

Including Practical Software Evolution in Software Engineering Education

Margot Postema, Jan Miller and Martin Dick
School of Computer Science and Software Engineering
Monash University
{margot.postema,jan.miller,martin.dick}@csse.monash.edu.au

Abstract

Software engineering typically requires more effort in maintenance than in development time. As such, software engineering education needs to actively include software evolution. Teaching software evolution to undergraduate students usually includes the theoretical aspects, but doesn't normally include the actual implementation. This paper describes the practice of teaching software evolution to undergraduate computing students at Monash University. It demonstrates how the four maintenance activities - corrective, adaptive, perfective and preventative - can be included into the practical component of a software engineering course, providing students with a much more realistic view of software engineering.

1. Introduction

Maintenance can be defined as the single most expensive activity in software engineering, requiring 65% to 75% of total effort [11]. Hence software engineering education needs to actively include software evolution.

The subject Software Engineering Practice (CSE2201) taught in the School of Computer Science and Software Engineering at Monash University is a second year core subject in an undergraduate degree program and comprises about 250 students per year. CSE2201 introduces software engineering concepts to students and expects students to view software engineering as an *evolutionary* process. Students are additionally introduced to and expected to implement the practical aspects of the Personal Software Process (PSP) [4]. To assist them in the PSP, students are exposed to a tool that greatly reduces the manual recording tasks. This tool – Personal Assistant for Software Engineers (PASE) was developed at Monash University [2,9].

This paper describes the practice of teaching software evolution to undergraduate computing students in the Bachelor of Computing degree, and shows how the four maintenance activities - *corrective, adaptive, perfective and preventative* - can be included into the practical component of a software engineering course, providing students with a much more realistic view of software engineering. The *adaptive* maintenance activity was introduced into the course this year.

2. Software evolution

Software systems typically need to be changed in their lifetime. Original requirements may change to reflect changing business, user and customer needs. The system's environment may change as new hardware is introduced. Errors, undiscovered during system validation may emerge, requiring repair.

The changes to a software system may be

- simple changes to correct coding errors,
- extensive changes to correct design errors, or
- significant changes to correct specification errors or incorporate new requirements [11].

As mentioned earlier, software maintenance is really software *evolution*, and is the process of changing a system to maintain its viability. [11]

The definitions for the different types of maintenance vary amongst practitioners in the field, however we use those defined by Pressman [10]. These are:

- *Corrective* – concerned with fixing reported errors.
- *Adaptive* – where the software is changed to a new environment.
- *Perfective (enhancement)* – involving new functionality to reflect changing needs.
- *Preventative* – also known as reengineering, changes the program to be more easily corrected, adapted or enhance.

Costs of these operations are additionally higher than that of developing the original software. There are several contributing factors for this, including the fact that maintenance staff are inexperienced or unfamiliar with the domain. This is typical of the recent software engineering graduates, usually employed as junior staff and assigned the maintenance tasks.

Thus, in CSE2201 students are introduced to software engineering practice via the mechanisms of software evolution. The tasks required of the students include *corrective*, *adaptive*, *perfective* and *preventative* maintenance and descriptions of the tasks performed are detailed in Section 3.

2.1. Dynamics of program evolution

The study of system change is known as program evolution dynamics. Lehman's Laws [11] have been proposed based on software engineering observations. These laws or hypotheses can be described as:

- Continuing change – programs used in the real-world environment must change
- Increasing complexity – the structure of evolving programs become more complex
- Large program evolution – program evolution is a self-regulating process. i.e. systems have their own dynamics which determines the trends of the system maintenance process and limits the number of system changes
- Organisational stability – rate of development of a program is approximately constant
- Conservation of familiarity – the incremental change of each release of a program is approximately constant.

As mentioned earlier maintenance costs may be higher than the original development cost of a program. Thus the activity of learning the software evolution process is deemed important.

2.2. Applications of the practice

A search of the *practice* of teaching software evolution has revealed little evidence of this occurring at tertiary institutes. Whilst the theory of software maintenance and software evolution is typically included in software engineering courses at the tertiary level, the practical implementation doesn't appear to be applied. Most courses teach software engineering practice with students involved in developing new applications.

One course that we discovered in our research included software evolutionary design [3]. Another course included teaching students how to read and apply technical documentation for computer networking [8]. However, it seems the inclusion of practical component in evolutionary software is limited. The practical approach adopted in CSE2201 is totally based on evolutionary software engineering.

3. The practice

CSE2201 focuses on teaching software engineering concepts to students. It aims to provide a foundation of good practice, forming the basis for a personal software improvement process in later studies and professional life; introduces object oriented design in a context firmly embedded in software engineering; and exposes students to team work on a reasonably large project [6].

The major part of the practical component is the programming project. It is based on Richard Wiener's [12] initial design, *SimOcean*, which he developed in both C++ and Eiffel. The project presented in CSE2201 is in Eiffel, and is initially presented to students as a complete running, although rudimentary system. It comprises several clusters such as the *application* cluster, the *marine* cluster, the *timing framework*, and the *library* cluster. Further details of the *SimOcean* project are described in Mingins *et.al.* [7].

As reuse is central to the course, in succeeding semesters we deployed, unchanged, all clusters of *SimOcean* except the marine cluster. The domain has successfully been changed each semester to include projects such as *CitySim*, *SimSpace*, *VirtualWorld*, *SimJungle*, *SimAntz*, *SimLan* and *SimToys*.

Figure 1 shows the overall structure of the *SimToys* system, used in semester 1, 2000. The *application* cluster comprises the *simulation*, *toyland* and *location* classes; the *components* cluster, which defines the hierarchy of components performing in the simulation; the *timing framework*, which manages the activation of the components of the toyland; and a small *library* cluster of utility classes such as *random* and *based-counter*. *SimToys* is essentially a two-dimensional wrapped array, containing locations that in turn contain up to four components. Depending on their type, components can process the ability to move, reproduce, predate on other components, starve and die. The *player* component is in effect the game actor within the simulation, and additionally is able to control the actions they take via simple command inputs.

The *components* hierarchy provides a rich canvas for exploring and understanding concepts such as specification (*moveable*, *stationary*) for class specialization (adding new components), and for discovering commonality, generalizing abstractions up in the inheritance hierarchy. The *timing* cluster is a true framework. Students do not need to understand the code associated with complex abstractions, such as *linked_priority_queue*, but simply access the timing

mechanism via one call to class *timeable*, to make use of the whole framework. The topic of framework reuse is introduced to the students after they have been through this process.

3.1. The evolutionary application

3.1.1. Stage 0. Students are provided with a reasonably large working program (see model presented in Figure 1), defined as stage 0. The code consists of approximately 2000 source lines of code. Additionally, the system specifications and a test plan are provided. As no program can be guaranteed to be error free, deliberate errors are introduced to the stage 0 code to encourage regression testing. Students are then expected to find these deliberate errors, and often, through thorough testing find others.

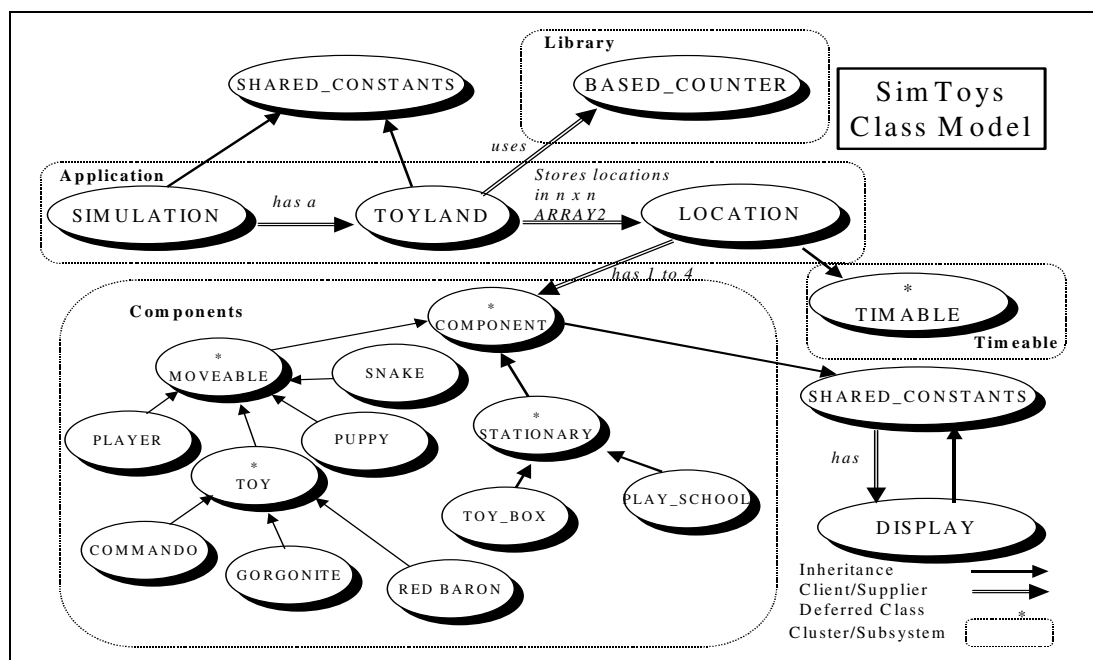


Figure 1. SimToys class model

Metric details defined in the PSP are also required of students. Progression through the assignment stages provides students with a view of personal software process improvement.

3.1.2. Stage 1 of the assignment consists of students performing three of the four maintenance activities outlined in Section 2. These are:

- *corrective* maintenance where the errors in **stage 0** need to be found via regression testing and subsequently fixed,
- *perfective(enhancement)* maintenance by adding new functionality to the system in the form of another class, and
- *preventative(reengineering)* maintenance where they are required to extract common code into an abstract class providing reuse facilities.

At the end of **stage 1** of the assignment, students are asked to exchange their source code with that of another group and perform a technical review. Not only does this provide them with experience of some of the quality aspects of software engineering practice, but also exposes them to a different solution for the same problem.

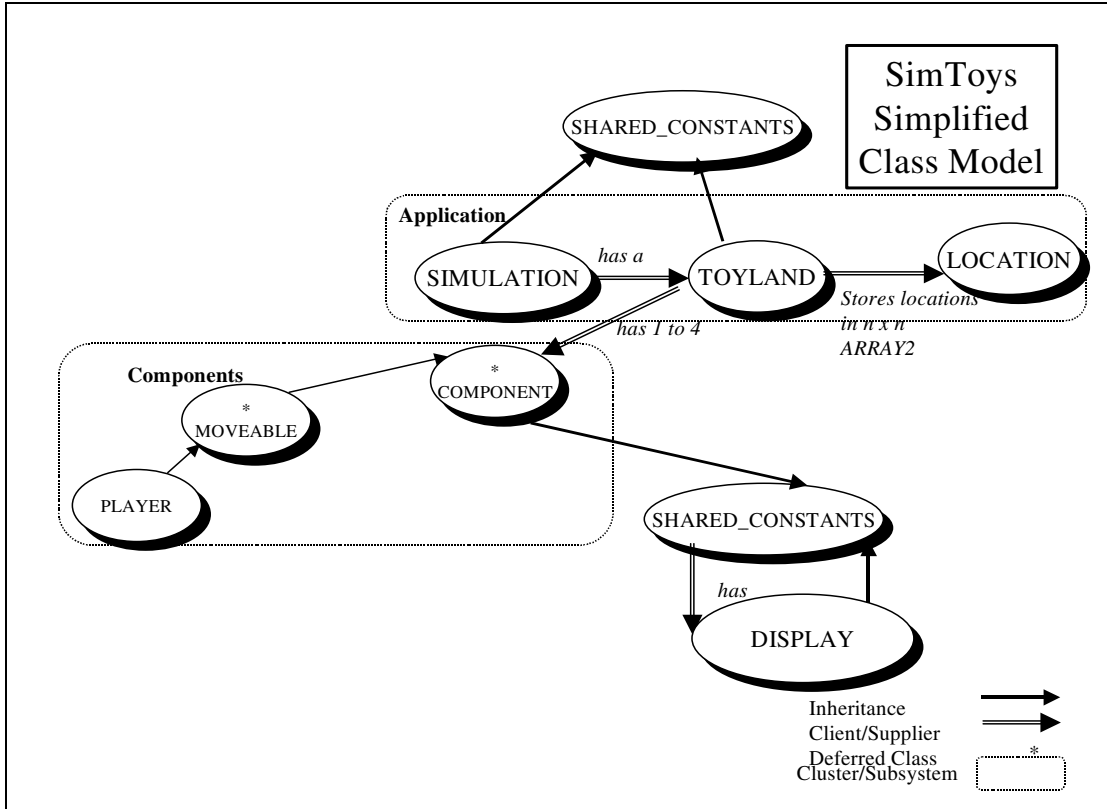


Figure 2. SimToys simplified class model

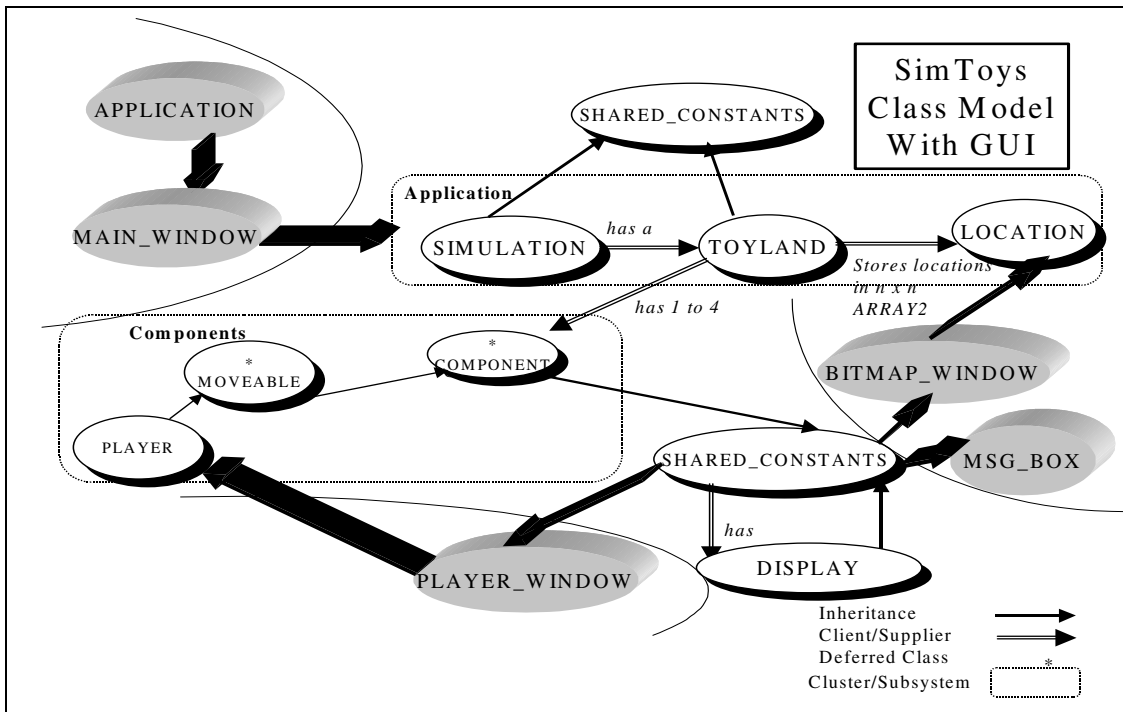


Figure 3. SimToys class model with GUI

3.1.3. Stage 2 further involves enhancement with more difficult functional requirements such as adding an O-O command schema to the program, and using an algorithm for components to traverse the array in an intelligible manner, rather than the previous random fashion. Prior to implementing the **stage 2** requirements, students need to conduct an impact analysis.

3.1.4. Adaptive maintenance - adding a GUI. In semester 1 2000, students were introduced to the fourth maintenance activity (*adaptive*). This involved the adaptation of a graphical user interface (GUI) to the program. The GUI classes developed using the Windows Eiffel Library (WEL), required integration to the existing program. The simulation was designed using the Model View Controller (MVC) design pattern. See Figure 2 for a simplified version of the SimToys Class Model. Figure 3 highlights how the GUI view classes could simply be plugged into the program. The *application*, *main_window*, *msg_box* *player_window*, and *bitmap_window* classes were developed independently, and could easily be integrated into the existing simulation, thus changing the interface. This made it relatively easy to substitute the original text output (see Figure 4), to the Graphical view as can be seen in Figure 5. The output in Figure 4 displayed a lot of text messages during the simulation requiring the user to “*press the return key to continue.*” It was also difficult to view movement of the components. The GUI simply displayed the components (up to 4) in each location of a 5*5 array, in the simulation as bitmaps (Figure 5). A visual display of components actively moving around the grid was now possible, which was previously cumbersome to view on the textual output.

```

C:\WORK\SFT2201\sem1-2000\stage1\EIFGEN\W_code\simtoys.exe
Welcome to SimToys
We may be small, but we pack a punch
Day: 1
<1:1> <1:2> <1:3> <1:4> <1:5>
P--- C--- S--- BG#- B---
<2:1> <2:2> <2:3> <2:4> <2:5>
#--- GG--- CB--- BD--- #-
<3:1> <3:2> <3:3> <3:4> <3:5>
CT--- GCGT *--- --- C---
<4:1> <4:2> <4:3> <4:4> <4:5>
--- BC--- #- C--- GDT-
<5:1> <5:2> <5:3> <5:4> <5:5>
--- DC#- BT--- GC--- $#R-
Press the return key to continue
-

```

Figure 4. Text output for SimToys

The graphical user interface to the system was designed and implemented by one of the subject staff members. Students were then given a document explaining how to integrate the GUI component of the program. It should have been a relatively simple task for the students to perform, by just cutting and pasting the provided code into the appropriate classes. A relatively small number of classes were affected. Most of the changes needed to be made in one class (*shared_constants*), and additional minor changes were required in the system (*Ace*) file to enable program execution to commence from a *wel_application*, creating the *main_window*.

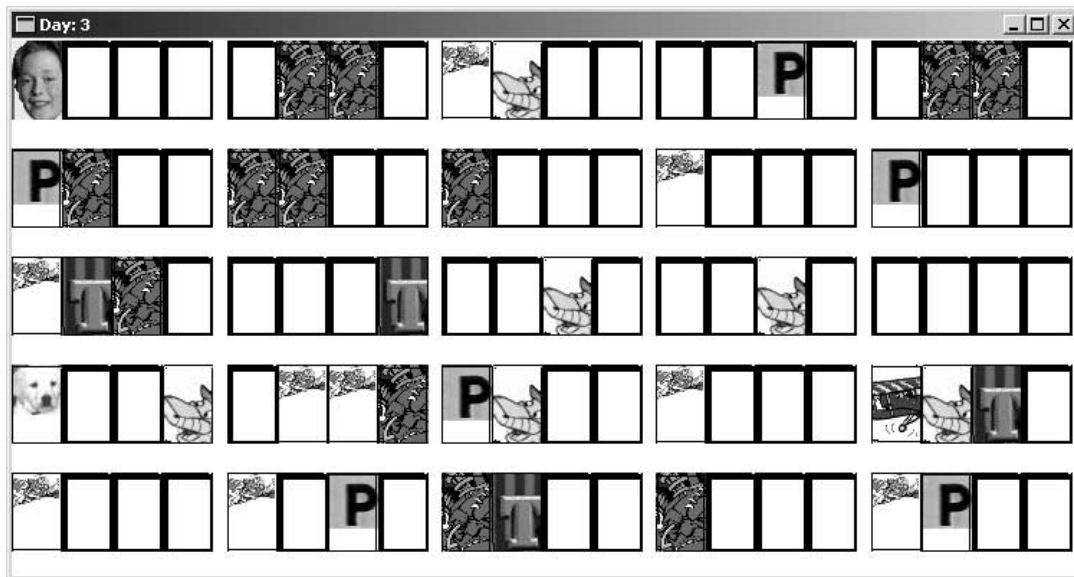


Figure 5. GUI output for SimToys

We initially thought that this task would prove to be too simple for the students to perform. Whilst the instructions given to the students were not simple step-by-step ones that would be presented to first year students, we considered them quite adequate for second year students. They could be compared to typical level of instructions for installing software upgrades, or adding new components to a software application. This was confirmed, by getting a subject tutor (not familiar with Eiffel WEL) to perform the GUI migration, prior to handing it out to students. The tutor found the instructions easy to follow and the task took a total of ½ hour to perform.

3.2. Student perceptions

Subject surveys have been conducted in CSE2201 over the last four years to evaluate the range of teaching tools used [1]. Whilst the assignment is consistently viewed as too heavy a workload, it is also rated as being of reasonable interest and is perceived as contributing to student knowledge of software engineering practice. The GUI was introduced this year in an effort to reduce the number of students (15%) that found the assignment boring.

We were totally unprepared for the comments, feedback and problems caused by the GUI for some students. Others seemed to have no problems with it, and enjoyed the GUI to the extent that they not only put more exciting bitmaps in the program, but added tool bars, sound etc.

The problems reported to tutors and posted on Anonymous Feedback¹ can be summarized as follows:

- not changing the directive to include the *wel* precompiled library instead of the *base* (i.e. changing one word in the Ace file). ***This was the most common error performed by the students, and typically gave compile errors. Students then assumed the instructions were incorrect, and either placed comments to that effect on Anonymous Feedback or sought assistance from their tutors.***

¹ Anonymous Feedback [5] provides an avenue for students to participate in subject administration.

- not changing the root class from *simulation* to *application* in the *Ace* file, or having more than one *application* class. ***This was the second most common error; whilst the program would compile and run, the window showing the bitmaps was not displayed. Once again students commented that the instructions were incorrect.***
- not understanding that input and output was now simply conducted via the use of list boxes and message boxes. ***Students were not required to understand the functionality behind these constructs, but provided with instructions on how to use these.***

Overall, it appeared students enjoyed seeing the GUI to an Eiffel program, as they previously thought Eiffel wasn't capable of displaying a GUI, even though we point out to them that the Eiffel environment is written in Eiffel.

Further comments from students in employment are encouraging. They report that the practice taught in this subject has been beneficial in their working environment.

4. Conclusion

The practice of teaching software engineering to undergraduate students usually involves students writing a program from scratch. The theory of maintenance activities is usually part of the course; however there seems little evidence of students actually applying this theory in the practical component. We have developed a very practical oriented subject, which teaches software engineering as software evolution.

We not only introduce students to the theory, but also additionally have them practice the four important activities of *corrective*, *adaptive*, *perfective (enhancement)* and *preventative* maintenance. Given that maintenance is where most of the effort is expended on software projects, our undergraduate students are provided with practical experience preparing them for obtaining their typical job of maintenance programmer.

5. References

- [1] Dick, M., Postema, M., and Miller, J. Teaching Tools for Software Engineering Education. *Proceedings of the 5th Annual Conference on Innovation and Technology in Computer Science Education*. July, 2000, ACM Press, 49-52.
- [2] Dick, M., Postema, M., Miller, J. and Cuce, S. Personal Assistant for Software Engineers – Relieving Software Engineering Tedium. In N.C. Debnath and R.Y. Lee (Editors) *Software Engineering and Applications (SEA'99)*. October 1999, IASTED/ACTA, 282-287.
- [3] Griswold, W.G. CSE218: Advanced Topics in Software Engineering: Methods and Tools for Evolutionary Design. Online: Internet. [August 4, 2000] <http://www-cse.ucsd.edu/users/wgg/CSE218/index.html>
- [4] Humphrey, W.S. *A Discipline for Software Engineering*. (1995) Addison-Wesley.
- [5] Lowder, J. and Hagan, D. *Web-based Student Feedback to Improve Learning*. Proceedings of the 4th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, Krakow Poland, June 1999, 151-154.
- [6] Miller, J. and Mingins, C. Putting the Practice into Software Engineering Education. In S. Cranefield, M. Purvis and S. Macdowell (editors), *Software Engineering Education & Practice*, January 1998, IEEE Computer Society, New Zealand, 200-207.
- [7] Mingins, C., Miller, J., Dick, M., and Postema, M. How We Teach Software Engineering. *The Journal of Object-Oriented Programming*. 11(9), February 1999, 64-74.

- [8] Nelson, D., and Ng, Y.M. Teaching Computer Networking using Open Source Software. *Proceedings of the 5th Annual Conference on Innovation and Technology in Computer Science Education*. July, 2000, ACM Press, 13-16.
- [9] Postema, M., Dick, M., Miller, J., and Cuce. S. Tool Support for Teaching the Personal Software Process. *Computer Science Education*, 10(2), August, 2000. Swets & Zeitlinger, 179-193.
- [10] Pressman, R.S. **Software Engineering: A Practitioner's Approach**. Fifth Edition 2000, McGraw-Hill.
- [11] Somerville, I. **Software Engineering**. Sixth Edition. Addison-Wesley, 2001.
- [12] Wiener, R. *Software Development Using Eiffel: There can be life after C++*. 1995, Prentice Hall.