Software Engineering Methodologies and Life

Scott Lennon

Columbia University
New York, USA
sl3796@columbia.edu

```
COMSE6156
    {
      sl3796;
    }
```

***Abstract* ~ The paradigms of design patterns and software engineering methodologies are methods that apply to areas outside the software space. As a business owner and student, I implement many software principles daily in both my work and personal life. After experiencing the power of Agile methodologies outside the scope of software engineering, I always think about how I can integrate the computer science skills that I am learning at Columbia in my life. For my study, I seek to learn about other software engineering development processes that can be useful in life. I theorize that if a model such as Agile can provide me with useful tools, then a model that the government and most of the world trusts should have paradigms I can learn with as well. The software model I will study is open source software (*OSS*). My research examines the lateral software standards of (OSS) and closed source software (*CSS*). For the scope of this paper, I will focus on research primarily on Linux as the OSS model and Agile as the CSS model. OSS has had an extraordinary impact on the software revolution [1], and CSS models have gained such popularity that it's paradigms extend far beyond the software engineering space. Before delving into research, I thought the methodologies of OSS and CSS would be radically different. My study shall describe the similarities that exist between these two methodologies. In the process of my research, I was able to implement the values and paradigms that define the OSS development model to work more productively in my business. *Software engineering core values and models can be used as a tool to improve our lives*.**

I. Introduction

Robert Love's description of the Linux kernel development process is contradictory to my findings. Love describes the OSS development process as "chaos [2]." I will posit the OSS development process is extremely well ordered and strictly disciplined.

One of the goals of my study is to describe to the reader the essential paradigms, methodologies, and practices that define OSS and CSS processes. The analysis will compare CSS and OSS core practices to establish their similarities and differences to the reader. I shall discuss findings with an emphasis on which methodologies result in better code quality. We will also focus on research that examines code to determine which model produces code with less security vulnerabilities, and is cohesive to industry compliance standards. My conclusions shall draw a direct nexus for the reader between models within the software engineering paradigm and their use outside of the scope of software development. I will underscore a variety of themes, including personal experience and a case study measuring effectiveness of paradigm implementation. I will present studies using Agile methods for child rearing [3] to enable the reader to see possible correlations of software engineering methodologies as it can apply to challenges in life. I have used myself as an ongoing case study, a study which undoubtedly will last far longer than the Spring semester of 2016. I will introduce the findings of my case study and urge the reader to experiment themselves with the ideas defined.

A. Audience and Plan

This paper is for anyone interested in learning about how software engineering paradigms work, and for individuals that want to know how they can be implemented conceptually in other areas of life. My goal is to explain and compare two models used in software development, and show you through both research and case studies that following these practices can make you a better developer, manager, husband, wife, mom, dad, soccer player, or whatever role you wish to apply the methodologies to.

First, I will introduce some of the basic definitions and concepts of both paradigms to the reader in the context of the concepts discussed. I assume only a basic familiarity of computer science concepts and will explain the topics succinctly, and strongly encourage the reader to pursue additional study on any relevent background they deem to be interesting. Just a basic conceptual understanding and a will to think outside the box are needed if you wish to implement programming paradigms in your life. After background information is defined, I will provide an overview of some of the research that has been conducted that compare the code quality and security between enterprise and OSS models. We will look into a detailed comparison of the values and paradigms of OSS and CSS models, and briefly discuss some of the tools and design patterns present on OSS. I will also share the findings of my personal case studies, with the goal to illustrate that both CSS and OSS possess methodologies that the reader can implement.

B. Contradiction

The Cathedral and the Bazaar credits OSS with the innovation of the software industry and asserts that proprietary software inhibits permutation and growth[1]. This fact preempted my interest in learning more about the development process of OSS. Contrary to what I expected, OSS design paradigms **are** well defined and structured. OSS values encourage collaboration and learning through others, learning through users, and listening to others. There are formal policies and procedures to ensure code quality and continuous integration. The systems and rules in place are detailed and ensure that its users are methodical in their understanding of developing software. The practices that have come to define OSS have shown themselves to be anything but the "chaos" that Robert Love depicts. A signal of how important OSS is in our universe can be discerned by reading our government's opinion on this concept. The White House hosts a blog and has taken a formal policy to support OSS. If nothing else, I hope the reader will gain a better understanding of why the most powerful world entity would take such a position on OSS. It motivates me to learn as much as I can about the OSS phenomena.

II. DEFINITIONS

A. Background

A CSS model is a set of software processes where the enterprise does not release the actual source code to the public. The internal team that produces the software maintains their product. A company ships proprietary code in a compiled executable state and the source code is secret to the user. A company can patent and enforce intellectual property rights to ensure that their secret recipe cannot be used by anyone else. This concept is present in many industries and can be referred to as trade secrets. An example of an CSS operating system is Microsoft's Windows. Additional examples of CSS software include the popular web browsers Internet Explorer and Safari.

An open source community develops OSS in a manner that allows users to view and modify a project's source code. The source code is open to the world to use, change, and fix to meet the needs of its users. Instead of the design of a project being done within the confines of a private company, an infinite number of sources can collaborate in the OSS model. OSS is freely available and distributes under different licenses. The glaring difference between OSS and CSS is code visibility. Space, patents, enterprise, geography, logistics, and intellectual property rights do not confine the development of OSS. Participation in OSS projects is often voluntary to any and all users [4]. We will compare the activities of software development as well as the values and paradigms of both models. An example of an OSS web browser is Mozilla, and an example of an OSS operating system is Linux.

B. OSS (Linux)

OSS development is defined analogously as, "a great babbling bazaar of differing agendas and approaches out of which a coherent and stable system could seemingly emerge only by a succession of miracles [1]." The OSS development model that we will focus on for research is Linux. In 1991, a student at the University of Helsinki created the Linux Operating System. Linus Torvalds wished for a free operating system that he could modify, so in the absence of what he wanted he developed his own project to satisfy his needs [2]. Linux grew exponentially due to its collaborative nature and today has thousands of applications that implement a version of the Linux kernel. Linux runs on systems and devices that we use every day.

C. Design

Linux development consists of a series of phases where developers work on new features for users until a feature freeze is declared [2]. Although the process is evolutionary according to the needs of the customer, there are clear development steps in the process. The stages of patch integration are design, early review, a comprehensive broad review, constant peer analysis, and finally, and rarely, integration into the kernel.

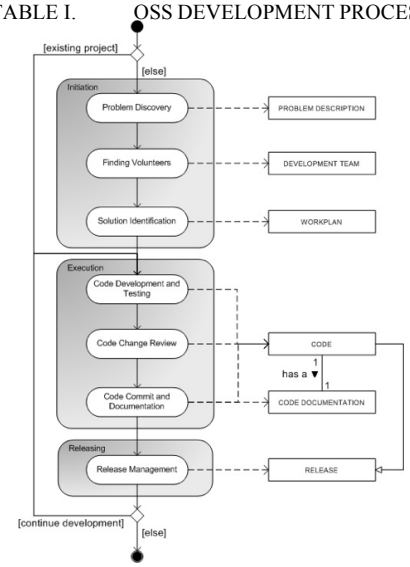
The documentation of requirements occurs during the design phase. Documentation will describe implementation details of the requirements. The purpose of putting the design in writing is to describe what the project is attempting to accomplish and the plan the project will implement to deliver the software. Initial documentation will identify the actors and the problem that needs to be solved. If applicable, documentation can contain legal, regulatory information, and anything else within the scope of the context of your project. The written design is a significant part of the development process, and proper documentation and review will help determine the plan of action when the time comes for actual coding. When implemented correctly a project's documentation can save time, energy, and money later down the development pipe. In the Linux documentation design phase, it is encouraged that the execution of this step involves the community.

D. Review, Review, and More Review

The first early review stage requires posting the design documentation to an email distribution list. During this first review, peer developers that participate in a community subscribe to email distribution groups unique to specific features. The members will give feedback on the initial design through the mailing list. At this stage, the design is checked to make sure a current design pattern has not already solved the problem. Peer developers evaluate and comment on the design. This initial review enables the community to uncover any design issues that demonstrate bad practices or may lead to further bugs in the code. If the documentation does not specify the project's goals and define the solution in a clear and simple way, the development process will stall until sufficient specifications are met to support the project.

Subsequently, the next development stage is a broad analysis. The Linux kernel is managed by subsystem maintainers that manage branches of the project. A Linux kernel maintainer performs a rigorous design and implementation study at this stage. After the formal review is further extensive community scrutiny while integrating the feature with other work. The development procedure methodically continues with iterations of cycles of debugging and testing. If the community and maintainers approve the feature, and no new bugs arise during integration, the last iteration culminates in a stable release which will include the patch [5]. The development phases are graphically depicted in Figure 1 for the reader.

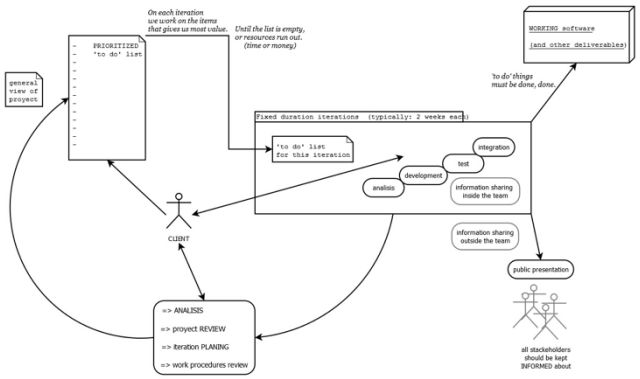
TABLE I. OSS DEVELOPMENT PROCESS



[a] By M. Abbing - Own work (Original text: self-made), CC BY-SA 2.5

Fig. 1. Example of OSS development process

TABLE II. AGILE METHODOLOGY



[b] By Benzirpi (Own work) [CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0)]

Fig. 2. Example of Agile Methodology

E. CSS (Agile)

CSS follows an iterative model such as Agile, which adheres to the fundamental principles of communication, adjustment, and perception. Agile describes paradigms and methodologies for iterative software engineering. Agile paradigms invoke beliefs that emphasize teamwork, synergy, and collaboration. The model stresses continuous integration, plan devising, and rigorous testing.

The initial phase in Agile consists of documenting requirements. Agile requires actor and stakeholder involvement in creating user stories and use cases. The customer should be involved in determining the project's requirements. Each iteration cycle requires user story creation, adjustment of tasks, team planning and design, pair programming, and repeated testing. Every iteration includes building working software that the customer can use to provide feedback on the project as often as possible. Regular team reflections and behavioral adjustments on how to become more efficient are important collaborative Agile properties [11]. I learned Agile methodologies in Advanced Software Engineering and began experimenting with Agile paradigms. It was at this time I was able to connect programming principles to problems I was trying to solve at work. The core values of Agile are very intuitive once abstracted from software engineering. Teamwork, reviews, testing, collaboration, continuous customer feedback, adapting to new issues, tracking progress, and empowering team members are all intuitve practices.

Figure 2 is a graphical diagram of the Agile development model to help the reader visualize the iteritive nature of Agile programming.

III. RESEARCH

The 2015 Future of Open Source Survey is an analysis conducted by Black Duck Software. The study found that the number of companies using OSS is continuously increasing [7]. The findings reveal that 78 percent of businesses run their operations on OSS.

A. Defects

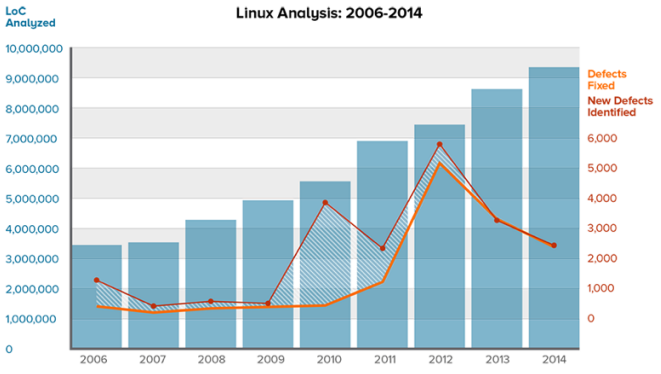
78 percent of companies use open source and I analyzed research and studies to find out why. To determine which software engineering paradigm produces the best quality and most secure code we dissect some of the research. One analysis of the code quality of four operating systems kernels used metrics to study the code quality comparing OSS to CSS and found no differences [8]. The study focused on FreeBSD, Linux, Solaris, and the Windows operating system kernels. However, subsequent analysis and reports by Coverity that tested ten billion lines of OSS and CSS source code found contradictory results. Coverity is a company instantiated by the United States Department of Homeland Security [10]. Coverity Scan, which is now managed by Synopsis, is the largest research project in the world focusing on OSS quality and safety [9]. Synopsis provides free scan analysis to the OSS community as well as CSS enterprises. The Coverity and Synopsis tools aid developers by scanning and testing for code quality and checking for security vulnerabilities. Coverity and Synopsis are industry standard services for calculating the state of software quality as well as security vulnerabilities [9].

The Coverity studies that span from 2006 through 2015 found that "Linux (OSS) remains the benchmark for quality" [10] and concludes that OSS consistently has a lower defect density than CSS. In The Cathedral and the Bazaar, Eric Raymond postulated fifteen years before the Coverity Report an explanation for this consistent outcome. Raymond theorizes that "given enough eyeballs, all bugs are shallow [1]." The greater and wider the review, the better code. The continuous feedback cycle that OSS perpetuates correlates to the defect density

analysis. Fifteen years after Raymond's hypothesis, Linux continues to maintain this focus. Collaborative review puts an exponential amount of eyeballs on the code. Implementation of additional testing requires security audits to catch even the bugs that the reviewer's eyeballs miss.

Figure 3 depicts an analysis of defects that the Coverity scan performed.

TABLE III. LINUX ANALYSIS: 2006-2015



To help ensure highly accurate static analysis results in the Scan service, the Linux team leverages Coverity Scan modeling capabilities to help the analysis algorithms better understand the patterns and behavior of the Linux code. The analysis automatically builds models based on the source code, but it can't always correctly infer what happens—perhaps there is no source code, like in the case of a dynamic library, or there are external effects that cannot be predicted, such as a remote procedure call.

[c.] Coverity and Synopsis Scan [10]

Fig. 3. Linux Defect Analysis

B. *Security*

An interesting finding of the most recent 2014 Coverity study found that although OSS results in lower defect density and higher code quality, CSS is more compliant and secure [9]. Coverity tests compliance and security by scanning the code for The Open Web Application Security Project (OWAP) top ten security vulnerabilities. Coverity theorizes that because OSS is becoming more feature-rich. People need software to do certain things which drive OSS, so adding features is more important than bug fixing during the development cycle. Conversely, competition and compliance drive CSS, which results in security taking the highest precedence [9].

By analyzing results over eight years, the Coverity Scan finds that both OSS and closed source are continually improving. Case studies on Linux discovered that by focusing on new bugs, the bugs were easier to fix. As code gets older bugs get harder to fix [9]. This data is represented in Figure 3 for the reader.

Considering the research, it is not clearly evident why companies choose OSS over CSS. Yes, code quality and feature development speed are paramount, but in my experience, compliance and security are above everything. Perhaps this gap in logic can be explained by further findings in the Black Duck survey analyzing how these companies that choose OSS manage their OSS components.

The Black Duck survey indicates that more than 55 percent of enterprises who use OSS said their business lacks a formal policy or procedure for OSS use [7]. The Black Duck survey also found that less than 16 percent of the enterprise using OSS use automated testing tools [7]. Over 50 percent of the companies using OSS are dissatisfied with their understanding of security vulnerabilities and even less plan to monitor OSS for cyber security. The Black Duck results show that enterprises that use OSS need to implement formal and documented policies and procedures, and perhaps unknowingly, choose development speed and implementation of new features over security and compliance.

It is evident both models have strengths and weaknesses. A method leveraging the strengths of both methodologies will yield the best results. An optimal paradigm needs to produce harmony between security and development speed. With a combination of values and models in OSS, software tools, compliance standards adhered to in CSS, along adherence to software engineering life-cycle principles will produce not just quality code, but also compliance and secure software.

IV. COMPARING PARADIGMS AND PRINCIPLES

I analyze a few fundamental principles taken from the Agile Manifesto [11] and paradigms from OSS best practices documentation to compare both models. Both models are similar in many aspects and share many of the same core values. For instance, both CSS and OSS models share the value that delivering working code should frequently for feedback. A core practice of all software engineering is to release early and release often. Shipping working software continuously illustrates the idea that delivering working software is optimal for implementing changes based on user feedback. For design comparisons, we need to dig a little deeper into the processes.

A. *Role of the Customer*

In Agile, the stakeholder plays an integral role in the feature requirements and the creation of user stories or use cases. User stories are succinct statements to describe a software feature, and identify the type of user, what they want, and why they want it. In OSS, there are documented requirements for the design development resulting in a similar process when drafting initial requirements. The Linux Foundation provides recommended questions for the developer to ask to establish the initial plan. When planning a kernel development project, the developer shall assess the project by asking; "What, exactly, is the problem which needs to be solved? Who are the users affected by this issue? Which use cases should the solution address? How does the kernel fall short in addressing that problem now?" [5] Both methodologies meticulously revolve around solving a solution for a user from the user's perspective. Agile rigorously supports a change in requirements throughout the iterative process and requirements may, and, in fact, should, evolve over the project cycle. Initial requirements are often vague and incomplete, but continuous customer feedback encourages a constant evolution of the needs during each iteration [11]. The OSS cycle starts with precise user requirements, but like Agile, new features, and requirement development can be implemented continuously. Contrast that exists when comparing the models is that in OSS, the user can be the developer, and the community or system

maintainers decide on feature implementation (as opposed to the customer).

B. Motivation and Teamwork

Motivation is an essential valued in OSS and CSS. Building software in a culture of teamwork and motivation is an important principle in the Agile manifesto. Given motivation and an empowering culture, the developers will get the job done [11]. Looking at this CSS principle, we can see both models share this value. OSS is motivation-driven. OSS developers can choose to work on a project that motivates them. If a developer has no motivation to participate in a project they can choose not to engage. Because OSS is voluntary, interest in the project is usually guaranteed. The Agile manifesto proclaims that the best ideas develop from teams [11]. Teamwork and peer cooperation are paramount to the design process in OSS, demonstrating the similarity of this core principle.

Another critical aspect of the Agile development cycle involves face-to-face communication within the team and customer [11]. The OSS model differs when comparing this particular practice such that there is less physical peer communication because developers live across the world. However, there is greater written communication via exponentially growing community email distribution lists. Linux and many other OSS communities require continuous communication and review over global communication tools such as a mailing list. So while actual physical face-to-face interactions may lack in OSS, the peer involvement aspect remains the same across paradigms.

The Agile Manifesto proclaims continuous software builds measures the pace of a project. Another principle stresses attention to simplistic detail, and meticulous design procedures make projects agile [11]. Both methodologies predicate simplicity and great design. Andrew Morton depicts the most important task for a Linux developer. "Make sure that the kernel runs perfectly at all times on all machines which you can lay your hands on [5]." Both OSS and CSS place critical value that software should always work the way it is supposed to. Both models promote tracking development progress via a board and burn-down chart so that the team can adapt and adjust planning to maintain a constant and infinite pace [11].

We look at the Agile principle that encourages that the team reflects together and change their behavior to become more efficient [11] as our last principle comparison. OSS communities participate in self-evaluation and adapt over time as well. Linux and Apache both started with a primitive organizational architecture, and evolved to include foundations with employees, layers of maintainers, management, and executives [12].

V. TESTING TOOLS AND SECURITY

A. Testing

In software engineering, there are testing and code refactoring tools to identify smelly code. A code smell is a flaw or issue in the design or code that can lead to software bugs. Both CSS and OSS continuously use these tools as part of each iteration of the development cycle. The tools employed in OSS test source code statically and dynamically. Linux uses a "lockdep" which dynamically measures dependencies among states [13]. OSS developers working on the Linux kernel use a debugging practice that "poisons" empty chunks of memory so unauthorized access leading to segmentation faults and buffer security issues will be mitigated [14].

The Best Practices Criteria for Free and Open Source Software (FLOSS) created documentation of requirements for developers to follow as quality guidelines. OSS developers use issue trackers such as Jira or Git to track all bugs and changes. Also, Linux requires that issue tracking is documented and updated on the specified email list. OSS developers need to test 100% of the project's source code iteratively using an automated test suite [15]. FLOSS also requires continuous project testing with a static and dynamic code analysis tool.

B. Security

An OSS project must include a lead programmer that is knowledgeable on developing secure software. FLOSS insists the developer must implement the eight principles from Saltzer and Schroeder [16]. Saltzer and Schroder emphasize simplistic software design. OSS must enforce access decisions on user permission over exclusion. Access to objects must require a check for the privilege. Each and every program and the user of the software program must work using the lowest set of authority rights to do the task. OSS must mitigate a load of procedure shared by multiple system users and depend on by the system users. The software design must implement a method that requires two keys to access protected regions. The plan should be open and not be secret. The last principle involves psychological acceptability, which states that the interface design is implemented for simplicity of use so that users can apply the security procedures correctly [16]. OSS uses a community badge system to identify and encourage adherence to OSS security, design and coding best practices [15].

VI. DESIGN PATTERNS

Knowledge and use of design patterns are essential in software engineering. A design pattern is a solution that has been proven effective to a recurring design problem [17]. Both CSS and OSS utilizes design patterns as part of the design phase.

The Linux Foundation requires kernel programmers to engage in peer review during the design iteration by posting the plan to the relevant email distribution list. In addition to the discovery of design flaws, another reason for the early review is to ensure that the developer is not trying to re-invent the wheel. The email distribution list is the process for OSS developers to discover existing solutions to recurring problems. "Code which reinvents existing wheels is not only wasteful; it will also not be accepted into the mainline kernel [5]." OSS members are a mass resource of current and evolving design patterns.

In 4156, we learned about the Model-View-Controller (MVC) Pattern, which uses a proven design for implementing user interfaces. A discussion of design patterns is beyond the scope of this paper, but it is important for the context of this paper, to mention design patterns **do** exist in OSS. An example of a particular design pattern used in the Linux kernel is the process of using a counting variable to manage the CPU's resources. Reference counting implements an object which

tracks used and free resources for the kernel to manage memory [13]. The Linux kernel uses data structures uniquely handled by smaller, simpler structures. An example of this is the way the kernel handles the task structures for all the processes running on the CPU. Linux places embedded head nodes in objects to construct a linked list, instead of creating linked list structures out of the objects themselves.

VII. Life

A. Agile For Child Rearing

The core values OSS and CSS methodologies hold in common can be extracted and used in any space. Bruce Feiler conducted a case study on his family using Agile paradigms to manage his family [3]. Family participation during stand-up meetings at dinner increased communication and empowered everyone to give input to problem-solving scrums. A big board was used to track family challenges, and Feiler empowered his children to create punishments. By using just a few values from Agile, Feiler was able to transform his team. The fundamental idea above all was creating a culture of adaptation. When goals are broken down into small steps, team reflection can frequently be done. The continuous self-reflection and peer feedback make it easy and natural to stay on course to the goal [3].

B. Personal Goals

Like Bruce Feiler, I implement many of these paradigms to help me reach my personal and professional goals. For instance, I seek continuous and iterative feedback from my clients and employees. The process of participating in self-reflection and constructive criticism from peers opens virtual pathways leading to creative ideas to solutions that I face each day. Involving and empowering my employees creates a shared vision for the values I build my company culture with. I simplify tasks to approach them iteratively and break down tasks methodically until they are small primitive blobs. We use the term *blob* to refer to a problem that we cannot break down any further. Once a problem is reduced to a blob, the pieces of the problem when attacked individually are that difficult to solve anymore. With core software paradigms embedded in my head, I stress the importance of simplicity and correctness during employee training. No shortcuts, and no easy way out. My employees and peers are empowered to be creative, and we reflect and self-adjust regularly. I implement these ideas every day in my professional, personal and academic life. Using iterative goal planning has helped bring order to my life and have enabled me to accomplish more each day.

C. Challenges For the Reader

Start each day with a quick meeting, this will allow your team or family to keep progressing towards your goals. If you are stalling in your journey, adapt so you can continue. Always seek constructive feedback from peers, co-workers, family, or whoever. Break up problems into small primitive blobs, this will help you find the solution to the big picture faster and more efficiently. Self-reflect often and think about other ways you could have approached a challenge. Self-reflection can be used as a process of self-evaluation of yourself, not just in a collaborative environment. These ideas, as well as many others in software engineering, are excellent tools we can utilize to help us be more productive and simplify problems we face every day.

VIII. Case Study

A. The Idea

Juggling the jobs life tosses at us is a common problem we all face in our everyday lives. My particular challenges come from running a business full time, getting a graduate degree, and balancing a work-school-life balance.

TABLE IV. License Plate Recognition Data

[d] PRI incorporated, 2016

Fig. 4. Example Of LPR Data Verification

Without clouding the reader with specifics about the specific details that define the operations of my business, to demonstrate the impact paradigms can have, I will discuss my business in the context of verifications.

My business provides license plate recognition historical geolocation data to clients. Hardware coupled with software captures the locations of millions of parked cars every day. Financial institutions, insurance companies, and government agencies buy this information and analyze the data to identify behavioral habits. For confidentiality and brevity, I provide purposely vague details with the hopes that the reader can abstract the purpose of the elements discussed as they apply only to the context of my study.

The license plate recognition algorithm is not as perfect as a human eye, and approximately one percent of our scans contain a defected read. A typical example of this bug can be illustrated when the software determines a zero to be the letter *O*. This type of error is not a priority issue in my business. Each scan has

multiple *hits*, so clients and my analysts can use the data from other reads to complete the verification. A *hit* is defined as a scan that reads and stores the correct license plate and maps it to the geolocation of the vehicle. Figure 4 depicts a valid hit. So, if we have twenty verified hits on a vehicle, chanches are the one with an error is not going to reveal any surprising data. With this understanding, the tedious process involved with cross referencing data and motor vehicle records, as well as physically verifying the data to meet compliance regulations takes the least precedence in terms of our daily operations. The data is placed in a repository for verification, and if and when, there is time to catch up, a member of my team will work on that task. The reason we care at all about the missing piece of data is that every once in a while, albeit rarely, that one scan may provide the location data the client needs to complete the last piece of a puzzle they need to solve.

I have always rigorously demanded a culture that reflects a the mission of attention to detail. So for me, letting any tasks pile up is not practicing a clean operation. In effort to address this issue, I chose to experiment with software paradigms to see if it was possible for my team to catch up on low priority tasks. As a case study, I decided to track these low priority verifications while implementing new methods in my business model during the Fall semester of 2015. My goal was to determine if I could measure an improvement in performance simply by applying software engineering methods to my business. Further details of the verification process or my company are beyond the scope for the context of my research. The reader can abstract the idea of verifications to any low priority issue or task. To put it in the context of software engineering, the defect analogous to a verification would be a feature that does not affect the functionality, security, or performance of a project and is strictly cosmetic or optional. An abstraction of something outside of software engineering could also be something as trivial of having a goal to learn a new language for fun and or spending an extra 20 minutes in the gym. On the days when you have some spare time you devote an hour to Rosetta Stone, maybe while you are in the gym on the treadmill, but obviously, work, school, family, and life's high priority commitments take precedence. Although low priority verifications are not urgent, they pile up quickly over time and eventually the data may become outdated and loses value for my clients. For me, its sloppy operations, and it is not the service I promise to my clients. So it was a natural choice to track level- verifications for me because it is easily trackable and a goal I want my team to conquer. Level- or level negative is a flag my company uses to identify low priority tasks.

To validate my analysis, I kept all other factors equal except adding a few software engineering paradigms to our workflow. I did not hire additional employees, I did not work extra hours on verifications instead of doing homework, and I did not tell my employees they must work overtime. I kept **all** other factors equal.

What I did do is start using a big board to track our goals, progress, and issues of our normal business operations. I started having morning stand-up meetings and encouraged my management to implement stand-up meetings in their departments. I broke up each week into iterations with specific goals and as a team we planned, set goals, tracked our velocity, and continuously adapted our design. My employees participate in morning meetings and contribute ideas in our brainstorming sessions. Employees are empowered to adjust the plan according to our progress, which we measure and track. My employees now play an exponentially greater role in molding our policies and procedures. My staff takes more ownership in responsibilities, and efficiency and employee engagement is evident when our iterations flow. The key element of this case study is when we were able to inject low priority verifications, along with additional optional tasks into our iterations. Tasks that sat in the level negative queue began to find their way onto the task board.

Outstanding verifications at the end of each day is zero, including the low priority verifications that were backlogged since I started my journey at Columbia. We achieved this with less employees and a record amount of business requests from our clients.

Figure 5 tracks the number of outstanding level negative verifications over the course of the study.

TABLE V. ANALYSIS OF PARIDIGMS ON CASE STUDY TASKS

[e.] PRI dATa Incorporated 2016

Fig. 5. LPR Level- Verifications Analyzed Implementing Methodologies

A bonus side effect of my case study is we reduced client complaints by 26%. Instead of reactive management, we methodically detect issues early. Proactive team behavior increased our internal customer satisfaction score.

B. More Paradigm Experiments

Over the course of the semester, I began to implement OSS paradigms to experiment their effectiveness. One specific activity I modified was I started utilizing community message boards to share some my business challenges with peers instead of trying to solve everything myself and thinking that only my approach works best. I found myself more open to peer feedback, willing to approach a problem with methods outside of my comfort zone and to search for design patters to solve business solutions.

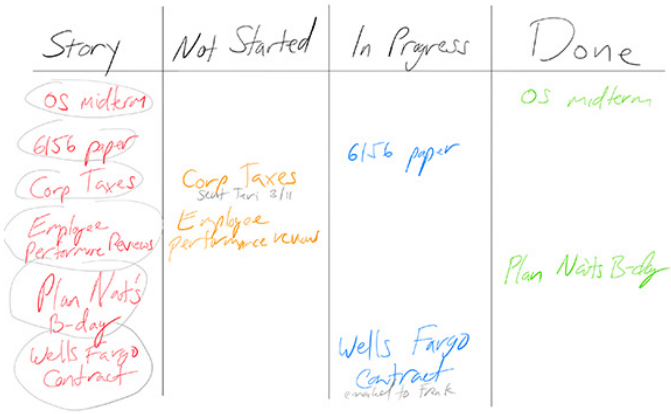
For academic challenges I use incremental cycles to break down reading assignments. I divide my homework up into smaller tasks. First, I read over the introduction to each section and read the questions presented at the end of the assignment. From this documentation, I draft up requirements and user stories to create a learning design model to document my goals of the assigned material. I track my progress and adapt. If I discover material I am not comfortable with, I modify my plan to include additional resources about the additional concept. By using iterative cycles with frequent self-testing, I can retain more of the material.

My life is chaotic owning a business and being in graduate school, so finding time for relationships has always been a challenge for me. I decided to implement software design strategies to help me in this department as well.

My girlfriends have complained I work too much or spend too much time studying. I convinced my current girlfriend Natalie to experiment with ten minute meetings in the morning, and for five months continuously we have our stand-up meetings every day. We effectively discuss our schedules and goals for the week and plan our time together. In the past, when work or school commitments unexpectedly come up, I wait until a few hours before date night to break the news that plans must change. Natalie is not innocent to this shortcoming either, as a lawyer she continuously has unplanned commitments come up and needs to cancel plans. By having our meeting each day, we play an active role in our commitments, and it has brought us closer. I feel that we have a better understanding of each other. We found our communication to each other is clearer, and we adapt to each other's needs. Most importantly, we are agile and work together as a team on planning our quality time together. Planning as a couple has also made us both more dedicated to making sure we spend quality time together. We also implement feedback from each other and bounce our personal, professional, and academic goals together to plan our iterations together. Implementing software paradigms brought us together as a couple and increased our quality time together by 25%.

Figure 6 is a recent snapshot of my board that I manage tasks on. I folllow a combination of CSS and OSS methodologies in my life and have found the tools they offer to be indispensable. I encourage the reader to engage in further study of software methodologies. Paradigms will make you a better programmer and will make teams more productive. They can also make you more productive in life and help you plan a path to your goals!

TABLE VI. MY BIG BOARD



[f] [My personal Big Board.]

Fig. 6. My Big Board to manage workflow.

IX. CONCLUSION

CSS and OSS paradigms predicate on many of the same core values. Both models emphasize frequently building working software, continuous reviews, and testing. Principles encourage simplicity and keeping the design as simple as possible. Values stress working in a culture of collaboration and teamwork. Listening and learning from your users during development can create bigger and better ideas. Equally important are values that require engaging in peer and self-reflection, adapting to change, and the focus on doing the job right, even when no one is looking.

The commonality of paradigms was something I was surprised to learn. Before I started this project, the idea that CSS and OCC methodologies shared core values was a contradiction. OSS was to me symbolic with large distributed teams creating software in a chaotic and unorganized nature. I was surprised to find the OSS development cycle to be so well-defined and methodical.

There is much to learn and implement from OSS. My hope for the reader after reading this paper is to be at least willing to try some of the paradigms from a software engineering in your life. I learned that like Agile, I can also now utilize OSS models and methodologies as my personal design patterns in life. I look forward to studying methods from additional spaces to see how humans can implement them outside their intended industry.

REFERENCES

[1] E. S. Raymod, "The Cathedral and the Bazzar," Tim O'Reilly (Ed.). O'Reilly & Associates, Inc., Sebastopol, 1999.

[2] R. Love, Linux Kernel Development, 3rd ed. ed., I. Person Education, Ed., Boston: Addison-Wesley, 2010.

[3] B. Feiler, Writer, Agile programming - for your family. [Performance]. http://www.ted.com/talks/bruce_feiler_agile_programming_for_your_family, 2013.

[4] V. Potdar and E. Chang, "Open Source and Closed Source Software Development Methodologies," in Feller, J. and Fitzgeralg, B. and Hissam, S. and Lakhani, K. (ed), ICSE 2004: Twenty Sixth International Conference on Software Engineering (with) Collaboration Conflict and Control: Proceedings of the Fourth Workshop on Open Source Software Engineering, Edinburgh, 2004.

[5] J. Corbet, "How to Participate in the Linux Community," Linux Foundation, 13 May 2011. [Online]. Available: http://www.linuxfoundation.org/content/1-guide-kernel-development-process.

[6] M. Bar and K. Fogel, Open Source Development with CVS, 3rd Edition ed., J. Duntemann, Ed., Scottsdale, Arizona: Paraglyph Press, 2003.

[7] Black Duck Software, "The Ninth Annual Future of Open Source Survey," 15 April 2015. [Online]. Available: https://www.blackducksoftware.com/future-of-open-source.

[8] D. Spinellis, "A Tale Of Four Kernels," CSE '08: Proceedings of the 30th International Conference on Software Engineering, pp. 381-390, 1 May 2008.

[9] Coverity Scan, "Open Source Report 2014," Synopsis, Inc, San Fransisco, 2014.

[10] Coverity, "2012 Coverity Scan Open Source Report," Coverity, San Francisco, 2013.

[11] K. Beck, "Twelve Principles of Agile Software," 1 January 2001. [Online]. Available: http://agilemanifesto.org.

[12] R. Goldman and R. P. Gabriel, "Innovation Happens Elsewhere: Open Source as Business Strategy," 25 April 2005. [Online]. Available: http://www.dreamsongs.com/IHE/IHE-28.html.

[13] N. Brown, "Linux Weekly .Net," 8 June 2009. [Online]. Available: https://lwn.net/Articles/336224/.

[14] S. McConnell, Code Complete, 2nd ed. ed., Walla Walla: Microsoft Press, 2004.

[15] D. Wheeler, "Best Practices Criteria for Free/Libre and Open Source Software (FLOSS) (version 0.5.0)," Linux Foundation, Portland, 2016.

[16] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," Project MAC and the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge.

[17] E. Gamma, Design Patterns: Elements of Reusable Object-Oriented Software, Zurich: Addison-Wesley, 1994.

[18] D. Bovet, Understanding the Linux Kernel, 3rd ed. ed., Sebastopol: O'Reilly Media, Inc, 2006.

[19] I. Bowman, "Linux as a Case Study: Its Extracted Software Architecture," ICSE, pp. 555-563, 1999.

[20] P. Hinds, "Distrubuted Work," in Distrubuted Work, Cambridge, The Mit Press, 2002, pp. 382-481.

[21] A. Silberschatz, Operating System Concepts, New Haven: Wiley, 2014.

[22] M. Kerrisk, The Linux Programming Interface, San Francisco: no starch press.

[23] A. Tannenbaum, "Modern Operating Systems," in Linux: A Case Study, Amsterdam, Pearson, 2015, pp. 713-857.

[24] A. Ampatzoglou, "An Empirical Study on Design Pattern Usage on Open-Source Software," in ENASE 2010 - Proceedings of the Fifth International Conference on Evaluation of Novel Approaches to Software Engineering, Athens, 2010.

.