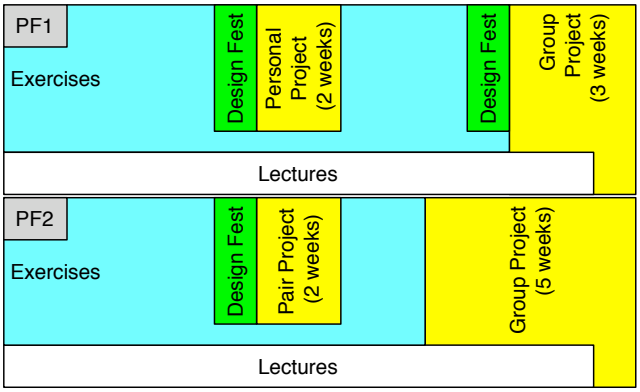

Figure 1: The semester course structure of PF1 and PF2.

The number of lectures and exercises progressively decreases each time a project starts, allowing the students to focus on its demands. The remainder of this article discusses the motivations for choosing multiple programming languages, and describes in detail the main teaching elements. We also provide a discussion of our experience.

http://doi.acm.org/10.1145/1368088.1368199
ICSE 2008 May 21-25 2008, Leipzig, Germany.

2. LANGUAGES AND TOOLS

The choice of the first programming language in a computer science curriculum is hotly debated [1, 6, 4]. We believe that the choice should be made according to the goals of the educational institution. Therefore our intent is not to argue that our mix of languages is the best for the first year, but rather to show how our choice supports our educational goals, namely to (1) get students started quickly without overloading them with too many language-specific details; (2) use a design-oriented programming approach, leaving algorithm-centered development for later courses; (3) teach students how to *use* interfaces first, and later to teach how to create them as well as their supporting implementations; (4) teach students the importance of code style, clarity, and testing. To meet the above goals, we chose the programming languages Scheme, Smalltalk, and Java.

Scheme - the functional clean slate. The majority of our students enters the program with no programming experience. Of those with some knowledge, it is typically with mainstream languages such as Java, C, or C++. Using Scheme as the first programming language both places all students at an equal start position and provides several other important benefits: Scheme has a simple syntax, it does not require a complex development environment, and it comes with easy to use multimedia libraries. All these factors have a tonic influence on the students as they see themselves solving problems shortly after the beginning of the course, both increasing motivation and productivity. We used the textbook *How to Design Programs* [3], provided by the PLT Scheme group together with a powerful and easy-to-use development environment called Dr. Scheme[1]. The book emphasizes algorithmic thinking and problem solving, presenting material at a reasonable pace. The first semester is dedicated to grasp the very basic aspects of programming: Manipulating data structures, learning what an algorithm is, problem decomposition, using functions, lists and recursion. Students also learn more advanced topics, such as: Using function libraries, higher-order functions and functional abstraction, vectors and state-based programming. The book enforces programming style by introducing an incrementally refined *design recipe*, which shows the students the importance of: (1) having a clean programming style, (2) documenting the source code, (3) defining clear interfaces, and (4) testing the written code.

Smalltalk - pure objects. Shortly before his death, Kirsten Nygaard (the inventor of Simula) stated during his keynote talk at ECOOP 2002[2] that it is wrong to lead the students from an imperative and procedural language to an object-oriented one, because the students would have trouble rearranging their concepts. Instead he proposed to "throw the babies into the water" and see what happens.

We argue that Java is not an appropriate choice in our context, as it comes with an overly complex syntax. However, since our later courses use Java as the primary programming language, we needed to find a transition to it from Scheme. We chose Smalltalk -the purest object-oriented language- for several reasons: It has a simple syntax, it is a reflective language, allowing the students to observe created objects and change them, and it features a professional development environment similar to Eclipse. Most of all, it helps us meet the following didactic goals: Teach the students the concepts of objects, classes, inheritance, polymorphism, message sending, and introduce them to frameworks. Smalltalk also features an easy way to evaluate code similar to Scheme.

Java - the mainstream. Java is currently one of the mainstream object-oriented languages. Its extensive libraries make it a powerful tool for both advanced university courses and real-life systems. To transition into Java, we start *from scratch* with the language constructs, to both give students who have fallen behind a chance to catch up, and to reinforce the importance of key concepts. Given the students' initial exposure to these ideas with Scheme and Smalltalk, they are covered quickly. Notably, this includes the complexities of object orientation, which are often difficult to teach in Java. Instead, by introducing the concepts in the context of the more straightforward Smalltalk language, we only need to cover the nuances introduced by Java, a much easier task both for teaching and for learning. As a consequence of covering the basics quickly, we have time to teach a substantial number of advanced Java topics. Finally, our focus is not on "how to program", but on "how to design (and build) object-oriented programs". This moves the motivation of the course from programming syntax to using available library interfaces, and developing solid design skills.

Our Java material is based on *Thinking in Java* [2], a good language reference book. However, it is important to note that the course always deals with design concepts, using Java as a tool. As part of this transition to Java, we initially enforce that the students *not* use an integrated development environment, as they have previously been accustomed in Scheme and Smalltalk. Instead, we use Emacs and a shell to expose the internal and external file structures of Java classes and systems as well as the edit-compile-execute steps. This basic knowledge serves the students well when we introduce Eclipse in the middle of the Java unit. Further, despite the complexity of Eclipse, students easily see how the basic components are integrated, giving explicit meaning to the sometimes obscure buttons. Test-based development (with JUnit), documentation (with JavaDoc), and code versioning (with CVS) are also integrated.

3. TEACHING ELEMENTS

Given our goals to teach programming fundamentals with an emphasis on group development skills, we exploit a variety of didactic elements throughout the first year. Some are novel, some are applied in novel ways.

Individual work. A group is only as strong as its members, therefore developing individual skills cannot be overlooked. Individual student progress is assessed throughout the semesters with graded individual exercises and written exams. A typical graded assignment lasts one week. Based on prior observations that students often delay even looking at assignments until immediately before the due date, each assignment includes a small, graded *trivia part* due 24 hours after we publish the assignment. Typically quick to complete, the goal is to force the students to demonstrate having read the assignment, thus putting its topics in their heads, even if they delay working on it. While not popular among the students, the trivia part gives us the opportunity for fast feedback if the student is going in the wrong direction, and has the desired effect to get them thinking about the assignment. At the beginning, when student programming skills are still low, and later when appropriate (e.g., following expert lectures), we augment the regular exercise series with non-graded *hands-on* exercises, completed in the lab with supervision. With the goal to introduce a concept, rather than evaluate its mastery, these assignments typically guide a student, step-wise, through a problem and its solution.

Single and pair projects. Before being thrown into large group projects, students must be confronted with problems of intermediate size. These problems must be bigger than usual exercises, but smaller than real-world applications, giving students the opportunity to practice their language knowledge and design abilities on a smaller scale before having to deal with team complexities.

[1] See http://www.drscheme.org/ and http://www.htdp.org/

[2] European Conference on Object-Oriented Programming.

Therefore, early in each semester, we introduce projects with both smaller scope and fewer people.

Personal Projects. To gain experience with Scheme, students do a personal project in the middle of the semester. Thus they practice problem identification and decomposition by themselves, and apply previously learned algorithms in different contexts. The proposed projects are mostly games, as games provide both good motivation and challenging problems. We define multiple distinct subjects, maximizing their chance to find an interesting area.

Pair Projects. To strengthen the concepts of OO design introduced early in the second semester, we assign a pair project. As the students already have experience with teamwork, a pair project is possible, allowing for a larger project than would be possible only by an individual. These projects share a common theme, namely the definition of web-based applications using Smalltalk. Again, student motivation is high, since they use web applications in their day-to-day life. To ensure diversity, we offer at least 3 different projects. They also learn the challenges of programming in pairs, which is a different mindset than both single and team programming. The skills of this project, such as defining class hierarchies, are reused in the group project. Students also are confronted for the first time with the concept of *frameworks*, as their projects use Seaside, a web framework[3].

Design fests. To expose students to a non-trivial project in a guided experience we organized three design fests (DF) during the first year. In general, a DF is an intensive one-day session in which the instructors (both teaching assistants and professors) work with the students to collaboratively design a system whose specifications are presented at the beginning of the day. Our DFs take place before the personal and group project of PF1, and before the end of semester project of PF2. By observing how the instructors guide the discussions, the students learn to apply similar techniques during later projects where the instructors are less involved.

PF1 Design Fest. The first DF takes place after 6 weeks of classes. By then, the students have learned to write simple programs consisting of a handful of functions. The DF revolves around implementing a board game similar to SameGame, which demands implementation of several non-trivial tasks: random board generation, token selection, making tokens fall, computing the score, defining the game's main loop, etc. The students face for the first time the task of dividing the program into distinct parts, recognizing the underlying data structures; it is the first hands-on experience with the *divide and conquer* principle. We split the students into sets of approximately 15 students, with each set independently working to develop a version of the game. We then divide the set into groups of 3 to 4 students, each of which works on a part of the problem for their set. Teaching assistants take the role of consultants, guiding the design discussion, and pointing out potential problems with the solution being proposed by the students. Key points in the design are the independent functions to be implemented by each group and the interfaces among these functions. Defining and developing these functions are important experiences for the students as they clearly understand the complexity of the system and the importance of cleanly defined interfaces. They also learn collaboration skills, such as relying on other teams, respecting previously defined interfaces, using stub functions, merging previously separated pieces of code, etc. As their first non-trivial design experience, they are exposed to potential problems during collaborative work, an important skill for subsequent projects.

PF2 Design Fest. The goal of the second DF is to introduce the students to the complexity of designing a non-trivial object-oriented system. The task is to design an elevator system involving several elevators, floors, sensors, etc. To drive the design process, we use CRC cards, a responsibility-driven design technique [7] in which the students define meaningful abstractions and write them on index cards. Subsequently, each student impersonates the actual objects that would be part of the system, acting out multiple scenarios. Each group of 4 students is assigned an instructor (professor or assistant) who asks questions to force the students to refine their design and think of exceptions. With the finished CRC cards the students are asked to write stub and test code, i.e., they implement the design without a working implementation, but with well-defined interfaces.

Expert lectures. In parallel with the primary lectures that build programming and design skills, we introduce a set of expert lectures to inject important concepts that do not fit with the main flow of the lectures. These introduce key concepts as needed (e.g., basic UML at the beginning of the group project). We also take advantage of having two professors co-teaching the course, with one continuing the main series of lectures, and the other taking care of the expert lectures.

Group projects. As one of our goals is to develop skills to work in a team, group projects occupy significant positions in the curriculum. Such projects have several advantages. First, they are much larger than a single-person assignment, making them closer to real-life experiences. Second, by working in a group, students are able to pick a direction in which to specialize their knowledge, e.g., combining one student's special interest in learning GUI programming and another's in networking. Third, students gain further experience in what it means to work in groups, how to address the non-programming requirements, and how to succeed with others.

In both semesters project topics are loosely defined by the instructors and refined by the students. First semester Scheme project ideas are mostly games, such as a virtual pet and a strategy game, all requiring the use of the Scheme graphics library. Java projects in the second semester are more "*serious*", featuring, for example, a learning game for school children and a personal media management system. By offering multiple project options, the students chose topics interesting to them and therefore motivating. Interestingly, one of the groups chose to include a networking component as part of their project[4], implementing a client-server strategy game in Java.

The group projects include weekly intermediate check points for design review or prototype presentation. The final deliverable includes significant written documentation, and a final, *public* presentation. It should also be noted that the duration of the projects increases from first semester (3 weeks) to second (5 weeks), reflecting both the increased maturity of the students to manage their time and the complexity of the projects.

To ensure groups make progress and stay on track, we assign a teaching assistant to follow each group. These assistants are present during all intermediate check points, and act as *consultants* to resolve complex problems. Course professors are also present during many of the check points, often taking the role of the *customer*. The complementary roles of consultant and customer gives the students the required amount of support and additional motivation to succeed and produce a solid project.

The final presentation is advertised to all faculty members as well as some VIPs (e.g., in the first year, the university president and a visiting professor from the USA attended), emphasizing to the students the importance of presentation skills in addition to programming. First semester presentations are primarily non-technical, mim-

[3] See htpp://www.seaside.st/

[4] The Networking course is taught in parallel to PF2.

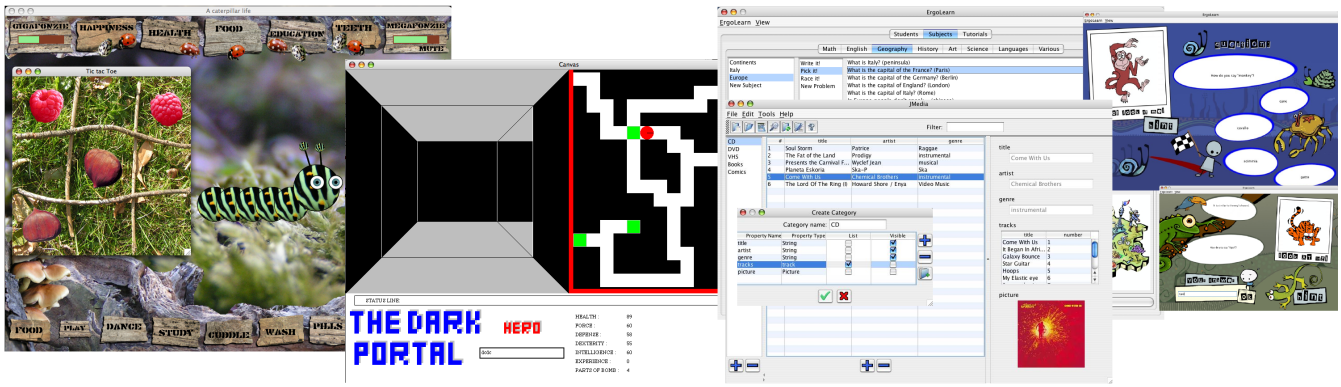

Figure 2: Screenshots from projects. Scheme (left): virtual pet, role-playing game. Java (right): media management, learning game.

icking high-level, managerial product demo presentations. In the second semester, we explicitly require presentations to include a technical component, e.g., some groups opt to present UML diagrams and code snippets. Attendees at all presentations have been very impressed by the complexity of the projects (see example screenshots in Figure 2) as well as the professionalism conveyed by first year students.

4. DISCUSSION

Next we offer our evaluation of key elements of the courses and applicability in other environments.

Multiple languages. It is worth emphasizing that our motivation is to bring students with little or no programming background to a point where they are competent system designers and developers using Java as a programming tool. Our goal is not to create expert Scheme and Smalltalk programmers, although the students do become proficient enough to produce significant projects and several did return to Smalltalk for individual project work later in their studies. Our use of multiple languages, instead, serves to introduce each concept in the best way. As such, our sequence, and the mix of exercises and projects obtained the desired goal.

End-of-semester projects. From a didactic perspective, placing a significant project at the end of both semesters clearly demonstrated the knowledge and skills acquired during the semester. From the student perspective, it served as motivation to learn, and was also a point of concrete satisfaction at the end of the course. Overall, the outcome of the projects exceeded our initial expectations.

Scalability. As designed, our course requires extensive interactions among the students and teaching staff. For example, to support 30 students, we employed two assistants throughout the semester and required additional help for the PF2 DF. During the group projects, the assistants spent significant time each week serving as consultants to the groups. Teaching larger student bodies would most likely require creating multiple sessions, as the techniques presented here, especially the group projects and DFs, require interaction that is only possible with small class sizes.

Notably, however, the use of the three-language sequence is not inherently unscalable. Instead, the fact that both PF courses are worth more credits than a typical course (8 each as opposed to 6), gave us ample time per week to cover the material, both in and out of lecture.

Integration with other courses. A significant benefit of the Lugano program is the *Atelier* in which essential topics such as CSV, LaTeX, and shell manipulation are taught in the first semester. The PF courses can therefore assume uniform knowledge among the students. With respect to the other academic courses, the students were encouraged to apply their knowledge in the projects. For example, some groups included a networking component to their group project, building on the networking course knowledge from the same semester.

5. CONCLUSION

This paper presented an innovative program to teach programming fundamentals in the first year, promoting good design and teamwork skills in addition to algorithmic skills. Our approach revolves around the flexibility of our faculty which allows us to easily introduce new teaching elements in the curriculum, and to use the most relevant tools for the task at hand. Our results are very positive, as the students enjoyed the course and demonstrated it by implementing non-trivial projects, far above the level of projects usually done by first-year students. Finally, based on observations by ourselves and our colleagues on the faculty, after completion of this PF sequence, the students are more than adequately prepared to face future courses in topics such as algorithms and data structures, net-centric computing, and software engineering.

6. REFERENCES

[1] S. A. Bloch. Scheme and java in the first year. In *Proceedings of CCSC 2000 (5th CCSC northeastern conference on the journal of computing in small colleges*, pages 157–165, , USA, 2000. Consortium for Computing Sciences in Colleges.

[2] B. Eckel. *Thinking in Java*. Prentice Hall PTR, Upper Saddle River, 1998.

[3] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs*. The MIT Press, 2001.

[4] M. Hitz and M. Hudec. Modula-2 versus c++ as a first programming language—some empirical results. In *ACM SIGCSE Bulletin*. ACM, 1995.

[5] M. Jazayeri. The education of a software engineer. In *Proceedings of ASE 2004 (20th International Conference on Automated Software Engineering*, pages 18–27. IEEE CS Press, 2004.

[6] S. Skublics and P. White. Teaching smalltalk as a first programming language. In *ACM SIGCSE Bulletin*. ACM, 1991.

[7] R. Wirfs-Brock and A. McKean. *Object Design — Roles, Responsibilities and Collaborations*. Addison-Wesley, 2003.