MAPPING SOFTWARE ENGINEERING PRINCIPLES TO STAGES IN GAME DEVELOPMENT*

Lakshmi Prayaga
Computer Science Department
University of West Florida, Bldg 4
11000 University Parkway
Pensacola, FL 32514
850 474-2973
Lprayaga@uwf.edu

ABSTRACT

We discuss the mapping of software engineering principles to various stages of game development and present examples from some of the courses offered by us to illustrate this progression. We demonstrate that software engineering principles can be implicitly and effectively taught not only to a mature audience (college students) but also to K-12 students when game development is used as a tool.

1. INTRODUCTION

Game Development is both a creative and a detailed procedural process. Hence it fits the description of software engineering as both a creative and rigorous discipline. Software engineering is rigorous in the sense that it involves use of strict procedural guidelines for software development, and is creative in aspects of requirements analysis, GUI design architecture and software testing (Hooper, C. and Millard, D. E. 2009, Budgen. D. 1995). Game development requires both these aspects, creativity and rigor to develop software that meets the goal of providing entertainment and fun. In the following sections we examine the mapping of software engineering principles to the various stages of game development that require both creativity and rigor.

---

2. SOFTWARE ENGINEERING PRINCIPLES

The importance of following good software engineering practices is widely accepted both at the amateur and professional levels of software development. We describe the implementation of the seven software engineering principles (C. Ghezzi, M. Jazayeri, D. Mandrioli, 1991) including design and development in our game development courses for two levels of students 1. interdisciplinary college students and 2. K-12 students in summer camps. The following sections describe the seven software engineering principles and the implementation of each of these principles in the two types of courses that were offered by University of West Florida.

The seven software engineering principles proposed by Ghezzi et al and used extensively are

- Rigor and Formality
- Separation of Concerns
- Modularity
- Abstraction
- Anticipation of Change
- Generality
- Incrementality

3. IMPLEMENTATION OF SOFTWARE ENGINEERING PRINCIPLES

As mentioned earlier in the paper, we offer game development courses at the University of West Florida to interdisciplinary students at a 3000 level and to K-12 students during summer camps. Our observations suggest that both these sets of students were very motivated in game development courses and camps, actively participated in class, and could be instructed to implicitly follow software engineering principles. Sections 3.1 to 3.7 describe our experiences in the effort to instruct students in basic software engineering principles.

3.1 Rigor and Formality

Software engineering is creative in that each individual creates a unique design for the software to accomplish the same goal. However, it must be stressed that there is a rigorous process that must be followed during software development. Rigor complements creativity in designing a product that withstands the scrutiny of ruthless testing, which is of utmost importance in any software, more so in game development.

In our courses we ensured that students were not hampered in expressing their creativity and yet employed rigor and formality to their projects by

1. Letting their creative aspects manifest in the initial storyline, story boarding and a sketch or concept art of their game and
2. Formalizing their story board into a flowchart to document the progress of the game
3. Using prescribed naming conventions and testing procedures

This strategy worked well for the college students and K-12 students. One observation from this experiment was that the K-12 students paid more attention to the creative aspect vs. the formal aspect of game development. For example, the students in K-12 were very interested in creating clues that were obscure yet interesting, but students in the college group were more interested in the code related aspects of game development such as including particle effects which rely heavily on code and physics. To address rigor, all students were given rules that they must abide by to ensure that the software they design conforms to the rigor in development and testing phases. Examples of rigor included that students follow a strict naming convention for the files, variables, and functions and test the code with a given set of test cases. The idea was to get students used to the notion of rigor and formality.

3.2 Separation of Concerns

Separation of concerns was introduced by Dijkstra(1976) and Parnas(1972) to deal with the complexity of large software projects. It allows for dealing with different aspects of the problem and treating each aspect of the problem separately. In terms of the product being developed separation of concerns can be maintained by separation in terms of functionality, performance and user interface and usability.

Application of this principle to game development in terms of concerns related to functionality resulted in the following activities: having students describe the story line of the game, game play including rules of the game, scoring system, health bar and lives to keep track of the progress of the player, skills required for win conditions, detailed descriptions of the characters, protagonists, and antagonists' in a role playing game.

Separating performance of the system was achieved by documenting and implementing test cases to verify that the system designed adhered to the expected performance outcomes of the test cases. Example test cases were checking that the score card, health bar, lives in the game were being updated based on player performance. Additional testing scenarios included checking boundary values and extremities a very common test case in software engineering. An example scenario was to have students test the speed variable of a car which has an initial speed set to 5 mph. Students were asked to test the performance and students immediately tested it with 5.1, and then 40 mph and noted its effects. The point from this observation was that students did not need any prompting for what kind of test cases they need to use, they were inherently motivated to try different test cases on their own. Most of the senior students also wanted to know why the test case failed when they pushed the limits which provided the instructor with the opportunity to explain the theoretical aspects of how the notion of speed works in a frame based environment such as Flash. Setting the speed to 5 miles means the car moves 5 pixels per frame (assuming advancing 5 pixels per frame), setting 40 miles would mean that the car moves 40 pixels per frame. The problem then is that if there is a road block set at 30 pixels the car won't see it, since it jumps in increments of 40.

User interface was captured by student descriptions / mock screens of how they wanted the players to interact with the game. College students were also required to read about best practices in screen layout to learn about the best spots for health bars, lives, score cards, and navigation buttons in designing a good user interface (Novak, 2008)

3.3 Modularity

Modularity is the division of a complex system into various parts. Following the process identified in the separation of concerns, students identified the various modules required for the game (main character/piece, enemy, score card, high score) and designed (graphics and code) each piece individually. They also wrote functions which had high cohesion - low coupling (Hortsmann, 2008) a well accepted software engineering principle and used them at various places. Examples include design of components in Flash (enemy ships), which could be reused several times in the game. Students even at the K-12 level appreciated the reusability of components and realized that they did not have to code each enemy ship individually since the component has all the code and behavior embedded in it. In group projects for college students, modularity also played another major role in that it allowed for assignment of individual modules to various members of the group which were all integrated at the end. This experience truly showed students the importance and effectiveness of the principle of modularity in software development which provides individualized implementation as observed by Parnas (1972) and further investigated by Huynh for measuring level of independence when working with modularity (Yuanfang Cai, Sunny Huynh, 2008). The team had to experiment with levels of independence for different modules, some of the modules could be worked on independently (example graphics, layouts) while others (code related aspects such as maintaining high scores, integration of modules) required more input from team members.

3.4 Abstraction

Abstraction is the principle of identifying the important aspects of the phenomenon and ignoring the details. This process was achieved by having students follow a top-down approach to the game. Students were asked to generate a block diagram for their game each block representing an aspect of the game at a high/abstract level. Each of these blocks were then converted to modules and coded (as discussed in modularity). The process of abstraction allowed students to develop a high conceptual level of their software (game) which could then be divided into different modules depending on the task for each module. In most cases this aspect resulted in a basic story board identifying the various blocks

3.5 Anticipation of Change

For software to be reusable, the designer should anticipate that many users would like it to be changed and tweaked to meet their customized requirements and identify areas/modules where such changes can be accommodated. A simple example in our courses was designing puzzle games. Students quickly found out that the number of pieces in a puzzle was a volatile variable, some people want a 4 piece puzzle, others may want a 20 piece puzzle. This meant that students had to tweak the code to accommodate both single and double or triple digit numbers in their code. Other examples were providing the facility to have different interfaces for the same code. An example is to have students design their own enemy ships, and warriors but use the same code. This was especially important in a two week summer camp where the goal was to use game

development as a tool to let students have fun, and motivate them to get into computer science related courses, not so much to teach programming. Figures 1 and 2 from k-12 audience show different front end scenarios used the same generic code, which centered around the theme of space invaders. Students had to include ideas of a good guy, enemies, score and a timer. Students used their creativity to design various objects that conform to these requirements, including space ships, ninjas, swords, and tanks. They could also easily change these images for others if they felt they wanted a change, thus implementing the notion of anticipation of change.

Figure 1 – Gold Hunt

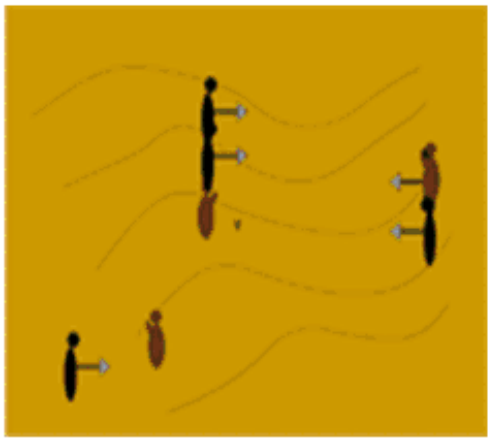

Figure 2- sword fight

College students were less interested in graphics, but anticipated change in terms of user interface, screen layout and designed their code to fit this requirement. Games included various levels of difficulty by changing the pattern of the targets on the screen in a Brickles game, or reducing the time limit to achieve the same goal. Figures 3 and 4

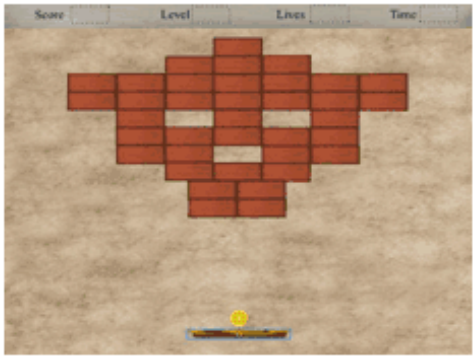

Figure 3 – Pattern 1

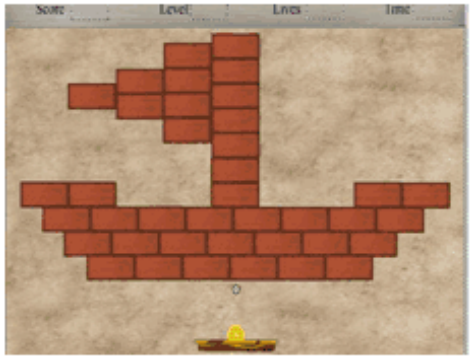

Figure 4 – Pattern 2

3.6 Generality

Generality is the process of discovering and documenting issues in problem solving and generalizing them so that they can be reused in other similar problem cases. Students had many opportunities to discover and document such findings in the game development courses. Examples are discovering the principle of pixel to frame rate correspondence and

its side effects as described in the separation of concerns section and it's applicability in other scenarios such as collision detection. The general principle here is that if an object is moving towards another object/obstacle and the increment in pixels per frame is more than the distance to the obstacle then the object will miss the obstacle completely. In this situation a hitTest() function which detects collision, would return false!

3.7 Incrementality

The process of building software in small segments or units is incrementality. The advantage of this process is that you can obtain feedback from users and add new features incrementally (Larman, Basili, 2003, Basili, Turner, 1975). This process was achieved in game development by having students build each level incrementally, example build the welcome screen, followed by building user instructions and other scenarios. Each scene had to be designed and tested before moving to the next scene following the incremental life cycle model in software engineering. During the summer camp we had to set up time limits for each level to ensure that the game could be completed in 2 to 3 sessions. We found that when such limits were not placed, students spent disproportionate amounts of time on certain aspects of the game (designing one particular graphic, or tweaking one code component) and neglected the others. At the college level students were given goals to be completed before they moved on to the next level and this approach was quite successful.

4. CONCLUSION

Game development can be used very effectively to teach basic software engineering principles to both K-12 and interdisciplinary students. In fact our experience has illustrated that game development can be a motivating tool for this population of students to enhance their programming skills. Students were more willing to experiment with the code, and user interface in this environment. Students did not hesitate to ask questions about their code in a game, which they usually refrain from in normal programming courses. We therefore believe that game development can be used efficiently to teach formal aspects of software engineering including testing, user interfaces, documentation, software process and other related courses.

REFERENCES

[1] Basili. V., and Turner, J., "Iterative Enhancement: A Practical Technique for Software Development,"*IEEE Trans. Software Eng*., Dec. 1975, pp. 390-396

[2] Budgen. D., (1995), 8th SEI CSEE Conference, Pg, 239, New Orleans, LA

[3] Dijkstra, W, E., A Discipline of programming, Prentica Hall, Englewood Cliffs, NJ, 1976

[4] Ghezzi, C., Jazayeri, M., Mandrioli, D., *Fundamentals of Software Engineering*. Prentice-Hall, 1991

[5] Hooper, C. and Millard, D. E. (2009) Creative Software Engineering. In: *Grace Hopper Celebration of Women in Computing 2009.*, http://eprints.ecs.soton.ac.uk/17920/

[6] Horstmann, C., (2008), *Big Java*, 3rd edition, Pg 339, Wiley Publishers

[7] Larman, C., Basili, R, V., (June 2003). "Iterative and Incremental Development: A Brief History". *IEEE Computer* (IEEE Computer Society) 36 (6): 47-56.

[8] Novak, J., *Game Development Essentials*, second edition, pg 243-247, Cengage Learning

[9] Parnas, L, D., On the criteria to be used in decomposing systems into modules, *Communications of the ACM*, Volume 15, No. 12, 1972

[10] Yuanfang Cai, Sunny Huynh, (2008), Measuring Software Design Modularity, *Proceedings of the 2nd Workshop on Assessment of Contemporary Modularization Techniques* (AcoM.08)