Small Group Software Development: A Case Study

Rob Jansen
University of South Carolina
University of Minnesota, Morris
[phone number]
jans0184@morris.umn.edu

ABSTRACT

The development process that a group uses to design software is important for determining the success of a project. Although considerable research discusses processes best suited for large companies, this paper presents a case study of software development by a small group and considers the process taken towards the completion of a project. The collaborative undergraduate team project analyzed included the design and development of a cell phone game using the programming language Java 2 Micro Edition. Many development processes exist, such as the waterfall model, the spiral model, the unified process, and extreme programming. These were compared with the process the team used. The analysis suggests the process was too detailed for a small group environment. In this situation, it was found that less time should have been spent on design. My research shows the need for a simplified method usable by small groups that allows adaptation to changes and takes advantage of the close communication environment of the group.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques – *software libraries.*

General Terms

Design, Experimentation, Languages.

Keywords

Software, design, development, engineering, small, group, methodologies.

1. INTRODUCTION

When discussing software development as a research opportunity, small groups are often overlooked. Most often the topic is software development for large corporations, with a goal to create an efficient method of development for the specific company. Large companies seem to be most concerned with their bottom line, so they look for methods that are efficient and satisfy their financial concerns. Small group software development is important and needs to be considered as well. This kind of development exists both in the engineering industry and in the world of academia. My goal is to analyze a method that is used by small groups at the University of South Carolina. I will analyze the method by participating in a small group project as part of my research. I will compare the process my group used to other well known general development methods while stating the strengths and weaknesses based on my experiences. My research will provide insight for professors wanting to find a development process that works for small groups. It will also be a resource for students wanting to look into which processes have worked for other students, what hasn't worked, and which processes are recommended. Students will be particularly interested because they might find themselves in similar situations. It can give them an idea of what happens during the development process and what to focus on while they are developing software applications.

2. RELATED WORK

As mentioned earlier, there is much more research done in the area of large group software development than small group. In a simple search on Google scholar, the term "small software development" returns 365 results, while the term "large software development" returns 1,160 results. However, this isn't to say that small development research does not exist. For instance, Mark Paulk discusses what small means in terms of a "small" project in his paper "Using the Software CMM in Small Organizations" [4]. There has also been comparisons made on small group and large group development projects in an academic setting as discussed by Michael Stein in his paper "Using Large VS Small Group Projects in Capstone and Software Engineering Courses" [3]. Specific engineering methods have been produced for small companies as well. In their paper "Wisdom: A Software Engineering Method for Small Software Development Companies" [5], Nuno Nunes and Joao Cunha present a method that addresses the needs of small software development teams. Their method emphasizes small teams' communication, speed, and flexibility.

3. DEVELOPMENT MODELS

The development process that a group uses to design software is important for determining the success of a project. There are many well know processes along with some less known methods. Here I will provide you with some background information on four general processes that exist. These processes are general enough to cover the ideas found in most existing processes. The models consist of 5 phases that form a general framework for development [1]: communication, planning, modeling, construction, deployment. During the communication phase, the project gets initiated and requirements are gathered so the team knows exactly what needs to be done for the completed application. In the planning phase, the team estimates the amount of time it will take to complete the project. After estimations are made, the team develops some kind of plan for completing all the work. This plan could include, among other things, dividing up work between employees and setting deadlines. The team might also set up a tracking system that will be used to store and share code between employees. This will ensure that the code is accessible to everyone and can be updated safely. During the modeling phase the team analyzes the requirements that were discussed during the communication phase. Based on the requirements, the team begins designing the system. Design includes user interface design, UML models used by the company, and system design. During the construction phase, the coding takes place. The interfaces and classes are constructed and tested in code. The design that took place during the modeling phase is followed. The last phase, deployment, represents an ongoing phase. During this phase, the final program is delivered to the customer. Support and feedback are offered as an ongoing part of creating the software. After all, who knows the program better than the designers!?

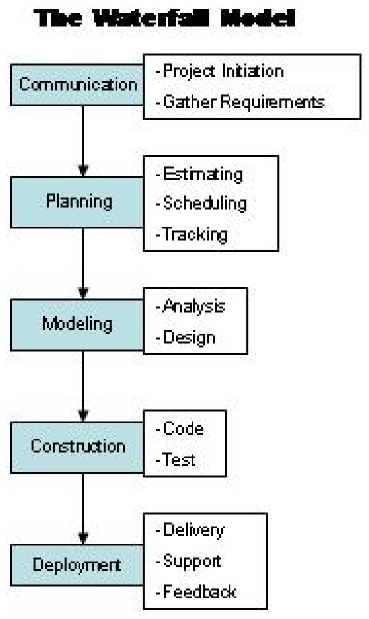

Figure 1- The phases of the Waterfall Model.

3.1 The Waterfall Model

The waterfall model is the oldest paradigm for software development [1]. The waterfall model represents a linear, systematic approach to software development, as represented in figure 1 above. The phases of the waterfall model include the general phases described above. The main idea behind the waterfall model is to follow the process in a logical order. By following each phase of the "waterfall", you theoretically finish the project after you complete construction. However, ongoing support for the product continues long after construction is complete.

This model is particularly useful when the requirements of the project are well understood and are very stable. However, this is almost never the case in real projects. Most software projects evolve as the customer makes decisions about the program. Since the process takes a while to complete, a change in plans near the end of the process could be disastrous. Many times the customer does not know exactly what the requirements are, forcing the team to make their best decisions on what would satisfy the customer. Also, a product will not be available until near the end of the process. This could make customers impatient and it could be difficult to decide if they like the program or not until it is finished. This does not give them much flexibility to change their mind about the way things look without spending more money on the project by expanding it. Another problem with the waterfall model is the possibility for blocking states to occur. A blocking state is when a part of development is held back because it depends on another part that isn't finished yet. Often, the time spent waiting on other parts to finish exceeds productive time. This method is thought to be an "old" style, but is still applicable in some current day projects.

3.2 The Spiral Model

The spiral model uses the systematic method of the waterfall model, but does so in such a way as to implement an iterative approach. Each iteration represents a pass around the spiral. Although the spiral model uses the same general framework ideas discussed earlier, it helps reduce risk and uncertainty by developing an early prototype. As the project proceeds through iterations, the prototype is developed into more complete versions of the actual program. This helps keep the customer in tune with the current state of the project. It also allows the team to get early feedback from the customer so a more desirable program can be produced as a final product. Each pass through the planning stage allows the team to adjust its schedule and the estimated cost. The spiral model can be applied to the project at any point in its development. If an enhancement is planned for a completed program, the spiral model can be reapplied using an entry point in the correct phase.

The ability to handle evolutionary projects is an advantage the spiral model hold over the waterfall model. However, the spiral model has its problems too. There is a possibility that customers will not believe the evolutionary approach to software development is controllable. This process also requires that risks are uncovered and addressed with expertise. As with the waterfall model, if a problem is not uncovered early, the results could be disastrous to the project.

3.3 The Unified Process

The unified process is an attempt to create a software development process that unites aspects of early prescriptive models with principles of agile programming. The unified process consists of four phases: inception, elaboration, construction, and transition. To tie these phases with the generic framework discussed earlier, the inception phase consists of communication and initial planning. The elaboration phase consists of the major planning and design. The construction phase is as described earlier, and the transition phase includes testing and product deployment. The product deployment represents a software increment. After deployment, the team continues with the project, working towards completion of the next increment. Communication and planning with the customer, as well as feedback from end-users will allow the team to make changes to the program during development. The unified process uses Unified Modeling Language (UML) design models during design and development. Simply put, UML is a set of terms and mapping standards that make it easy to visualize system requirements, flow, and functionality. It helps create an abstract model of the software system that the entire team will understand. The unified process makes use of production time by conducting several phases of development in concurrency.

3.4 An Agile Process: Extreme Programming

3.4.1 Agile Process

An "agile" software process is created addressing three main assumptions:

1. It is difficult to predict in advance which initial software requirements that have been identified will change and which will not. It is also difficult to predict how the customer's priorities will change as the project progresses.
2. It is hard to know how much design to do before staring with construction. Design is important to gain an understanding of the program, but construction is important to test the design.
3. It is hard to plan the scheduling, planning, design, construction, and deployment phases and how they will carry out as the project progresses.

Basically there is much uncertainty in software development. Agile processes try to address this by providing adaptability to an ever-changing project, program, etc. If a software team is to succeed, it must possess competence, must have a common focus, must collaborate with each other, must have decision making ability, must have a fuzzy problem solving skill, must have mutual respect and trust each other, and must be self organized.

3.4.2 Extreme Programming

Extreme programming consists of activities that form four phases: planning, design, coding, and testing.

During the planning phase, the customer creates user stories for the development team. User stories are similar to use cases in that they describe the functions and features that the program should include in the finished product. The customer breaks up functionality into small parts and writes each "story" on an index card. Then the customer assigns a value, or priority, to each index card. This value is dependent on the business value or priority of the specific function. After the customer sets values on all functions, the development team assigns a cost to each index card. The cost is the length of time the team expects it will take to implement the tasks of that index card. If the team estimates a function will take too long, the function can be broken up to multiple cards. Once a cost has been assigned to all features, the customer will work with the team until a commitment can be made. After the first release of the product, the team computes the project velocity. The velocity is the amount of work the team expects to complete in a release and is usually dependant on the initial iteration. After subsequent releases, the velocity can be updated, however, is not expected to dramatically change. The customer can choose to add, remove, or split stories, or change the priority of a function anytime during the development process. The team will change its plans accordingly.

The main focus during the design phase is the keep it simple approach. The idea behind this approach is to keep all designs as simple as possible, and never add things that are unnecessary for the current task. Instead, the team uses spike solutions. A spike solution is developed when the team encounters a difficult problem. Before it attempts to complete the associated tasks, it breaks up the problem and makes an operational prototype of that part of design. The prototype is then evaluated. The idea is to reduce risk as early as possible and stay within estimates for the story. Refactoring is also part of design. Refactoring is where the code is changed around in such a way that it is easier to read, but doesn't change functionality. Refactoring improves the design of the code after it has been written. It is important to remember refactoring can get quite difficult as the system grows. Since refactoring is part of code design, the design phase occurs both before and during coding. Code is refactored and redesigned continuously as the system is developed.

The coding phase starts with the development of unit tests. The notion of creating tests before constructing code is known as test-first development. The idea is to write tests that the code needs to follow according to the user stories, and then construct the code in such a way that it passes those tests. The team only does what is necessary to pass the tests, nothing more or nothing less. It follows the keep it simple approach that is implemented during design. After the team completes coding the system, these unit tests will serve as instant feedback. Another key feature of extreme programming is pair programming. While writing code, the team breaks up into groups of two people. These two people work together at the same computer. They both work on problem solving with the idea that two heads are better than one. While coding, one person might focus on code design while the other ensures that the code is accurate and that it will fit in with the rest of the system. After the code is completed, it will be integrated together with the work of other team members. This can be done either by a separate integration team or by the same programmers who wrote the code.

The testing phase focuses on the unit tests that have been created during design and coding. The tests are automated so that whenever the code is changed in the future, which will be often due to refactoring, the tests can be rerun to check for accuracy. Testing also includes acceptance tests. Acceptance tests are tests that are specified by the customer and target system functions that are reviewable from the outside by the customer. Once all unit tests pass and the customer agrees with all acceptance tests, the software is ready for its next release.

It is important to remember that extreme programming focuses most of its time on programming and tests. While this method keeps things as simple as possible and gets straight to the point, some people believe that not enough time is spent on design. Spending to little on design results in coding and re-coding and could reduce productivity. It is important to have a sufficient amount of design so that the team is confident about its coding tasks.

4. THE PROJECT

As an experiment, I participated in a group project where we designed and developed a cell phone game suitable for a cell phone using the programming language Java 2 Micro Edition. We designed everything from the beginning with no game requirements given to us. We followed a development process that our advisor uses in his undergraduate classes. The process consists of defining system requirements, building the system, and testing the system. The goal of the project was to have a working game. We were restricted by time since this was a summer program.

4.1 Define System Requirements

4.1.1 Define project

A brief two-paragraph description of what the project will do and who will use it.

4.1.2 Define user personas

Personas are "artificial persons" that are representative of the different types of users. The system is designed to make the primary persona happy but the other personas should not be unhappy.

4.1.3 Define user scenarios

User scenarios describe how someone will interact with the system. This includes actual scenarios that could happen to a person while playing through the game.

4.1.4 Develop the system use-cases

Use-cases refine the user scenarios and define the functions that the system provides. This includes both the normal flow of events and the exceptional conditions that can occur. The use cases are drawn out using UML notations.

4.1.5 Describe the user interface

Describe main interface that the user will interact with. This includes screen interactions and all options the user will have.

4.1.6 Storyboard the scenarios

A story board describes the sequence of actions, user inputs and system responses, to complete a task on the system. It explains what the display will look like at each key point of execution.

4.1.7 Write detailed requirements

Includes all functional, nonfunctional, and constraint requirements that the system must satisfy.

4.1.8 Develop user and usage profiles

Usage profiles quantify how much each system function is executed. Profiles can be developed for multiple personas and compared.

4.1.9 Triage requirements

Based on the usage profiles, the requirements are prioritized based on user needs and utility. Resources needed to implement the requirement are estimated and a subset of the requirements that will optimize the projects success is selected.

4.1.10 Verify requirements

Review the selected requirements to ensure that they are feasible and will work as part of the system.

4.2 Build System

4.2.1 Define the system classes

Classes are derived from the nouns in the user scenarios. Every system activity or primitive function should reside in a class.

4.2.2 Define system sequence diagrams

Sequence diagrams show how the classes work together to execute each use-case function.

4.2.3 Define an interface for each class

The interface specifies the functions each class will show to the outside world.

4.2.4 Implement the system

The class skeletons are coded based on the code generated from the class diagrams. All coding takes place here.

4.3 Test System

The third and final phase in the process was to test the system. However, due to time constraints, we were unfortunately unable to do extensive testing on the system. Instead, we ran visual acceptance tests by running the program and playing through the game. This proved helpful in ensuring that the game is error free, not necessarily that the system is error free.

4.4 The Game

My group followed the steps required in the process described above to complete the design phase. Once design was complete, I completed the build phase. Carlos Rivera created all the graphics that would be used in the game. The design phase took about three weeks, and the build phase took two weeks. Most of the design ideas remained constant through the building phase; however, most system plans did not remain intact. Several changes needed to be made once coding started. Steps in the design phase that were useful during building include the user scenarios and the requirements. These two design steps were helpful as a place to get started coding, but did not in any way limit me in the creation of the game. The other design steps I view as helpful only in that it gave us direction and gave the game an identity.

The game concept is to help a ladybug through a maze and help it find the exit. The game idea is very simple. As the ladybug progresses through the level, it can turn various “seeds” into “flowers” as a way to keep track of where it has already been in the maze. The ladybug receives points for creating the flowers and for completing the level. Our final game had two simple levels. The user has various options throughout the game, including viewing instructions, credits, the current score, restarting the current level (which will reset the score to 0), restarting the game, and exiting the game. After the completion of the game the ladybug finds its “princess” and presumably lives happily ever after.

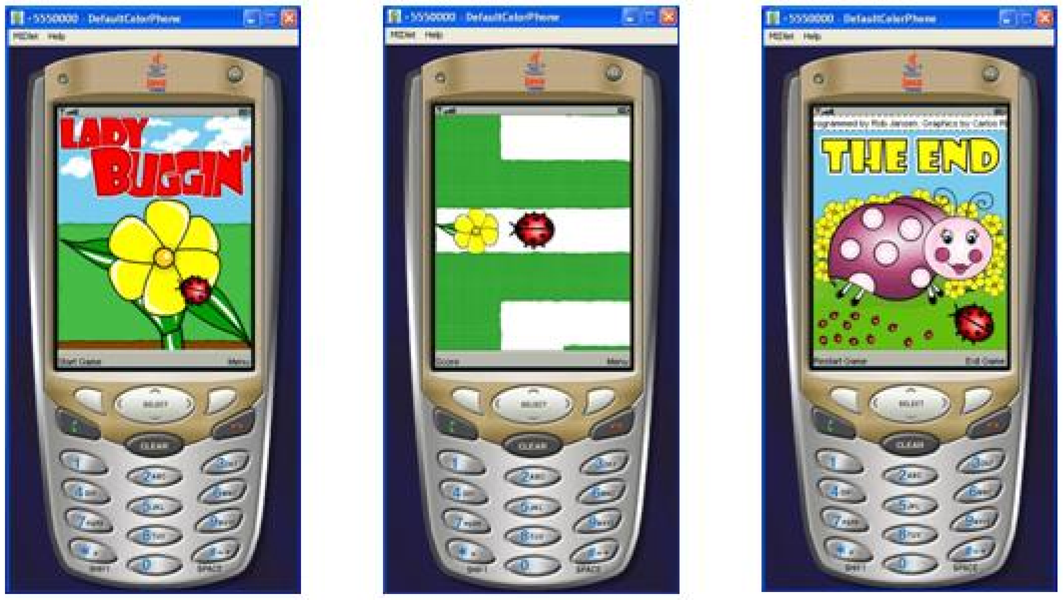

Figure 2- The game being executed on Sun's Wireless Toolkit cell phone emulator. Shown is the starting screen, the game screen, and the ending screen.

The levels are implemented with a two-dimensional array map that represents the spaces the ladybug can and cannot interact with. When the ladybug tries to move to a new space, the array is checked to see if the ladybug made a legal move. If so, the graphics are updated accordingly. The maze graphic had to be created precisely so that everything would fit together. The game was programmed with a subset of Java SE, called Java 2 Micro Edition (J2ME). J2ME is a freely available programming specification and there are several tools available for working with the language. I programmed the game so that it is understandable and simple. There was no specific method I used but my own personal style. The game is still in its early stages, but is suitable as a demo at this point.

5. CONCLUDING REMARKS

By participating in this project, I have come to learn about the general process my advisor uses in his classes. I have formulated some strengths and weaknesses of the process and of the group environment that I was a part of. I have suggested some recommendations for improvements to the process. I have also included some elements that I feel were missing and would have made development go much smoother.

5.1 Strengths

One strength of the process is that everything is laid out for the team. This is useful to undergraduate development teams because it provides them with a guide to what needs to be done. It helps the team with direction and gives them an idea of what to do next. Another strength of the process is that it gets the team to think about every aspect of design. Well this might not always be necessary; it ensures that the team knows exactly what the program will do, how it will look, etc, providing the constructed plan is followed. Another advantage of this process is that once the plan is made, the coding phase can be done relatively individually. A plan has been constructed by the team, with all aspects predetermined. This makes coding just a matter of following and implementing the design into code.

The small group also has strengths over large groups. The small group is very good at communication, and tends to be very flexible. It is very easy to call a meeting with the group, or to just ask questions during the day and get instant feedback or clarification. Different members of the team can help out in different parts of design based on their personal strengths. Each member can contribute to the team to make an complete effort at creating the plan.

5.2 Weaknesses

The process we used also has numerous weaknesses. A lot of time needs to be spent on planning and design. Much of this design could be changed later or the plan could be trashed in turn of a more efficient aspect of the plan. Also, the design doesn't ever get tested until coding begins. If a flaw is found in the plan, major changes will need to be made. This could result in a waste of productive time. This process is very design heavy and that will not always work. Most of the time the team is not sure how to design things until they start to code and get a feel for how things work. They will just be guessing at good design strategies and wasting time on planning things that will never get used. In my situation, most of the design ideas got modified or changed completely during the construction of the game. This affected the way the system interacted with itself and also the functionality the game would provide.

A weakness of the small group environment I was part of is that it did not have a manager. A manager is extremely important to keep team members on task and make sure everyone is contributing to the project. This is particularly important in undergraduate projects where team members have different levels of skill. The project will be forced onto one person with while the rest of the team will not contribute appropriately. A manager is one of the biggest needs that our group had during the project. It causes too much pressure to the team and makes the final product fall below its initial expectations.

5.3 Recommendations

Overall our group spent too much time on design and wasted productive time on steps that never actually got used. For this reason I recommend that small groups use a modified version of extreme programming. Extreme programming is efficient in that it doesn't waste time on design ideas that it will never use. It also keeps things as simple as possible and produces readable accurate code thanks to pair programming. It also forces testing to become part of the construction process, and helps create more complete versions of the program. However, I do not feel that design should be completely cut out of the development process. I feel that having some sort of initial plan that the group agrees on is important for the success of the program and for team unity. The team also needs a manager. This is almost necessary in that the manager will provide the team with focus and organization and will help ensure that the team stays on track. The manager also can modify plans as necessary and help with team communication. Plans are always changing in software development, and an agile process is better able to handle these changes.

In the future it would be desirable to create this mutated process that is specifically suited for small groups. The new process should be tested and implemented to ensure that it will work. The process could be explicitly written out with documentation on what parts of the new process worked and what didn't work. This is the next step of the research.

6. ACKNOWLEDGMENTS

I would like to thank my advisor Dr. John Bowles for help with design and for his continued support throughout the project. I would like to thank my group members who helped develop the game with me: Carlos Rivera, LaShea Johnson, Lalique Gumbs-Prince. I would also like to thank Dr. Caroline Eastman, Dr. Libby Alford, Roxanne Spray, and Tiffany Mintz for their mentorship. I would like to thank the National Science Foundation for funding this project and making this research experience possible for me, and the University of South Carolina for hosting the program.

7. REFERENCES

[1] Pressman, Roger S. *Software Engineering: A Practitioner's Approach 6th Edition*. McGraw-Hill, NY, 2005. pp 45 – 94.

[2] Li, Sing and Knudsen, Jonathan. *Beginning J2ME: From Novice to Professional 3rd edition.* Apress, CA, 2005. pp 1 – 102.

[3] Stein, Michael V. "Using Large vs. Small Group Projects in Capstone and Software Engineering Courses." *Journal of Computing Sciences in Colleges*, 2002, volume 17, issue 4. pp 1 – 6. Date accessed: 6-23-06. url: http://delivery.acm.org/10.1145/780000/774291/p1-stein.pdf?key1=774291&key2=0407951511&coll=ACM&dl=ACM&CFID=12405&CFTOKEN=94402717

[4] Paulk, Mark C. "Using the Software CMM in Small Organizations." *Carnegie Mellon University*, 1998. pp 4-5. Date accessed: 6-26-06. url: http://www.tilysoft.com/SoftwareManage/file%5Cusing_software_Cmm.pdf

[5] Nunes, N. J. and Cunha, J. F. "WISDOM: A Software Engineering Method for Small Software Development Companies." *Software*, IEEE, 2000, volume 17 issue 5. pp 113 – 119. Date accessed: 6-26-06. url: http://ieeexplore.ieee.org/iel5/52/19003/00877877.pdf?isnumber=&arnumber=877877

[6] Wells, Martin J. *J2ME: Game Programming.* Premier Press, MA, 2004. pp 601 – 605.