A life cycle for creating an uncomplicated software

Michael Dorin
mike.dorin@stthomas.edu / Universität Würzburg. Würzburg, Germany

Sergio Montenegro
sergio.montenegro@uni-wuerzburg.de / Universität Würzburg. Würzburg, Germany

Receipt: 1-7-2018 / Acceptance: 20-8-2018

Abstract. Modern software development life cycle models tend to be less formal and less rigid then Waterfall based models of the past. However, creating software without following even the most basic of plans often results in poorly structured, faulty, and hard to maintain software. This paper proposes a practical development model for the purpose for software development. Following some sort of plan produces better code than no plan at all. This model has been shown to be successful even with inexperienced developers.

Keywords: software development, life cycle model, waterfall, agile, RUP, uncomplicated software

Un ciclo de vida para crear un *software* no complicado

Resumen. Los modelos de ciclo de desarrollo de *software* moderno tienden a ser menos formales y menos rígidos que los modelos basados en cascada del pasado. Sin embargo, crear *software* sin seguir un plan básico frecuentemente trae como resultado *software* de pobre estructura, con fallas y difícil de mantener. En este documento se plantea un método práctico con el propósito de desarrollar *software*. Seguir cierto orden en el plan producirá mejor código que no seguir ningún plan. Este modelo ha mostrado ser exitoso incluso con los desarrolladores inexpertos.

Palabras clave: desarrollo de *software*, modelo ciclo de vida, cascada, ágil, RUP, *software* de código legible

1. Introduction

Software complexity is an expensive and important topic in the field of software engineering. Complicated software has been shown to contain more defects than uncomplicated software.

Research demonstrates a 50 percent decline of developer productivity as the software gets more complicated. Staff turnover also increases when complicated software is part of the environment (Sturtevant, MacCormack, Magee, & Baldwin, 2013). Software complexity is said to contribute to 25 percent of the maintenance costs of a software project and 17 percent of the overall cost of the development effort (Banker, Datar, Kemerer, & Zweig, 1993).

One of the causes of complicated software is the absence of planning when a project begins (Foote & Yoder, 1997). In the popular Agile life cycle model, development of software architecture is not specifically mentioned. The Waterfall life cycle model does accommodate architecture de- sign, but in general Waterfall has been considered too overbearing and inflexible for many projects.

Establishing an architecture is not a new concept and there are many established architectural design patterns which can be used as a basis for a system (Larman, 2012). By providing toolkits, Apple and Google have established architectures for designing applications. This has led to millions of applications being created.

The model described in this paper will lead to less complicated software by creation of a design before coding begins. Through requiring some upfront work before a project starts, a team can establish a flexible architecture and reduce software complexity. To verify the efficacy of this model, students at the University of St. Thomas were instructed to design an application using the steps described in this paper.

2. Current Practice in Life Cycle Models

A short review of current life cycle models is helpful for understanding what is presently done and what is missing for current methodologies.

2.1 Waterfall

A survey by TechBeacon shows 9 percent of organizations use Waterfall or lean towards Waterfall (Jeremiah, 2018). When the software crisis was first identified the Waterfall life cycle model emerged as a potential solution and become popular (Schach, 2007). The Waterfall model is characterized by phases which have a well-defined beginning and end.

There are phases for requirements gathering, analysis, design, implementation, and testing. The cornerstone of Waterfall is that developers do not move from one phase to the next until they have completed the current phase (Schach, 2007). Over time, it was determined Waterfall was not ideal for all projects be- cause of inflexibility adapting to changing requirements (Zhang, Hu, Dai, & Li, 2010). However, the strengths of Waterfall should not be overlooked as it is a good fit for projects that cannot be iterative, such as some aerospace or medical devices. When a Waterfall project is done well and by a disciplined team, high quality documentation is produced, and an architecture is well defined (Singhto & Phakdee, 2016).

2.2 Agile

Because of the inflexibility of Waterfall, more modern methods of developing software were desired and Agile was created. In fact, the principles of Agile have become firmly accepted. Ac- cording to TeachBeacon, 67 percent of projects are pure Agile or leaning towards Agile (Jeremiah,2018). Agile is not a software development life cycle model in the sense of giving a checklist of activities to perform, but instead the Agile principles form guidelines for developing software (Beck et al., 2001). It is commonly accepted in Agile that software requirements and architecture evolve through a collaborative effort of self-organizing teams (Williams, 2012).

Scrum has become a very popular form of Agile in recent years and 50 percent of Agile developers use scrum (Alliance, 2015). Scrum is a very iterative and incremental process which encourages each deliverable to be something useful towards accomplishing the desired task.

This approach gives stakeholders an opportunity to see a working product in near real-time and comment on aspects that are deficient. A video introduction to Scrum suggested a model where if you are building a car, your first two-week sprint may deliver a skateboard, then a scooter, then a bike, then a motorcycle, then a car. (Henrikkniberg, 2012) In principal, this is a good idea, however, in many cases the architecture that evolves is not sufficient for the task and an overly complex system emerges. For example, though a skateboard is a mode of transportation, a skateboard will not provide a sufficient foundation for a car.

Agile more defines the personality of the team than the process by which the work is done and omits a mandate for doing analysis and design. Other Agile disciplines such as Scrum give more detail on the process but are still missing definition of how work flows should actually happen. In fact, Agile disciplines can be thought to tacitly discourage documentation through putting more emphasis on people rather than processes (Nerur, Mahapatra, & Mangalaraj, 2005).

2.3 Rational Unified Process (RUP)

The Rational Unified Process (RUP) is considered by some to be a hybrid of Waterfall and Agile. It is iterative and incremental (Agile) but defines phases with specific deliverables. The Rational Unified Process has four phases, Inception, Elaboration, Construction, and Transition. Activities in the four phases are made up of the work flows. In the Inception phase, the project is launched, and heavy emphasis is place on requirements gathering. The emphasis moves to analysis and design in the Elaboration phase and programming during the Construction phase. The project finishes up with a Transition phase where documentation is finished, and final acceptance testing is completed. RUP allows the defined work flows and their associated activities to be conducted at any time during the project regardless of the current phase. It has been accused of being too rigid and indeed Craig Larman, an advocate of RUP, warns engineers to be careful of rigidity in RUP projects. (Larman, Kruchten, & Bittner, 2001).

3. Proposed Life cycle Model

Less complicated software is easier to review (Dorin, 2018). Additionally, code reviews and inspections are the least expensive way to detect faults (Basili & Selby, 1987). What is proposed here is to combine positive aspects of Waterfall and RUP to create a new Agile Software Development Life Cycle model which encourages implementation of less complicated code.

Antoine de Saint-Exupéry said, "A goal without a plan is just a wish." As the current state- of-the-art methodologies do not suggest a plan for designing an architecture, it often seems good architecture is often just wishful thinking. This Life Cycle Model is broken up into sections, but it is essential to keep in mind that since this is process is iterative and incremental, each of the parts should be further managed using time-boxed intervals, known as sprints, as done in Scrum.

3.1 Part 1. Kickoff

Each project will begin with a kickoff, which will make up the first sprint of the project. It is recommended the kickoff does not exceed one sprint, as taking longer may push the project towards Waterfall. A kickoff document needs to be created and approved as the project starts and should contain the content described below.

3.1.1 Vision Paragraph

It is important to be able to describe what the product is going to do. This should be possible to do in one or two paragraphs. The vision paragraph should quickly describe what the application is going to do and the benefits.

3.1.2 Boundaries Paragraph

The boundaries paragraph is equally as important as the vision paragraph. The boundaries paragraph needs to describe where development will stop. For example, the boundaries paragraph may say that “the first version will not include this specific feature.” Or it may say, the first version will only be written for Android devices and will not run on an Apple device.

3.1.3 Requirements

Requirements should be very customer centric and should be written from the perspective of user stories. User stories should be in the form of: “As a [type of user], I want [some goal] so I can [some reason] (Cohn, 2018). User stories need not be more complicated than that. Creating user stories should be a social activity which generates as many user stories as can be thought of. It is important that you do not omit the “so I can” clause as this is often important later to determine the context of the user story. As user stories are written, assign points to them based on their complexity. These points will be necessary for generating an initial cost estimate for development.

3.1.4 Business Case and Initial Cost Estimate

A large collection of completed user stories will be needed, but it is then possible to get a rough idea of what the project is going to cost. There are many ways of doing estimations, but the easiest way seems to be assigning points to each user story and creating a multiplier based on how much time or money each point means. For example, if the points in your user stories total to 10,000, and it is estimated it to take three hours of development time per point, you can roughly estimate the project will take 30,000 hours of development time.

It is not the purpose of this document to describe how a business plan should be modeled, but minimally the software needs to pay for itself somehow. If it is believed that writing the software will save money for an organization, this should be documented here. If it is believed that the software will make a good product, this should be documented and supported here as well. Business risks, such as marketability of a software product should also be included here.

It is important to have sufficient background information to be able to make a buy or build decision, and this decision should be made before leaving the kickoff phase.

3.1.5 Technical Risks

At the beginning of a project technical risks should be addressed. For example, if a team has never worked with a specific hardware device, it should be recorded so a plan can be made on how to deal with it. If a team has never programmed on Android before and the project is an Android application, this is a risk that should be included and managed. Risks such as turnover of engineers or popularity of the project do not belong here as these are business risks and belong in the business case.

3.1.6 Nonfunctional Requirements

Non-functional requirements are how a system is judged when complete. If a system is to be secure, specify a non-functional requirement indicating that. Non-functional requirements can certainly inspire several functional requirements. Packaging requirements, legal requirements, and ease of use requirements are also examples of non-functional requirements.

3.1.7 Glossary

A glossary is very important. Domain specific terms need to be listed and described here. This is not a clearinghouse for obvious words, but if a group is doing software in a new field, a central location for terminology needs to be created so team members are not constantly struggling to understand terminology. For example, if a project is in anesthesiology, important anesthesiology terms need to be included in the glossary.

3.2 Part 2. Analysis and Design

Once the kickoff document is complete, analysis and design can begin. Note, that as the breakdown happens, it is highly possible that new requirements are discovered. This entire model is Agile, and it is possible for new items to be added and existing items changed during the project. The analysis and design phase should be carried out in about two sprints. The sprint limit once again, is to discourage accidental shifting to a Waterfall life cycle. These sprints can be managed as best for an organization, but it is suggested the first sprint deals with risk mitigation, completion of the user stories, and the rehearsal. The second sprint should review the results of the first sprint and create the UML architecture diagrams from the rehearsal.

3.2.1 Risk Mitigation

At this point a plan is needed to mitigate all the risks listed in the kickoff document. For example, it may be beneficial to do a proof-of-concept prototype demonstrating the capacity to handle a technical risk. It may also be that additional training is scheduled or that outside consults are brought in. The important thing is that the risks are mitigated and do not become "brick walls" as the project progresses.

3.2.2 Rehearsal

For more information on this topic, see the companion paper on simplifying the design workflow. Many military organizations perform a rehearsal walk through to make sure parties understand what is being planned. This is also an important tool for software development.

During the rehearsal expected actions are practiced improving performance during execution. This rehearsal is important as it allow participants to become familiar with the operation. The entire plan should be rehearsed by the stakeholders. A team needs to be able to rehearse how the system as going to work. If that is not possible, it is very likely there is not sufficient understanding of the problem for a proper implementation.

A natural architectural design pattern for this analysis method is Model-View-Controller (MVC). Within MVC, the model manages the application data while the view provides the output of information to users. The controller is in charge of operations, it accepts input and sends commands to the model and the view (Rosenbloom, 2018).

Before a rehearsal begins, participants select actors to deal with interactions with users, actors to deal with storing and updating of data, and finally an actor to be a leader. This staging describes the MVC architectural pattern. The MVC pattern nicely separates control from user interactions and maintenance and updating of data.

3.2.3 UML Sequence and Class Diagrams (Architecture Definition)

The Unified Modeling Language (UML) defines a standard set of diagrams helpful for creating a system (Larman et al., 2001). UML sequence diagrams can be drawn up visually describing the actions settled on by the rehearsal. It is not necessary that all aspects of UML are perfect in this diagram, it is more important that a flow is shown. An example of a sequence diagram is shown in figure 1.

In the Unified Modeling Language, class diagrams show the relations and dependencies among classes. When the sequence diagram is complete, it is very easy to create a class diagram in which methods are apparent and the overall structure of the architecture emerges. An example of a class diagram is shown in figure 2.

Based on the flow of events observed in the rehearsal, a UML sequence diagram should be created. When satisfied with the sequence diagram, a class diagram should be created which will provide an initial candidate architecture for the project.

It has been shown that creating UML diagrams has a positive effect on defect density (Nugroho & Chaudron, 2009) and program correctness (Hurme et al., 2011).

3.3 Part 3. Implementation

At this point the coding can begin in earnest as the architecture is not just wishful thinking. Implementation can follow any development model a team is comfortable with. If a team is doing a form of Kanban, Post-it notes can be put up and developers can select and implement the feature described. With the architecture defined coding can follow an established path.

Other required documentation such as users' guides should be written and completed during implementation. Soft- ware testing should be ongoing during implementation. At the end of implementation there may be some final acceptance tests to be run, but there should be no surprises as testing is an ongoing work-flow. Implementation should be carried out over many sprints and, as once again since this is an Agile process, analysis and design activities are also performed as necessary.

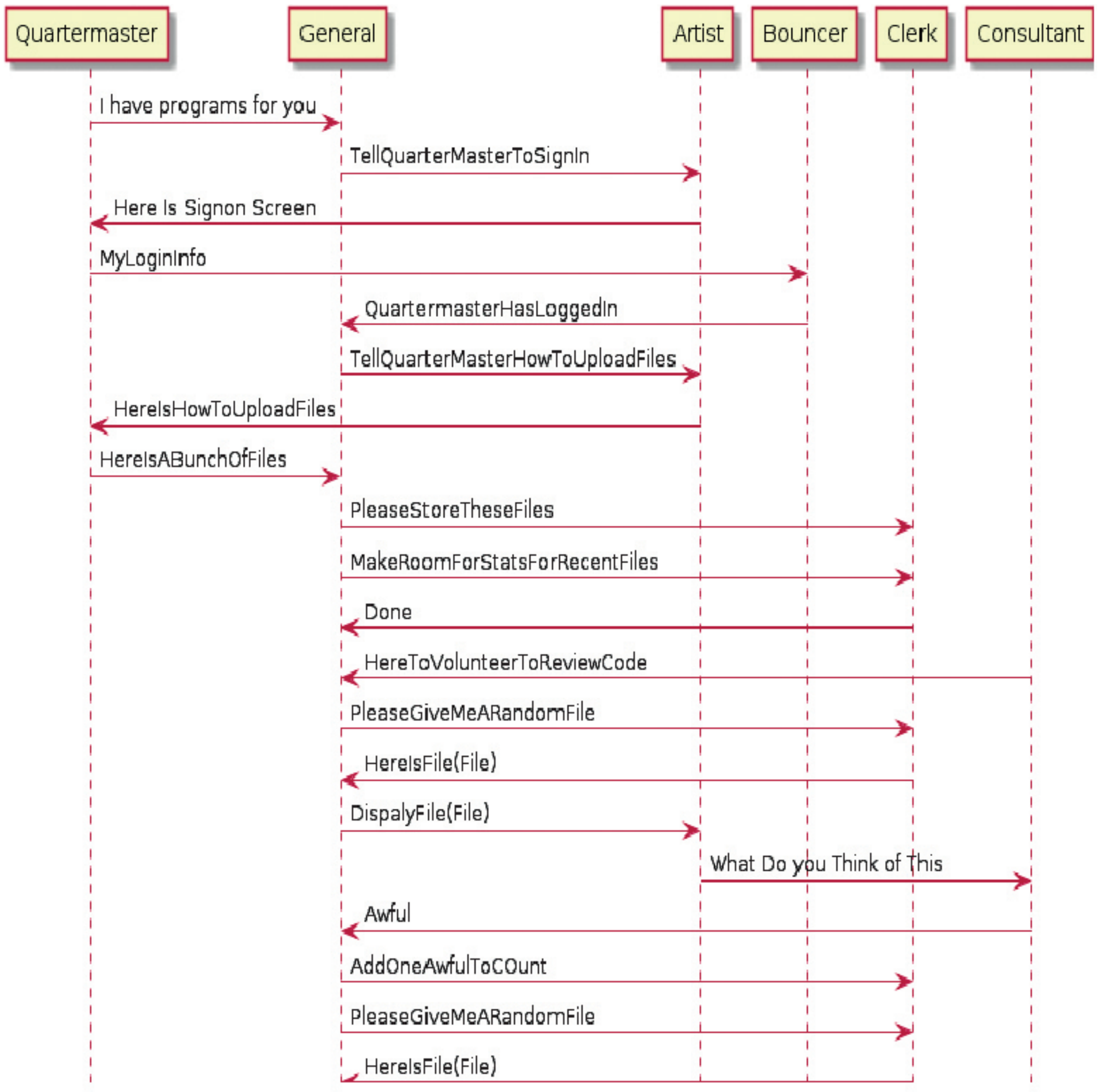

Figure 1. Sample Sequence Diagram
Source: Dorin, 2018

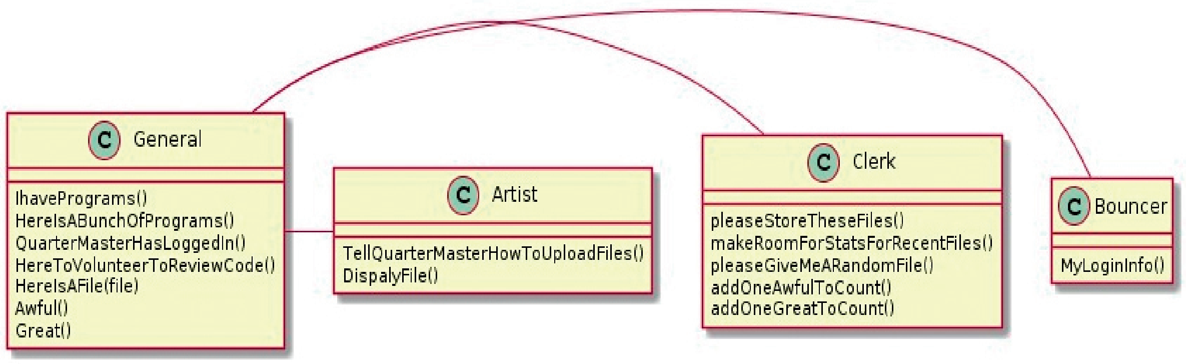

Figure 2. Sample Class Diagram
Source: Dorin, 2018

5. Results

The Graduate Programs in Software department (GPS) of the University of St. Thomas (UST) provides an avenue for people without a technical background to study for a career in information technology. Students arrive with different backgrounds including teachers, human resources employees, and even history majors. One course all students must complete is SEIS-610, Introduction to Software Engineering. This class exposes students to various aspects of software engineering including the various life cycle models and other topics such as budgeting and scheduling. As part of this semester-long class, students are required to design an application to gain experience with various aspects of the software development life cycle. Often this class was taught with a pure Rational Unified Process (RUP) life cycle model. Many students were confused by aspects of RUP, such as how it called out the definition of non-functional requirements and concepts such as vision and scope. The requirements work flow was also more complicated as RUP calls for using use cases instead of user stories. In many projects, the non-functional requirements were contrived as students felt they needed to have each RUP check box completed. Though most all of the student projects were successful, they spent unproductive time figuring out little frustrating aspects of RUP.

When hybrid concepts were introduced, most groups were able to produce a successful project with far less frustration. This was demonstrated by fewer calls and emails with questions. The hybrid model using user stories allowed for the faster definition of requirements. By simplifying the description of non-functional requirements, the hybrid model eliminated invented unneeded items. Many students had no software background before beginning the class and were still able to design very sophisticated systems.

5. Case Study

In one example, a group of students formed a team which who's goal was to design an application which would manage satellite communications. The system they intended to manage was a private project created by a small startup business. The goal of the system was to perform both message- based communications as well as voice communications. Custom devices which connect via USB to Android cell phones are to be marketed which would interface directly with the satellites. The startup company faced problems managing the mobile devices which needed to enable, billing needed to be activated, and other aspects of the communication paths needed to be configured. This group had the same amount of time to complete the project as other groups, which was about 14 weeks. The group had both experienced and inexperienced people when it came to software engineering.

The first required deliverable was the "Kickoff" document. For this group this document pro- vided a very clean communication mechanism between the students and the outside organization. Especially helpful were the "Boundaries" and "Technical Risks" paragraphs. It seemed all too easy to add features during verbal discussions, and the boundaries paragraph helped manage adding new features. The "Technical Risks" paragraph helped identify areas that the students would truly have to research if they were to continue on the project.

The second required project deliverable was to prepare class and sequence diagrams, and the result of their work was a substandard submission. Initially, this group had decided not to do a rehearsal and even classified the approach as absurd. When the group was interviewed regarding their project, they were unable to describe interactions nor an accurate structure of their system adequately. Finally convinced to do a rehearsal, within two weeks they had appropriate class and sequence diagrams and expressed surprise with the effectiveness of the approach. Without rehearsing the operations, the system was just too complicated for the group to design. Actual implementation was not required for this project; however, it is evident that had real coding begun before the rehearsal the code produced would have been of haphazard design and too complicated.

6. Conclusion

The hybrid lifecycle model proposed in this paper is Agile and is compatible with the four tenants of the Agile Manifesto which emphasize individuals, working software, customer collaboration, and response to change (Beck et al., 2001). Through the use of user stories instead of more formal use-cases, individuals and customer collaboration are encouraged. Through a practical architecture design process, working software is championed.

Outside the university setting, hybrid models have been shown to be successful in many areas. In the paper, "The impact of process maturity on defect density," it was discovered that hybrid life cycle models had a lower defect density than Waterfall or Scrum (Shah, Morisio, & Torchiano, 2012). Some students working professionally in the software industry indicated their employers appreciate the "Kickoff" document for launching new projects.

This paper has also covered aspects of complicated software and issues have been shown. The benefits of various traditional lifecycle models have been explained and some criticisms enumerated. The hybrid life cycle model described was successfully presented to graduate students at the University of St. Thomas. This model encourages architecture definition while still allowing for the communicative strengths of Agile. Further research is necessary to measure how well this lifecycle model will work for professional development organizations.

References

Alliance, S. (2015, July). The 2015 state of scrum, scrum-alliance-state-of-scrum-2015.pdf. Retrieved from https://www.scrumalliance.org

Banker, R. D., Datar, S. M., Kemerer, C. F., & Zweig, D. (1993, November). Software complexity and maintenance costs. *Commun. ACM*, 36(11), 81–94. Retrieved from http://doi.acm.org.ezproxy.stthomas.edu/10.1145/163359.163375 doi:10.1145/163359.163375

Basili, V. R., & Selby, R. W. (1987). Comparing the effectiveness of software testing strategies. *IEEE transactions on software engineering* (12), 1278–1296.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., . . . others (2001). Manifesto for agile software development.

Cohn, M. (2018). Scrum certification via certified scrummaster training. Retrieved from https://www.mountaingoatsoftware.com/training/courses/certified-scrummaster

Dorin, M. A. (2018). *Coding for inspections and reviews*. New York, NY, USA: Association for Computing Machinery.

Foote, B., & Yoder, J. (1997). Big ball of mud. *Pattern languages of program design*, 4, 654–692.

Henrikkniberg. (2012, Oct). Agile product ownership in a nutshell. YouTube. Retrieved from https://www.youtube.com/watch?v=502ILHjX9EE

Hurme, J., et al. (2011). The benefits of using uml-modeling tools in evaluation and testing of etm software.

Jeremiah, J. (2018, May). Agile vs. waterfall: Survey shows agile is now the norm. TechBeacon. Retrieved from https://techbeacon.com/survey-agile-new-norm

Larman, C. (2012). *Applying uml and patterns: An introduction to object oriented analysis and design and interative development*. Pearson Education India.

Larman, C., Kruchten, P., & Bittner, K. (2001). How to fail with the rational unified process: Seven steps to pain and suffering. Valtech Technologies & Rational Software.

Nerur, S., Mahapatra, R., & Mangalaraj, G. (2005). Challenges of migrating to agile methodologies. *Communications of the ACM*, 48(5), 72–78.

Nugroho, A., & Chaudron, M. R. (2009). Evaluating the impact of uml modeling on software quality: An industrial case study. In I*nternational conference on model driven engineering languages and systems* (pp. 181–195).

Rosenbloom, A. (2018). A simple mvc framework for web development courses.In *Proceedings of the 23rd western canadian conference on computing education* (pp. 13:1–13:3). New York, NY, USA: ACM. Retrieved from http://doi.acm.org.ezproxy.stthomas.edu/10.1145/3209635.3209637 doi: 10.1145/3209635.3209637

Schach, S. R. (2007). *Object-oriented and classical software engineering* (Vol. 6). McGraw-Hill New York.

Shah, S. M. A., Morisio, M., & Torchiano, M. (2012). The impact of process maturity on defect density. In *Proceedings of the acm-ieee international symposium on empirical software engineering and measurement* (pp. 315–318). New York, NY, USA: ACM. Retrieved from http://doi.acm.org.ezproxy.stthomas.edu/10.1145/2372251.2372308 doi: 10.1145/2372251.2372308

Singhto, W., & Phakdee, N. (2016, Dec). Adopting a combination of scrum and waterfall methodologies in developing tailor-made saas products for thai service and manufacturing smes. In *2016 international computer science and engineering conference* (icsec) (p. 1-6). doi: 10.1109/ICSEC.2016.7859882

Sturtevant, D., MacCormack, A., Magee, C., & Baldwin, C. (2013). *Technical debt in large systems: Understanding the cost of software complexity*. Unpublished thesis, MIT.

Williams, L. (2012). What agile teams think of agile principles. *Communications of the ACM*, 55(4), 71–76.

Zhang, X., Hu, T., Dai, H., & Li, X. (2010). Software development methodologies, trends, and implications. *Information Technology Journal*, 9(8), 1747–1753.