

## How we teach software engineering

Christine Mingins, Jan Miller, Martin Dick and Margot Postema  
Monash University, Melbourne, Australia.  
Cmingins@csse.monash.edu.au

*Column editor's note: Anyone who teaches software these days is asking the same question: How, within the constraints of an academic environment and the academic calendar, can we teach students how to deal with real-life, large-scale system development? Christine Mingins and her colleagues at Monash believe they have a solution.*

For several years we have been teaching software engineering using object-oriented principles. The success of the course comes from the way in which object oriented design, software engineering process and the case study work so well together. An essential ingredient in the recipe is Eiffel, which provides not only an attractive integrated development environment, but also a whole lifecycle method and language supporting the software engineering principles we are attempting to impart.

## The Challenge

All of the usual problems of teaching rigorous, large-scale software development in a student learning environment are compounded by the large numbers of students enrolled in this course. Our university is under the same pressures as others to 'do more with less' and to achieve even higher quality results (that is, higher pass rates, higher student satisfaction) from an increasingly diverse student population.

Our response has been to develop a course centred around a large but enjoyable software project

## The Course

The course, Software Engineering Practice, introduces software engineering concepts to second year undergraduate students in a generalist computing degree. It is presented to over four hundred on-campus students each year, and is also delivered to students enrolled in a parallel degree program in Malaysia. Students studying this subject have already completed two semesters of introductory programming in C++.

## The Design Rationale

Our aims may sound ambitious:

- **Understand the importance of software engineering:** we want to make students understand the need to establish and follow a pre-defined process when developing software products. The ability to analyze, reflect upon and improve one's software development practice is a critical requirement of any professional software developer.
- **OO design as a software engineering method:** we aim to introduce OO design as an approach which supports the engineering and maintenance of high quality software.
- **Apply the concepts:** we want to introduce these concepts and enact them in the context of a reasonably large project, involving teamwork. Some introductory Software Engineering courses present the principles in isolation from practice. Other approaches have used small projects to avoid logistics problems, but the students miss the bigger picture. Yet another technique relies on an industrial experience project, which allows students to work on a real-world system; however students need to be taught the appropriate software engineering principles before being allowed to embark on such a project. The methodologies, languages, tools, etc. used in this type of project development are often specified by the client, not by the academic supervisor or the students themselves.
- **Motivation:** finally, the course should be fun! To encourage the students to continue their studies in this area the material needs to be interesting and stimulating and maintain a balance between the whole cohort of students implementing the same specification and at the same time each team having the freedom to develop new and interesting extensions to its project.
- **Experience a large-scale software engineering project:** we did not confront the students with a realistic, large-scale, on-going software engineering project. As software systems grow, problems grow because of the complexity and sheer scale of the software. A large software project becomes more than just a small software project scaled up with greater complexities in relation to project management, resources, co-ordination, communication, tools, techniques and quality management. [1] Although our project is medium-sized, we believe that we have introduced and dealt with many of these issues (see below).

- **Use an industrial strength CASE Environment:** we are limited in choice of development environments for the following reasons. The large amount of out-of-class project work required exceeds the capacity of our laboratories and therefore we try as far as possible to use software systems that can be licensed to run on home machines at reasonable cost, especially in large subjects like this. The cost of purchasing 400 licenses for a commercial whole-of-life-cycle package is prohibitive. We are wary of being trapped into training for a particular product, rather than presenting appropriate abstractions, developing and applying principles and processes without getting lost in the complexities of a particular product. On the other hand, many commercial CASE products *hide* complexity which we wish to expose to students. We want our students to understand the importance of designing robust software architectures in engineering software which is responsive to changing requirements, and many CASE products hide inter-modular dependencies, managing them ‘under the cover’.

## The Development Environment

We use the Eiffel [2] OO development method and programming language, and the ISE Eiffel 4 EiffelBench development environment. Over a span of 6 semesters, with more than 1000 students having undertaken the course, we are extremely happy with Eiffel.

The Eiffel notation is simple and elegant, each syntactic element relating directly and clearly to the underlying semantics. In fact our main difficulty in introducing Eiffel was to rescue students from the misconception, after two semesters of C++, that all programming languages were complex and difficult to learn, requiring metre thick tomes to aid interpretation and program development! Being simple and easy to read, it is easy to learn. We provide only three hours of formal lectures, supported by six hours of laboratory work, to describe the basic syntax and structure of Eiffel systems. Many of the complexities of C++, such as memory management and casting are not part of the Eiffel lexicon

Many essential software engineering principles are inherent in the language and others are enforceable as part of the writing style. For example, class and feature header comments and assertions form essential, *living* documentation. This combination of assertions and comments forms a semantic-rich class public interface which guides students in understanding how to use classes, thereby providing invaluable assistance in the task of understanding large amounts of pre-existing code.

In fact we do not believe that students would be able to familiarize themselves with such a large body of pre-existing code (approximately 20 pages of source text) in any other language new to their acquaintance, in such a short time

The whole notion of design by contract which follows as a consequence of having executable specifications, is adopted very easily by students. Contracts are a good foundation for designing test plans. Executable assertions are especially useful in quickly pinpointing not only the locale of run-time errors, but also the actual lines of code responsible. The subject of formal specification can be introduced gently by way of the Eiffel assertion mechanism, thus providing an excellent lead-in to later formal specification and formal methods subjects.

Abstract classes which form high-level specifications are particularly useful. Students inspect the specification for a non-implementable, abstract class like *reproducible* in order to determine which behaviours to implement in the specialized classes below. Here again, assertions provide the semantic richness which fleshes out the specification.

The ISE Eiffel4 EiffelBench development environment, although idiosyncratic by Microsoft Windows standards, is affordable, and the same source will compile and run on a full range of Unix and Windows platforms.

## The Project - SimOcean

The major part of the practical component is centred on a programming project. We are indebted to Richard Wiener for the initial design, *SimOcean*, which he developed both in C++ and Eiffel [5]. The project is initially presented to students as a complete, running, although rudimentary system. Figure 1 shows the overall structure of the system as produced by ISE’s EiffelCase tool.

SimOcean is extremely successful as a student project for a number of reasons. As shown in the figure, the system is structured so that the classes divide naturally into subsystems: the *application* cluster, comprising the *simulation*, *ocean* and *location* classes; the *marine* cluster which defines the hierarchy of ocean creatures performing in the simulation; the *timing framework*, which manages the activation of the ocean creatures; and a small *library* cluster of utility classes such as *random* and *based\_counter*. The Ocean is essentially a two-dimensional array, wrapped, containing locations which in turn may contain a marine object. Depending on their type, marine creatures can possess the ability to move, reproduce, predate on other creatures, starve and die.

The *marine* hierarchy provides a rich canvas for exploring, understanding and enacting concepts such as abstract specifications (*reproducible*, *predator*) for class specialization (adding new marine creatures), and for discovering commonality, generalizing abstractions up the inheritance hierarchy.

The *timing* cluster is a true framework, and is an excellent pedagogical tool for reinforcing many of the software engineering lessons associated with design by contract. Students used to scanning the complete code of a class in order to understand how to use it are initially annoyed that they are expected to use such seemingly complex abstractions as *linked\_priority\_queue* without a word of explanation. It takes some time to sink in that the entire timing subsystem is encapsulated from the user's perspective, with only one routine in class *timeable* needing to be 'effected' in order to make use of the whole framework! We have actually found it more effective to introduce the topic of framework reuse *after* the students have been through this process.

Here too Eiffel proved its value: students find it easy to maintain a "system" view of the software. It is very easy to move to and from a graphical view of the static software architecture and the textual notation, the classes encoded in Eiffel. When the system must be modified, we find that students discover commonality in classes and abstract code upwards in the inheritance hierarchy quite naturally, not hindered by problems such as those associated with virtual classes in C++. The fact that they do so, so readily without prompting, is further evidence of the structural view facilitated by the Eiffel notation.

## Reuse

Reuse is central to the course. Students must understand and use the Eiffel class libraries, and understand the project classes provided for them to build upon.

At the project level, what does one do with SimOcean in the following semesters to generate new work, and not just resubmission of old material? The answer lies in reusing the abstractions! In succeeding semesters we have re-deployed, unchanged, all clusters of the SimOcean except the Marine cluster, to create *CitySim*, followed by *SimSpace*, *MUDSim*, *SimJungle* and currently *SimAnt*. Instead of plankton and sharks we have had cars and trains, spaceships and planets, rooms and objects, Soldier and Queen Ants. The application domain is different, but the problem remains the same, and the system must be understood in order to adapt existing code. The clusters comprising SimOcean can be viewed as a framework with only the marine object hierarchy needing re-specification and implementation.

This in itself is testament to the power of OO approaches to engineering systems which are adaptable to changing requirements.

## Content selection and ordering

A general issue of teaching software engineering is to decide what topics are best talked about first and then implemented by the students and what topics are best implemented first and then discussed in lectures. Here is how we go about it.

For some topics we teach the ideas first: Eiffel language syntax; quality management; personal software process levels; data structure selection. For other topics, however, we let the students follow and experimental approach:

We give students sample code to type in and run before explaining the code.

Students first become familiar with frameworks in their project work, then frameworks are discussed in lectures.

Students implement thorough test plans from requirements before some of the theories of testing and test strategies are presented in lectures.

Students begin to get errors in their code just before we give them guidance on some of the common errors each stage of the project is an enhancement, that is maintenance, to an existing system. Students are completing the third stage of their project before we discuss the theoretical aspects of evolving systems and maintenance.

Students count lines of code to estimate their productivity and ask questions such as 'which lines of code do I count?' before we give them a LOC counting standard so that they understand the relevance of the standard and the reasons for consistency.

Reuse, and its effect on LOC as a notion of productivity is discussed towards the end of the course, when students have been involved in a substantial amount of reuse of both library classes and their own code. students start to count defects in their code and time each phase of their project before we discuss metrics and show them how these figures can be used for estimation and to work out productivity rates and defect rates

Other topics such as class libraries, assertions and inheritance use a mix of the two approaches. The course reinforces the advantages of object oriented development. For example, many students discover for themselves the reuse advantages of abstracting functionality up the inheritance hierarchy.

## Conclusion

We believe we have achieved a balance between the conflicting requirements of dealing with the complexities of 'real-world' software engineering based on large projects with the need to establish a firm personal software process which students can use as a basis for reflection and further professional development. The course also highlights the benefits of OO development in a proper software engineering context. The selected project encourages students and staff to have fun, and just as importantly motivates the students to continue in this software engineering stream at the university.

That we are able to cover so much introductory software engineering and OO design material, working with a medium sized project, in a language new to the students in one semester, is thanks to the fitness of the tools for the task. By using Eiffel we are giving students a glimpse of the future – a seamless, whole of lifecycle approach to Software Engineering which encompasses a development environment, design method, and programming language, employing a consistent set of concepts realized with a single notation.

## References

- [1] W.S. Humphreys, *A Discipline of Software Engineering*, Addison Wesley, Reading, 1995.
- [2] B. Meyer, *Object Oriented Software Construction*, (2nd edition). Prentice Hall, Upper Saddle River, 1997.
- [3] D. Hagan and J.Lowder, "Use of the World Wide Web in Introductory Computer Programming", *Proceedings of ASCILITE'96 conference*, Adelaide 1996.
- [4] R.S. Wiener, *Introduction to Computer Science with Eiffel*, Prentice Hall, 1996

