Complecto Mutatio: Teaching Software Design Best Practices Using Multi-Platform Development

Randy Connolly
rconnolly@mtroyal.ca
Department of Computer Science & Information Systems,
Mount Royal College
Calgary, Alberta T3E 6K6, Canada

Abstract

This paper argues that students can best appreciate the benefits of software design principles when they have to work on a project in which requirements change repeatedly in some substantial way over the course of a semester. This paper describes two different semester-long projects in which substantial change was enforced upon the students by making them develop a system that had to work on three different user interface platforms (text-based console, desktop Windows, and a mobile Pocket PC). By making the students plan and adapt for this change the students were better able to truly appreciate the benefits of good design and were willing to take the extra effort to implement a design that reflects the principles taught in most object-oriented design courses. One of the key principles engaged by this approach was the importance of a layered architecture to software projects driven by change.

Keywords: software design, layers, user interface, extreme programming, agile software development, mobile computing, game development

1. INTRODUCTION

"Observe always that everything is the result of change, and get used to thinking that there is nothing Nature loves so well as to change existing forms and to make new ones like them."

-- Marcus Aurelius (1882)

"Software changes its own requirements."

-- Ken Beck (2000)

As Beck's quote indicates, Emperor Marcus Aurelius's advice to himself to be stoic in the face of change is as relevant today for software developers as it was in the 2nd century AD for Roman emperors. Of course, today's developers are generally less concerned with barbarian incursions and unreliable Praetorian Guards and more concerned with shifting requirements and fast-approaching deadlines. For if we replace the word "Nature" in Aurelius's maxim with "Clients" we will then have some very sound advice for any practicing or prospective software developer. From this author's own experience, clients very much do love to change existing (Windows or Web) forms and force developers to make new ones that are very much like the old ones but yet different enough to cause anguish to the project deadlines!

Certainly many software design researchers and practitioners have noted the ubiquity of change in the typical software project. For instance, one high-profile study showed that business rules for a typical software project changed at the rate of 8% per month; another study indicated that over 40% of requirements arrive only after development is well under way (Larman, 2004). These types of figures do lead one to conclude, as does Hazzan and Dubinsky that for us developers, "Changes are all around us" (2006a). While change may be a good thing from a

global, evolutionary perspective, it does result in significant problems for software developers. "Managing the effect of changing requirements remains one of the greatest challenges of enterprise software development." (Datta and Engelen, 2006).

The commonplace and yet problematic nature of change in the software development world is perhaps the principal reason for the decline in commitment to waterfall development models and the concomitant rise in interest in iterative and agile methodologies. In fact, the subtitle of one of the key texts (Beck, 2000) in the field of iterative development is the English equivalent of the Latin words in the title of this paper, namely, *Embrace Change*. In Beck's celebrated formulation of the extreme programming (XP) approach, developers must have the courage to face up to change and use a method that frees the developer from excess documentation and analysis so as to be able to respond quickly to changing requirements.

The Latin word *complecto* captures the feeling that Beck claims that developers need towards change perhaps better than the English equivalent, since it also connotes the grappling embrace of hand-to-hand combat. Developers do indeed often struggle with the inevitable strife caused by changing requirements. Yet as some authors have noted, a developer's attitude towards change does not have to be just that of a warrior; it can also be that of a lover. That is, rather than just fighting change, developers and designers should see "change and adaptation as unavoidable and indeed essential drivers" in the creation of more maintainable and adaptable software (Larman, 2005).

Yet despite the current wide-spread use of iterative approaches in real-world software development and the attempt by many teachers to integrate these more agile processes into computer science education (Jones, 2003; Koster, 2006; McKinney and Denton, 2005; Sherrell and Robertson, 2006), the essential ingredient of change can be difficult to add into a typical one semester course. If we accept the premises of the constructivist, problem-centered learning approach (Ben-Ari, 2001), students will only learn the benefits of these agile methods if they gradually engage in their use, since "methodology embodies meaning only after engaging in the process." (Laware and Walters, 2004) Several software engineering educators have tried different approaches to achieve this aim (Hazzan and Dubinsky, 2006b; Loftus and Ratcliffe, 2005; Mitra and Bullinger, 2007; Reed et al, 2004). Yet, as Christensen has noted, "many programming assignments in education are formulated by using the exact same parameters as [waterfall-based] industrial projects." (Christensen, 2008) That is, students are typically given a complete and unchanging set of functional requirements that must be implemented by some fixed deadline. The result is that "the predominant way of stating assignments contributes to the same negative impact on quality as is often observed in industrial projects." (Christensen, 2008) This is a particularly unfortunate shortcoming since the value of many of the most important software design best practices can only be appreciated in a project that is undergoing a significant amount of change.

To avoid this problem some researchers have advocated integrating change into the students' software assignments by making the assignments in a semester form part of a larger, integrated project (Christensen, 2008; LeJeune, 2006; Loftus and Ratcliffe, 2005). By changing an assignment's requirements over time, the key pedagogical reward is that "the value of the design phase becomes very clear after a design feature must be modified." (Laware and Walters, 2004)

The rest of this paper details this author's approach to forcing students to manage changing requirements in two different semester-long development projects. The novel element in these courses was the change element. Each project had to be implemented on three different user interface platforms: text-based console, desktop Windows, and a mobile Pocket PC. The paper will also examine what was perhaps the most lasting lesson learned by the students in this approach: the importance of design principles in general and of layered architectures in particular, to software projects driven by change.

2. THE PROJECTS

In our three-year applied degree program, students are exposed to a variety of application development environments. Students take three programming-focused courses

(from structured to object-oriented) using Java and C#, two courses devoted to web application development, and two courses teaching database design and development. The course referred to in this paper is an additional fifth-semester course in Windows development that uses C# and Windows Forms within Microsoft's .NET Framework.

In one version of the course, the development project was a game based on a popular board game. Within the field of education, there is "an abundance of literature to support the use of games as tools that help learners." (Mungai, Jones, and Wong, 2002) Within the context of computer science, a variety of researchers have found game assignments to be helpful for teaching and motivating introductory programming students (Becker, 2001; Giguette, 2003). Game projects also provide an ideal context for teaching the more "higher-order" and abstract software development topics such as architecture, design patterns, and software methodology. Indeed, it has been noted that games can provide "an extremely project-oriented, upper-division course to exercise and enhance the programming and problem-solving skills of advanced students" (Jones, 2000). It should also be noted that in this section, unlike the first mentioned section, the student body was entirely male.

In the other version of the course, which had a mixed gender balance, the development project was a restaurant browsing and ordering application. Since this cohort had already had a game project in their third programming course, it made sense to have them finally do a more "real-world" project. The number of students in each version of the course was quite low (less than twenty). The students worked in pairs, but did not pair program or follow any explicit methodology. A small number of design artefacts (i.e., class diagrams and screen prototypes) also had to be created at various points during the semester.

Change was a vital part of both projects. Each project was broken down into four distinct milestones. In three of the milestones the project had to be adapted to a completely different user-interface platform. In the first milestone the students had to provide a text-based console version of the basic project functionality (see Figure 1).

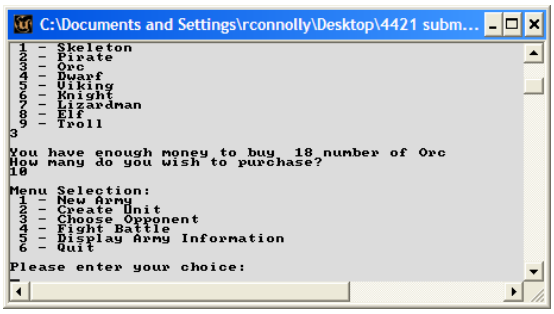

Figure 1. Console version of game project (first milestone)

In the second milestone the students had to convert their first milestone to work as a Windows Forms application (see Figures 2 and 3).

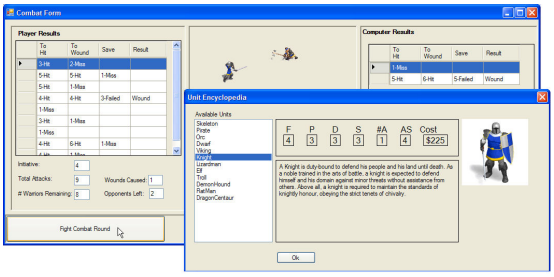

Figure 2. Windows Forms version of game project (second and third milestones)

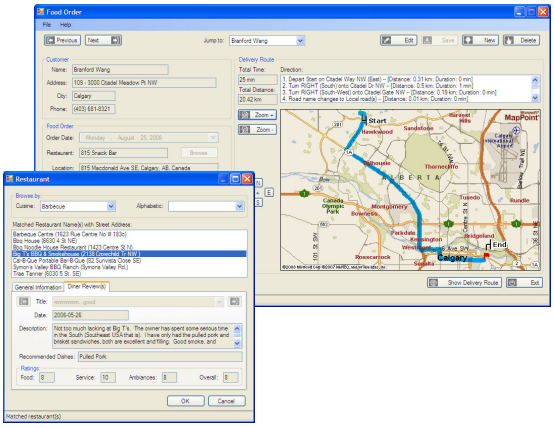

Figure 3. Windows Forms version of restaurant project (second and third milestones)

Rather than changing the user interface, the third milestone made other changes to the projects. In the game project, the third milestone added animation and XML-based load and save game functionality. In the

restaurant project, the third milestone added web service-driven mapping as well as the ability to add reviewer comments.

Finally, in the fourth milestone the students had to convert their Windows Forms version to run on a hand-held Pocket PC (see Figures 4 and 5).

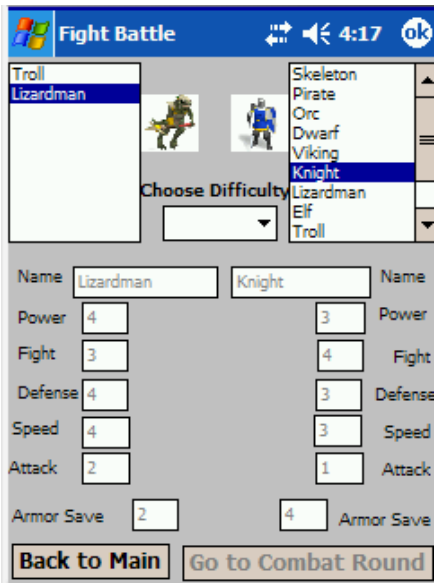

Figure 4. Pocket PC version of game project (fourth milestone)

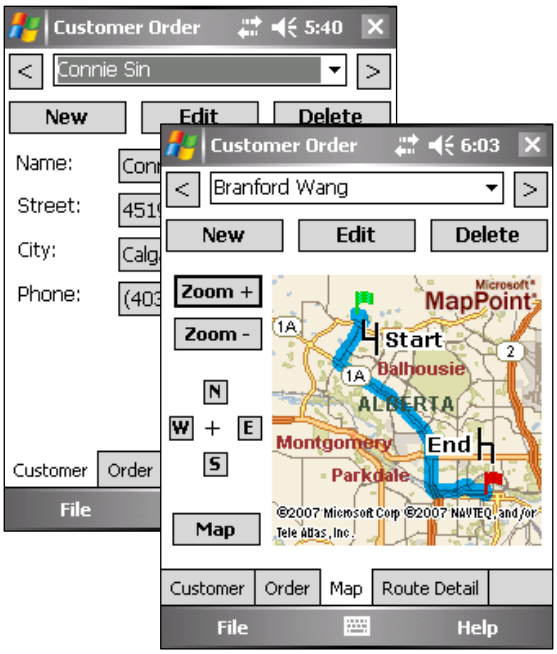

Figure 5. Pocket PC version of restaurant project (fourth milestone)

The rationale for this approach was mentioned in the introduction: namely, to give the students exposure to the kind of requirements change encountered in most real-world projects. While there is certainly nothing new in trying to expose students to requirements change, what was perhaps innovative about the approach taken in the course (and what was especially useful from a pedagogical standpoint) was the manner in which students were exposed to change: that is, by forcing the students to progressively adapt their projects to radically different user interface platforms.

The experience gained by the students here was especially beneficial in several important ways. First, the vast majority of change in real-world projects is in fact at the user interface level. Second, the students became truly appreciative of the benefits of proper object-oriented design, especially the general object-oriented principle that one should separate that which varies from that which stays the same (Gamma et al, 1995). The students also begin to appreciate what Adams (2006) calls the Janus Principle: "Design and write object-oriented applications so that they support multiple, reusable, user-interfaces with minimal redundant coding."

In order to adapt their milestones to these different user-interface platforms, the students were forced to refactor their initial milestone in order to make future transitions less time-consuming. Almost without exception, in the first milestone students intertwined user interface logic into their basic domain model and were faced with spending time eliminating the console user interface elements from their design. To help with this initial refactoring, the students were taught how to separate their domain logic and their user interface logic into two distinct layers.

Using Layered Architectures

What is a layer? A layer is simply a group of classes that are functionally or logically related (Buschmann et al, 1996). Using layers is a way of organizing your software design into groups of classes that fulfill a common purpose. Thus, a layer is not a thing, but an organizing principle.

Layers have of course been an essential part of professional software design since the late

1990s (Evans, 2004). The reason for this convergence on layered architectures is that a layer is not a random grouping of classes. Rather, each layer in an application should be cohesive (that is, the classes should roughly be "about" the same thing and have a similar level of abstraction). Cohesive layers and classes are, of course, generally easier to understand, reuse, and maintain.

The goal of layering is to distribute the functionality of your software among classes so that the coupling of a given class to other classes is minimized. When a given class uses another class, it is dependent upon the class that it uses; any changes made to the used class's interface may affect the class that is dependent upon it. When an application's classes are highly coupled, changes in one class may affect many other classes. As coupling is reduced, a design becomes more maintainable and extensible.

There are many advantages to be gained by designing an application using layers. The first and most important benefit of using layers is that the resulting application should be significantly more maintainable and adaptable by reducing the overall coupling in the application. If there is low coupling between the layers combined with high cohesion within a layer (along with a well-defined interface for accessing the layer), a developer should be able to modify, extend, or enhance the layer without unduly affecting the rest of the application.

This discussion on layers is not that different from what is generally covered in any upper-level design or software engineering course. In this author's experience, students typically are able to echo this material on layers in exams relatively successfully but have a much harder time integrating it into their actual programming practice. To the students, layers and other design best practices often seem like an unnecessary burden for the typical three- to five-week programming assignment. In this project by contrast, student attitudes towards design began to change due to the need to adapt their projects to the different user-interface platforms.

Managing Change via Proper Design

By refactoring their first milestone design into layers the students were able to more easily implement the subsequent platform changes in the remaining milestone. In this author's opinion, the students had become truly receptive to the idea that proper design will actually save them time and effort. Surveyed student comments at the end of the course did seem to verify this impression. Over half the surveyed students indicated that the most important thing learned in the course was "spending time doing good design actually saved me time in the long run because I had to do less coding and debugging," as one student noted.

The key changes in the third milestone (i.e., XML-driven and animation or mapping) required the students to further subdivide their application layers. A new data layer was created in order to isolate the XML interaction and remove it from their domain layer. To handle the complexities of animation, the students were encouraged to split the user interface into two separate layers: a presentation display layer that implemented only the visuals of the user interface, and a presentation helper layer that contained the presentation logic.

The payback for this additional effort arrived in the fourth and final milestone. On the face of it, this milestone was quite intimidating. The students had to move their project to a completely different piece of hardware: a hand-held Pocket PC running Windows Mobile 2003. Yet because the students were using the Compact .NET Framework, they were able to port their domain, data, and presentation helper layers with little or no change. The students only had to redesign and re-implement their presentation view layer in order to fit their project's user interface into the constrained space of the device; as well, their user interfaces had to be changed in order to accommodate the limited GUI controls available to Windows Forms in the Compact .NET Framework. As a result, the final milestone was by far the easiest: most students reported that it only took a day or two to complete. Certainly at this stage of the course the students in both sections had become true believers in the benefits of proper software design. For the very first time in this author's teaching experience, students had not just memorized the design principles nor simply believed in them as an article of faith because the professor told them so. Instead, the students had their own empirical evidence of their utility in managing requirement changes in a

software project. This then is the principle message of this paper: *forcing the students to adapt to change by developing a project on multiple user-interface platforms allowed the students to internalize and integrate software design best practices into their own emerging development practices*.

3. TEACHING EVALUATION

It can, of course, be difficult to construct a project that can be successfully completed by students and which also supports the pedagogical goals for a course. Trying to also integrate change into it makes this process even more difficult. Having the students adapt a project to different user interface platforms is one way to achieve this goal in a relatively painless way for an instructor. In this project, the .NET environment was used, but a similar effect could be achieved by using the Java platform. For instance, the students could create the application first for a Java console interface, then a richer Swing interface, followed by a Java Micro Edition interface or a Google Android interface.

Student Evaluation

The students' written evaluations after the course indicated that there was a great deal of satisfaction with their learning in regards to software design. Several of the students' comments attributed this to the multiple user-interface development approach taken in the course. While the number of students in each section of the course was too low for any meaningful statistical analysis, for what it is worth, the average final exam mark for both groups was over 10% higher than the previous year. In the follow-up sixth semester course (advanced web development), the majority of the students now adopted a layered architecture in their capstone project. More importantly, the average design mark for the follow-up course's project was significantly higher (over 25% higher) compared to the two previous years. Of course, there are a number of factors independent of this paper's theme which could also account for these changes. Nonetheless, it might be *some* (albeit heavily qualified) evidence that the students who had the multi-platform project better integrated the benefits of software design into his or her future practice.

4. CONCLUSION

It can be difficult to get students to fully appreciate the benefits of a proper software design. For most assignments, proper design just seems to be an instructor-enforced hassle because it generally only increases the amount of work for the student in a given assignment. To appreciate the benefit of a proper design, students need to work on a project with substantially changing requirements. In such a project, students are able to see for themselves that proper design can save time and effort.

This paper described two semester-long development projects in which change was enforced upon the students. The most important of these changes was that the project had to be implemented on three quite different user interface platforms. This provided the kind of dramatic change necessary for the students to truly appreciate and willingly implement a software design that reflects the precepts and principles taught in most object-oriented design courses.

5. ACKNOWLEDGMENTS

Pocket PCs were made available to the students thanks to a Dell Mobile Computing Pilot grant.

6. REFERENCES

Adams, J (2006). "OOP and the Janus Principle." Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education, 38 1 (March).

Aurelius, M (1882). The Meditations of Marcus Aurelius, iv 36. Translated H. Crossley. Macmillan, London.

Beck, K (2000). Extreme Programming Explained: Embrace Change. Addison-Wesley, Boston.

Becker, K (2001). "Teaching With Games: The Minesweeper and Asteroids Experience." The Journal of Computing in Small Colleges, 17, 2 (December).

Ben-Ari, M (2001). "Constructivism in Computer Science Education." Journal of Computers in Mathematics and Science Teaching, 20 1.

Buschmann, F., et al (1996). Pattern-Oriented Software Architecture: A System of Patterns, Volume 1. John Wiley & Sons, New York, p. 31-51.

Christensen, M. E. (2008). "Experiences with a focus on testing in teaching." Reflections on the Teaching of Programming. J. Bennedsen, M. Caspersen, and M. Kolling, editors. Springer-Verlag, Berlin, p. 150.

Datta, S., and R. Engelen (2006). "Effects of changing requirements: a tracking mechanism for the analysis workflow." Proceedings of the 2006 ACM symposium on applied computing (April).

Evans, E (2004). Domain-Driven Design: tackling complexity in the heart of software. Addison-Wesley, Boston, p. 68-71.

Gamma, E., et al (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston, p. 23-4.

Giguette, R. (2003). "Pre-Games: Games Designed to Introduce CS1 and CS2 Programming Assignments." Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education, 35 1 (January).

Hazzan, O. and Y. Dubinsky (2006a). "The concept of change in technology transfer." Proceedings of the 2006 international workshop on Software technology transfer in software engineering (May).

Hazzan, O., and Y. Dubinsky (2006b). "A cognitive perspective on software development methods: the case of extreme programming." Proceedings of the 2006 international workshop on Workshop on interdisciplinary software engineering research (May).

Jones, C. G. (2003). "Integrating Agile Development Methodologies into the Project Capstone – A Case Study." Information Systems Education Journal, 1 18.

Jones, R. M. (2000). "Design and Implementation of Computer Games: A Capstone Course for Undergraduate Computer Science Education." Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education, 32, 1 (March).

Koster, B. (2006). "Agile methods fix software engineering course." Journal of Computing Sciences in Colleges, 22 2.

Larman, C. (2004). Agile & Iterative Development: A Manager's Guide. Addison-Wesley, Boston.

Larman, C. (2005). Applying UML and Patterns, Third Edition. Addison-Wesley, Boston.

Laware, G. and A. Walters (2004). "Real world problems bringing life to course content." Proceedings of the 5th conference on Information technology education (October).

LeJeune, N. F. (2006). "Teaching software engineering practices with extreme programming." Journal of Computing Sciences in Colleges, 21 3.

Loftus, C., and M. Ratcliffe (2005). "Extreme programming promotes extreme learning?" Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education (June).

McKinney , D. and L. Denton (2005). "Affective assessment of team skills in agile CS1 labs: the good, the bad, and the ugly." Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education, 37 1 (March).

Mitra, S. and T. Bullinger (2007). "Using formal software development methodologies in a real-world student project: an experience report." Journal of Computing Sciences in Colleges, 22 6.

Mungai, D., D. Jones, and L. Wong (2002). "Games to Teach By." Proceedings of the 18th Annual Conference on Distance Teaching and Learning.

Sherrell, L. and J. Robertson (2006). "Pair programming and agile software development: experiences in a college setting." Journal of Computing Sciences in Colleges, 22 2.

Reed, K. et al (2004). "Agile management of uncertain requirements via generalizations: a case study." Proceedings of the 2004 workshop on Quantitative techniques for software agile process, (November).