New Methodology of Automatic Design Collaboration

by

Shawn Pike

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2016 by the
Graduate Supervisory Committee:

Ashraf Gaffar, Chair
Timothy Lindquist
Richard Whitehouse

ARIZONA STATE UNIVERSITY

December 2016

ABSTRACT

When software design teams attempt to collaborate on different design documents they suffer from a serious collaboration problem. Designers collaborate either in person or remotely. In person collaboration is expensive but effective. Remote collaboration is inexpensive but inefficient. In, order to gain the most benefit from collaboration there needs to be remote collaboration that is not only cheap but also as efficient as physical collaboration.

Remotely collaborating on software design relies on general tools such as Word, and Excel. These tools are then shared in an inefficient manner by using either email, cloud based file locking tools, or something like google docs. Because these tools either increase the number of design building blocks, or limit the number of available times in which one can work on a specific document, they drastically decrease productivity.

This thesis outlines a new methodology to increase design productivity, accomplished by providing design specific collaboration. Using version control systems, this methodology allows for effective project collaboration between remotely located design teams. The methodology of this paper encompasses role management, policy management, and design artifact management, including nonfunctional requirements. Version control can be used for different design products, improving communication and productivity amongst design teams. This thesis outlines this methodology and then outlines a proof of concept tool that embodies the core of these principles.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF FIGURES

Chapter 1

INTRODUCTION

Designers have suffered through poor design collaboration tools for too long. It's time for a new methodology, combining existing tools and workflows to collaborate in a way that is easy and scalable. As the complexity of design projects increases, work must be split into specialized groups, and collaboration between these groups is a necessity. There are tools being used for code level collaboration. It is time to use these tools for natural language design work products.

One of the most difficult things in Software engineering is to understand just what needs to be built. Requirements engineering is incredibly challenging because the problem domain is not fully understood. This lack of understanding increases with the complexity of the project. Another reason it is so difficult to understand the problem is that it needs to be understood at different levels of abstraction Ruckert 2015. It must be understood at a high-level first: communication with all of the stakeholders, with the purpose of having a shared understanding Bittner and Spence 2008. Each stakeholder needs to have the same understanding in the same way because not all implemented solutions are the same. The implemented solution must be done in a way that the stakeholders are expecting it to be done. This is important is because the solution must be adopted, but if implementation is done in a way that causes the users to drastically change their current workflow, adoption is unlikely.

The point of a software solution is to implement a given business process, created to solve a given business goal. Humans are not computers. Humans communi-

cate using natural language, and computers communicate using logic gates of ones and zeros. Humans will implement a given process in a myriad of ways, whereas a computer will only implement a given process in the exact way it was instructed. One can describe a process to a human at a very high level with a goal in mind, and the human will fill in the necessary details and assumptions that fit their understanding of the problem to solve a given goal. A computer, on the other hand, needs to know a given process at a very low-level and in a very detailed way to solve the process. Once the computer has this information, the process will be solved at lightning speed, with incredible rates of repeatability and efficiency.

Bridging the gap from a very high level to a very low level of communication is not easy. To achieve this one must understand a problem in a layered approach, each layer exposing information to a lower layer until the lowest layer can be implemented in a way a computer understands. Each layer adds the necessary details required for a lower layer to perform that layers functions. Our work will concern itself with how to implement a thorough understanding of a given problem at a high-level. This thesis will detail a methodology to facilitate collective high-level design. High-level requirements will be discussed in the context of two high-level methodologies: use cases and task modeling. Each of these paradigms have the same goal, which is to describe a set of requirements that will be understood by all stakeholders. The final set of formal specifications will be written so that software construction can begin. They both have their advantages and disadvantages. After we explore each approach, then they will be compared and contrasted, and an argument will be made for one approach over the other.

For a given set of requirements to be built using either use cases or task modeling, one must gather all of the information that one can for the requirements

to be thorough. This can be a real challenge for several reasons. One reason is that the number of stakeholders can be extensive. An application such as Facebook which has over 1.7 billion users. Getting the input of all of these stakeholders would be next to impossible. This is solved by having a stakeholder representative Bittner and Spence 2008. This is a representative that can speak for the users in their stakeholder group. This representative will represent a specific group such as teenagers, the elderly, this disabled, etc. This will reduce the burden of coordinating with all of the stakeholders.

Stakeholders in a given project are actors that will in some way be affected by the system being built. This means that stakeholders can be both internal and external. Internal stakeholders are those people who have a stake in what is being built. The main ingredient for gathering requirements from stakeholders is communication. Communication is needed for collaboration to occur. However, collaboration has a much broader definition in that there must be coordination along with the communication. Collaboration is more appropriate than communication when working with internal stakeholders. The reason for this is that internal stakeholders have work products that are specific to them. These work products can vary amongst the different stakeholders. Each member of a given stakeholder group has their set of necessary requirements that they feel should be added to a given software project. It is essential to know what these requirements are at design time to effectively build a given project. Doing this prevents rework down the road. There are many software methodologies that deal with the software lifecycle. These methodologies vary from one extreme where all aspects of each project phase are to be known prior to moving on to the next phase, (the waterfall method) vs.

other approaches where only the requirements that need to be known thoroughly are the ones that are being implemented in the current sprint, (the agile method).

However, with either approach, there is a need for extensive collaboration amongst the various stakeholders to gather the entire scope of requirements. Collaboration is not new and there are an extensive set of methodologies and tools out there to facilitate it. This work is going to outline another methodology that will use existing methodologies and tools to have people collaborate asynchronously for use case creation. A proof of concept tool will be outlined in this thesis.

The purpose of our work is to not only argue that use cases are a good choice for creating thorough high-level requirements, but to also argue that there is a need for a collaborative tool that will facilitate the creation of them.

Chapter 2

GENERAL PROCESS MODEL

2.1 Overview

Software projects are built using processes, and this is true regardless of the size of software projects. Even if a single developer creates a project, then a process will be followed no matter how disorganized the given tasks may seem. In these small projects, one can have a detailed process such as the personal software process Humphrey 2000. Organizations often switch the responsibility of productivity to the individual developer. The personal software process is powerful and has lots of value but it does suffer from an adoption problem. Which some suggest is due to having to make a conceptual switch to recording the process along with the overhead of actually doing it Johnson et al. 2003. On the other end of the spectrum, large projects follow a more in-depth software process. The purpose of having a process and software engineering is because of how chaotic creating quality software truly is. Chaos in software development comes from its complexity Pressman 2005. The project will need to complete many steps from conception to completion. Different people have ideas on what tasks to complete, how to do it, and how long it takes for them to complete the task. As a result, each of the people who are involved in the creation of the software will often conflict and work at cross purposes.

There is a push for some to try to create a mechanical way to do software engineering in much of the same way as compilers were used to bring assembly code to a higher level language Dijkstra 2010. John Backus in the fifties led a group

to create Speedcode and FORTRAN Allen 1981. These were the first high-level languages that began to take programmers away from writing assembly language. However, since assembly language was the language of a computer there needed to be a compiler to translate the high-level language into assembly language. Compilers were a way to deal with the abstraction of a high-level language. Creating software is an exercise in levels of abstraction from conception to the 0 and 1s of a computer. So the question then arises how abstract can one go and still get down to a language that a computer can understand? The answer to this question is beyond the scope of this thesis but needless to say there is a limit. It's hard for a designer to translate a vague set of requirements, say something that "will do great things," into a web app that will implement those features. There are many activities outside of writing the software that is required for an application to work Dijkstra 2010.

Quality software entails a vast number of things like software engineering, code generation, and system engineering for deployment and maintenance. Software engineering concerns itself with things like processes and process models, requirements engineering, analysis modeling, design engineering, architectural design, component-level design, UI design, and testing, along with ways to measure and evaluate the quality of these aspects of software engineering Pressman 2005. Code generation entails the need to understand algorithms, network concerns such as minimization of code for web applications, databases and ORMs, version control systems for sharing code and collaborating, security, and the implementation of the architecture, plus many other things. System operations engineers concern themselves with maintenance and deployment. The one thing that helps coordinate all these different pieces is to have sound processes in place.

The purpose of having a process is to understand and control the activities that are required to meet the business needs of the project and to do it on time and in a cost-effective manner. There is great difficulty in accomplishing these goals. In a paper written by Walt Scacchi at the Institute for Software Research at the University of California in Irvine, he defines it this way,

> ".. developing large software systems is difficult because it involves complex engineering tasks that may require iteration and rework before completion." Scacchi 2001.

Pressman outlines the process framework Pressman 2005. This process framework describes at a high-level all the activities that would be required to complete the software project. There are three aspects to this process framework. There are the umbrella activities that are done to support the specific activities needed to complete a physical task. Some umbrella activities would be risk management, software project tracking and control, software quality assurance, formal technical reviews, measurement, software configuration management, reusability management, and work product preparation and production. These umbrella activities support the framework activities. Pressman defines five main framework activities. These activities are communications, planning, modeling, construction, and deployment. Then inside each one of these framework activities is software engineering actions. These are software engineering guidance task sets. A development team will need to complete these task sets to satisfy an engineering action. An engineering action will need to be completed to satisfy a framework activity Pressman 2005.

There are many different processes out there that are used to solve a given software project or business process. When an effort is made to combine the processes

that are similar, one ends up with a process model in which Colette Rolland classifies into four types or facets, and they are activity-oriented, product-oriented, decision-oriented, and contextual Rolland 1998. Process models are important because they outline in an idealized form of what a proper process should be. They are similar to classes and objects in the sense that a class is a blueprint for an object whereas an object is an instantiation of the class. A process model is a blueprint for an actual process. The process is an instance of the process model.

Scacchi defines a process model as

> "...a networked sequence of activities, objects, transformations, and events that embody strategies for accomplishing software evolution" Scacchi 2001.

He argues that process models have power due to their "rich notation, syntax, or semantics" Scacchi 2001. Scacchi means that process models offer the ability to use modeling languages to represent the process model. Formal languages help solve a fundamental problem of communication, and that is that two people do not only understand each other, but they understand each other in the same way. Standardizing a language facilitates this implementation. This standardization offers a way for people to have the same understanding of how a process works.

2.2 Business Process

Business processes are essential to designers. Designers build software that is in synch with business processes. A software solution is unable to automate a process if a business process is undefined. For instance, when one goes to the DMV, many steps must be followed to renew a license. These steps entail such things as looking

up the individual, collecting the money, taking the picture and potentially much more. A designer must know which steps to follow, in what order, and the purpose of each step. Business processes are essential to designers.

The abstraction level that this thesis is concerned with is the cloud level or the need level. The business process level is the level that is used to either understand the current processes or to create new processes that help a business achieve greater efficiency in accomplishing its goals. Following a business process is a vital step in requirements engineering. Requirements engineering is the first step in software engineering because it must understand what a stakeholder wants to build. It is important to point out that doing requirements engineering does not mean that the only solution to a problem is creating software. Once one understands what one is building then, one should think about all possible solutions to the problem.

A problem with software is adoption. Learnability is a significant barrier to software adoption Rafique et al. 2012. People have a tendency to solve a problem in a certain way and stick to it. A prospective user must weigh the cost-benefit between the difficulty of learning something new vs. the benefit of what the new solution will offer. Defining the business process will facilitate this understanding. One such way to describe a business process in a standardized way is to implement BPM, BPML, and BPMI.

2.3 BPML BPMI

In the context of this thesis, we are discussing two different types of processes. The first type is to understand the problem that software is meant to address. The second type is concerned with the creation of software requirements. BPM is used

to understand the problem and then help designers design the software that is needed to solve these problems.

There are different ways to define BPM. Theodore Panagacos defines it this way,

> "BPM is the science of building, identifying, and managing processes so they can be improved for maximum efficiency. BPM deals with identifying all the processes associated with your organization; analyzing them for efficiency and effectiveness; measuring the results over a period of time; and optimizing these processes. Panagacos 2012"

The European Association of BPM defines it this way,

> "Business Process Management (BPM) is a systemic approach geared to capture, design, execute, document, measure, monitor and control automated as well as non-automated processes in order to meet the objectives that are aligned with the business strategy of a company. BPM embraces the conscious, comprehensive and increasingly technology-enabled definition, improvement, innovation and maintenance of end-to-end processes. Through this systemic and conscious management of processes, companies achieve better results in a faster and more flexible way." Business Process Management (Hrsg.) 2011.

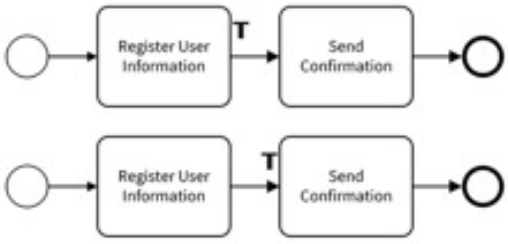
Each of these definitions describes BPM as a way to understand and improve the processes of a business or an organization. Understanding the processes of an organization allows one to improve on it. Improvements in a process can happen in many ways, and one of those ways can be a software solution.

BPMN is used to bridge the gap between a higher level understanding of how a business works and a lower level understanding of how a business works. Much like a programming language requires a deeper understanding of a problem than a designers understanding, so does BPMN. When a business analyst uses BPMN, they are forced to fill in all assumptions they may have on how a process works.

BPMN is a Business Processing language that is used to model business processes. Esteban Herrera defines it this way, "The Business Process Modeling Notation provides a STANDARD way of representing business processes." Herrera 2015.

These notations are diagrams that describe a sequence of activities. The token signifies the processes state as it moves through the process.

Figure 1. Showing a BPM activity



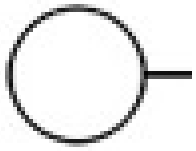
Herrera 2015

For instance, this is an example of a diagram that starts by registering a user and then sending them a confirmation once the user is registered. An open circle signifies the beginning of a process:

The arrows point toward the processes flow:

The big rectangles with rounded edges describe a given activity. There are two activities in this diagram, one for registering a user and one for sending a confirmation:

Figure 2. Starting point of a BPM activity

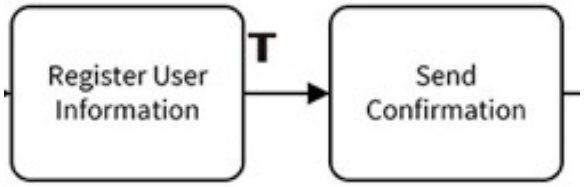


Herrera 2015

Figure 3. Shows the direction of the action

Herrera 2015 

Figure 4. Shows 2 events



Herrera 2015

A circle with a bold edge signifies the completion the activity in the process.

Figure 5. Shows a stopping point

Herrera 2015 

A capital T that is bolded represents an essential aspect to this diagram.

When all of the processing of the current step completes, then the token transitions to the next step. So, if we were to lay out the diagram entirely then we would begin with an activity that has not yet begun:

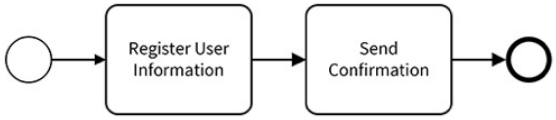
This is the starting point:

The T on the left of the activity diagram denotes the beginning state of that

Figure 6. Shows a token

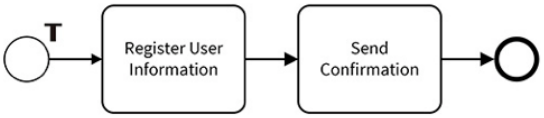
Herrera 2015

Figure 7. Shows a business process



Herrera 2015

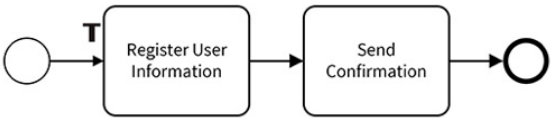
Figure 8. Shows a business process with a token



Herrera 2015

activity. Upon completion of the activity, the T moves to the right of the activity and indicates the end of that activity:

Figure 9. Shows a token moving from the starting point to the first event.



Herrera 2015

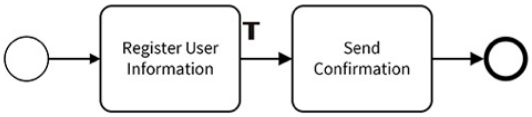
Ending the Registration Process:

The token begins the "send the confirmation" activity:

The following diagram represents the completion of the activity:

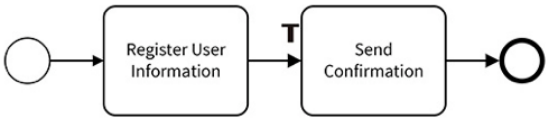
Then finally once the activity has been completed, the token moves to a bolded circle which denote that the activity has finished.

Figure 10. This shows where the token is after it finished the event.



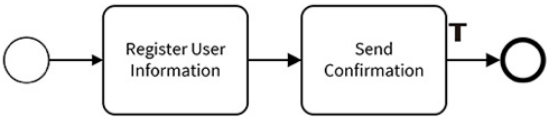
Herrera 2015S

Figure 11. Shows the token starting the send confirmation event



Herrera 2015

Figure 12. Shows the token finishing the send confirmation event



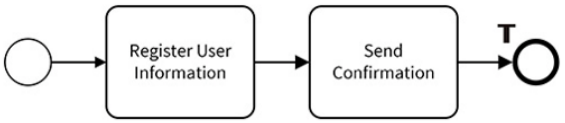
Herrera 2015

This activity is a small example of BPMN. BPMN encompasses more notations and is applied many more contexts such as Swimlanes and Collaboration Diagrams, conversation diagrams, choreography diagrams, and data objects Herrera 2015.

Different stakeholders need different levels of abstraction to effectively do their jobs. Managers on all levels only need to know what is going on at a high level because they focus on strategy and direction. Managers are concerned with strategy and guidance, so they only need to know the what and the why, and they are less concerned with the how. Developers, on the other hand, need to understand how it gets done rather than on the why and the what.

BPMN focuses very heavily on notation, and this has the advantage of being able to see what a process is doing at a glance. Pictures are ideal for conveying a detailed process in a small amount of space, but this requires one to understand the language to make sense of the diagrams. This restriction may not be right for all stakeholders, so there are other methods to describe the process. We will explore

Figure 13. Show the token reach the finish



Herrera 2015

two such ways in the next two sections. These other ways are Use Cases and Task Modeling.

Use Cases and Task Modeling are one small level of abstraction below BPM. The reasons these two paradigms are less abstract is because they are used to create a set of specifications for software construction. We will first talk about design, and then we will speak of use cases in more depth.

Chapter 3

SOFTWARE DESIGN

3.1 Overview

This chapter is largely based upon the software design section of the SWEBOK. The SWEBOK defines the guide as "The purpose of the Guide is to describe the portion of the Body of Knowledge that is accepted, to organize that portion, and to provide topical access to it" Bourque, Fairley, and eds. 2014. The SWEBOK V3.0 is published by the IEEE Computer Society Board of Governors. It forms the basis of two of their software development certifications. This book seeks to synthesize the current body of knowledge in software engineering and to have software engineering recognized as an engineering discipline.

Software design is an essential part of the software lifecycle. The purpose of software design is to translate software requirements into a blueprint that can be handed off to software construction. This blueprint has two aspects. These two aspects constitute the two parts of software design. The first part is architectural design. Architectural design focuses on the high-level aspects of software design. The main purpose is to create components and the detail they communicate between them. The other part of software design is the detailed design. Detail design focuses on a lower level understanding of each of the components. It is this design that closely resembles something that software construction can use. There are many different types of design, but two of the major design types are D-design and FP-design. D-design means decomposition design. Decompositiondesign breakss

down the requirements into components. FP-design focuses on reusability. So if a pattern is created, then there is a likelihood that there will be something else within the same family of products that will be able to reuse that design. Web frameworks are a perfect example of this.

Below is a diagram from SWEBOK Guide V3.0.

The diagram below outlines all the major aspects of software design. It tries to touch on every area. I will summarize each of these areas.

3.2 Fundamentals

It is important to know software engineering sits amongst the other pieces of the software lifecycle. Software design sits in between software requirements and Software construction. It must make sense of all the elicitation efforts and then translate them into something that a developer can construct. Software design is a process. It is this process that does this translation, and the result of this process is the blueprint.

The figure below shows the traditional software stack:

This figure shows a traditional design stack:

One must understand the principles of software design to have a fundamental understanding of it. A good architectural model takes into account the following seven principles.

The first principle is abstraction. The purpose of abstraction is to make available only those pieces of information that are relevant and then hide the rest.

Figure 14. Breakdown of Topics for the Software Design KA

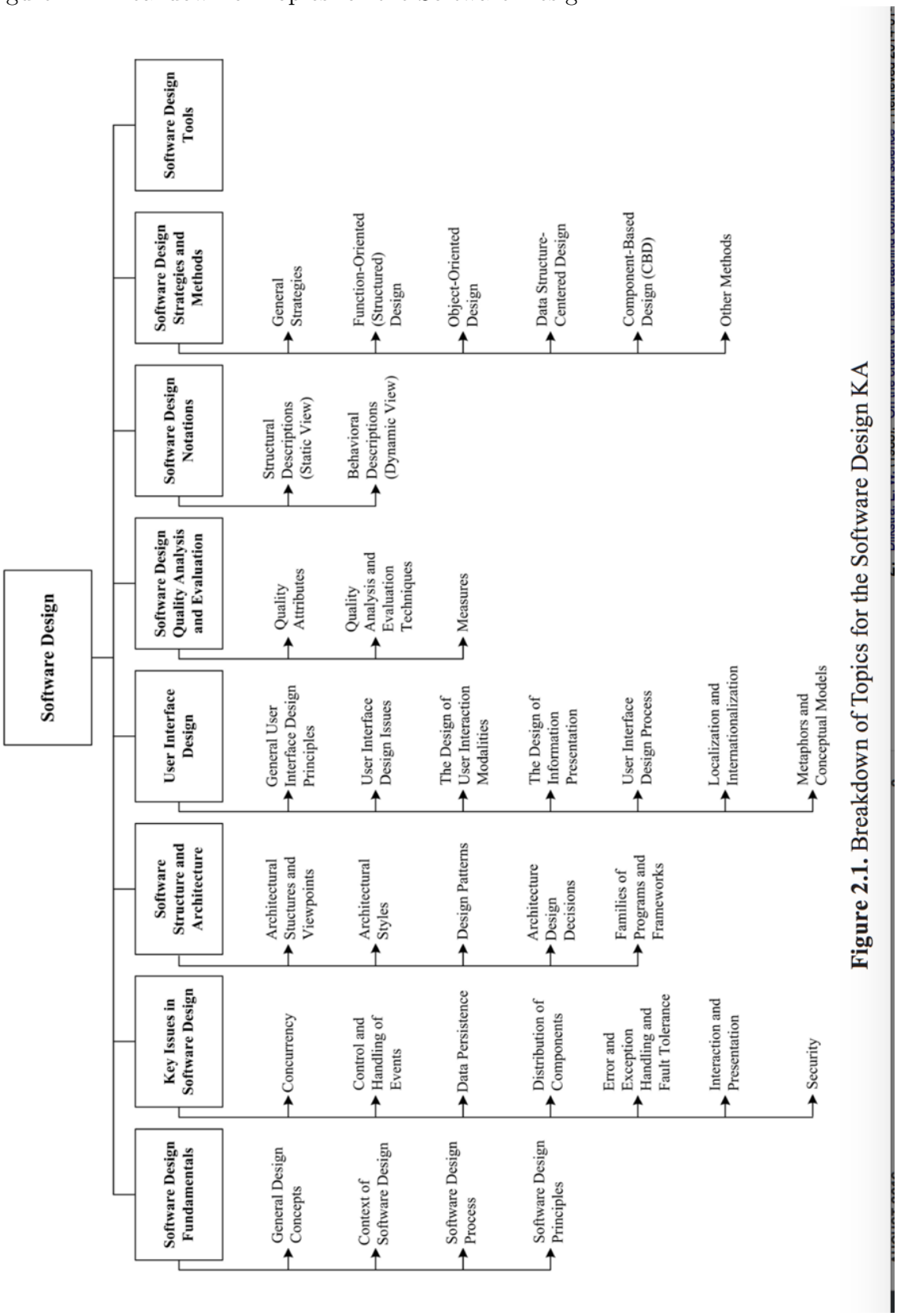


Figure 2.1. Breakdown of Topics for the Software Design KA

Figure 15. Traditional Software Stack

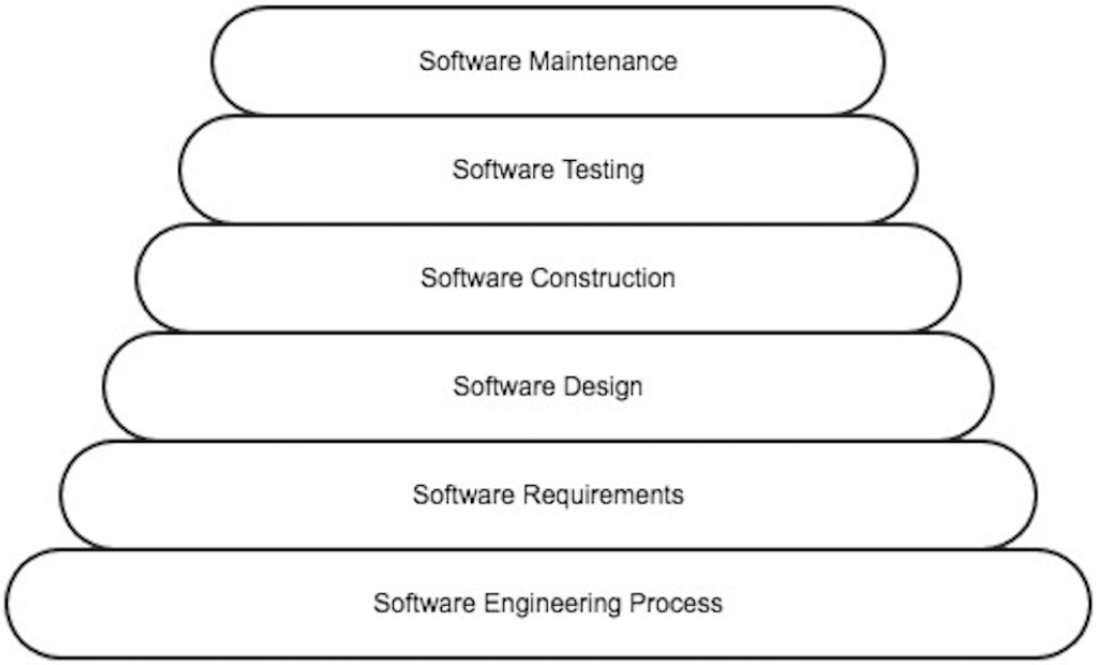
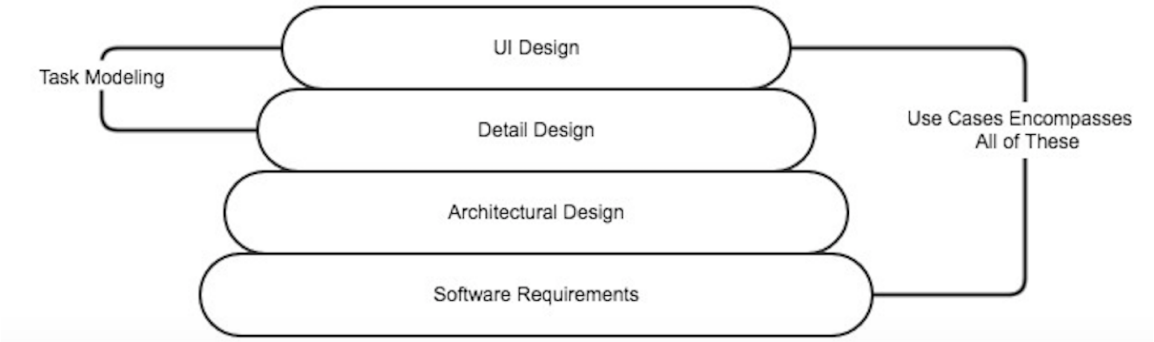


Figure 16. Traditional Design Stack



The second principle is coupling and cohesion. Coupling focuses on the interdependence between two components and cohesion focuses on the strength of those bonds.

The third principle is decomposition and modularization. The point of this

principle is to group the different functionalities and features into their own components.

The fourth principle is encapsulation and information hiding. The point of this is to protect the information within such things as classes and other data structures.

The fifth principle falls in line with the fourth principle, and that is to separate the interface and implementation. The implementation encapsulates and hides the data. The interface is what exposes information to any outside source that wants to make use of it.

The sixth principal focuses on sufficiency, completeness, and primitiveness. Sufficiency and completeness refer to having an architectural model that will satisfy all the requirements that were uncovered during the requirements engineering phase. Primitiveness focuses on utilizing design patterns that are easy to implement.

3.3 Crosscutting Issues

Creating an architectural model will need to take into account nonfunctional requirements. These are requirements that affect the quality of the software.

The SWEBOK talks about seven major areas. These are concurrency, control and the handle of the events, data persistence, distribution of components, error and exception handling and fault tolerance, interaction and presentation, and then finally security.

The area of interaction and presentation is interesting because it is separate from the architectural and detail design of software design. UI design focuses on the interaction of the user and software product. A good design separates the data

and the interface. All data structures should be in a different layer. The very popular MVC structure is an example where the view and the model are in two separate layers.

3.4 Architecture and Details

When a designer creates an architectural model, the designer will be building a multifaceted model. Each facet relates to a different person or stakeholder. There are many different views, and some of them are the logical view, process view, the physical view, and the development view. Each team is concerned with one view over another.

At a high level you are trying to build a different architectural style, and at a low level, you are trying to build different design patterns. At a high level, some different architectural styles are distributed systems, interactive systems, and adaptive systems. Then at a low-level, the designer will be concerned with the design patterns. There are three general patterns. They are creational patterns, structural patterns, and behavioral patterns.

There are trade-offs with whatever choices the designer makes that concern either the architectural style or the design pattern they choose. Making decisions is more important to design than the creation of the design. An example would be trading maintainability for speed.

3.5 UI Design

UI design is different than both architectural design and detail design. UI design has seven principles. These principles are meant to maximize the user experience for each of the users. These principles are learnability, user familiarity, consistency, minimal surprise, recoverability, user guidance, and user diversity. These principles also take into account any user disabilities.

There are six different forms of human interaction with user interfaces. These six interactions are question and answer, direct manipulation, menu selection, form fill in, command language, and natural language.

One thing that is imperative is to use metaphors in UI design to a mental model. If software can provide a real world metaphor for a task or process, then the software is more intuitive to use. However there is a corollary, in that metaphors often have a regional or cultural connotation. So one metaphor will make sense in one region or culture, but will not make sense in another.

3.6 Notations

For software design to be understood, there must be a way in which one can understand the resultant designs, such as using notations. There are many different kinds of notations. Each of these notations serves a different purpose. Sometimes these notations are used for architectural design, detail design, or both.

Design can be split up into a structural view and a behavioral view. The structural view refers to a static design, and the behavioral view refers to a dynamic design.

Since design has many views and many facets it is important to have tools that reflect that. Some of the tools are going to refer to static design and other tools refer to dynamic design. From the static view, there are eight types that the SWEBOK refers to. These are architecture description languages, class and object diagrams, component diagrams, class responsibility collaborator cards, deployment diagrams, entity relationship diagrams, interface description languages, and structure charts. From the dynamic viewpoint, there are nine different tools that the SEWBOK outlines. These are activity diagrams, communication diagrams, Data flow diagrams, decision tables and diagrams, flowcharts, sequence diagrams, State transition and state chart diagrams, formal specification languages, pseudo code and program design languages.

3.7 Strategies and Methods

A designer needs multiple different kinds of strategies in order to design software. The reason for this is that there are many different types of software, so one type of design will not apply in all cases. There are four main types of strategies. The strategies are function-oriented design, object-oriented design, data structure-centered design, and component-based design.

3.8 Tools

When a designer creates a design, the designer will need tools to make the design process easier. Many tools can be used for this purpose. However, before we discuss these tools let us discuss use cases.

Chapter 4

USE CASES

4.1 Overview

Use cases span both requirements engineering and software design. Use cases are typically a part of the elicitation of requirements. The purpose of software requirements is to determine what to build and the purpose of software design is to determine how to build it. Use cases cross over this because they outline a communication dialog between the actor and the system to achieve a goal. This communication consists of the goal of the actor, the what, and the steps to achieve this aim, the how.

Use cases are mostly text, and therefore they are an appropriate design building block to discuss in detail.

4.2 Shared Understanding

Like BPM use cases are used to create a shared understanding. The question of how one can develop a shared understanding of requirements can be answered by creating use cases. Use cases are a form of modeling that is specifically designed to do one thing, and that is to create a shared understanding of the requirements of a system by modeling the systems behavior. This modeling centers around two primary components, actors, and the system Bittner and Spence 2008. An actor and the system interact in such a way that a story can be constructed to depict

this interaction. This story is one that can be told to all stakeholders so they can understand the represented behavior in the same way as each of the other stakeholders.

4.3 Use Case Basics

Use cases depict behaviors of the system that add value to the actors who are using the system Bittner and Spence 2008. Actors initiate the interaction of a system. The system then communicates back to the actor. The actor and the system communicate until the actor determines that something meaningful has been achieved. The use case model is considered to be the entire "set of all actors and use cases" Bittner and Spence 2008.

4.3.1 Diagrams

It is easy to want to represent the use cases in diagrams. Diagrams offer a fairly intuitive and powerful way to understand a concept quickly. However, diagrams are not explicit enough. Thus the understanding may be different between two stakeholders. Each one of these stakeholders are able to depict the goal in the diagram. However, the actual flow of events that take place to reach this goal can be very different. The use case descriptions are where the bulk of the understanding takes place. A textual representation is the only way the flow of events can be represented explicitly enough to facilitate a shared understanding amongst the various stakeholders Bittner and Spence 2008.

4.3.2 Functional vs Nonfunctional Behavior

Use cases are not intended to model all aspects of the system within the use case model itself. Stakeholders can mistakenly think that a system can be built quiconce a specific agreement has been made among the various stakeholders. After all, the purpose of creating a system is to provide value to a set of users that will ultimately provide value, monetary or otherwise, to the creators of a solution. However, even though use cases excel at modeling the behavior of a system, they fail at modeling the aspects of the system that are required to support those behaviors Bittner and Spence 2008. These two different aspects of a system are often categorized by functional and nonfunctional requirements.

Functional requirements are those requirements that describe the behavior of the system. The behavior of a system describes the goal of the system. In the case of requirements engineering, functional requirements are those that take the input of an actor or user and get an output from the system back to the user Bittner and Spence 2008.

Nonfunctional requirements, on the other hand, are those requirements that support the functional requirements so that they can work. These are known as cross-cutting issues in software design Griswold and Shonle M. 2006. If one only focused on functional requirements then one would have ignored security requirements, hardware requirements, performance requirements, supportability requirements, and any other requirements that are needed to effectively allow the functional requirements a way to provide value to the users of a system Bittner and Spence 2008.

Even though use cases do not provide a direct way to model nonfunctional requirements, it does not mean that nonfunctional requirements are excluded from

use case documentation. In fact, these nonfunctional requirements are included in the supplementary use case documents Bittner and Spence 2008. A complete picture of a system cannot be described by use cases alone, and the use case model requires these supplementary documents to be complete. These requirements are written in declarative statements, and they are directly traceable to the use case or use cases that depend upon them Bittner and Spence 2008. A few examples of a declarative statement for a nonfunctional requirement could be.

- For an actor to log in to the web application, the actor must be using an SHA-2 SSL certificate.
- The web server that is hosting the web application must be running on Tomcat 7.0.
- The web application must be built using the Groovy programming language.

Each one of these nonfunctional requirements can be traced to either a specific use case or a group of use cases and are required for the functional requirements to be truly functional.

4.3.3 Users and Actors

Users and actors are not one and the same. Actors are much more granular than users. Actors are all the roles that users partake in when they interact with the system. Thus a user can represent many actors Bittner and Spence 2008. If one were to log into the home page of a web application, one would likely see a great many numbers of features that a user could partake in. The user could look at their account settings, any videos that might happen to be available, or possibly read some newly created news content. The user gains value from the system by

clicking on any one of these links. However, when the user chooses a specific link to click and then communication is initiated with the system. The user becomes a specific actor. The communication between the system and the actor is outlined in a specific use case Bittner and Spence 2008. So, with each separate click the user becomes a separate actor and partakes in a separate use case.

4.3.4 Use Cases Are Active

Use cases are active scenarios with active communication and dialog between the system and the actor Bittner and Spence 2008. A given use case should be thought of as a situation in which the actor is doing something to the system and providing the system input for which the system can now respond. So when one names a use case one should name the use case in a verb-noun order Bittner and Spence 2008. Some examples include, "Gets account information," "Watches newest video," or "Reads the latest news item" Bittner and Spence 2008.

The verb-noun naming convention facilitates traceability. It surely follows that a customer would want one of their users to be able to view their account information, see the latest video or read the latest news item.

4.3.5 Features vs Use Cases

Features and use cases while related are quite different. Features have value in that they generate excitement for a product, they facilitate marketing and publicity, but are ineffective in conveying detailed information for developers Bittner and Spence 2008.

A better possible understanding of this is that use cases create context for all the functional features and requirements of a given system. It is easy to sit and brainstorm many cool and wonderful features that one would want a system to perform. However, it is difficult to be able to understand how to place all those features in a comprehensible form that a given stakeholder can understand. If a customer wants a user to be able to view their account information, see the latest video, or news article, then that is all well in good but it leaves some important contextual questions such as:

- In what order do you want a user to be able to do these things?
- Will the latest video or news item be a part of free content?
- If one is watching the latest video will the user have to navigate backwards to see the latest news items?

Each one these questions can only be answered through a thorough understanding of the flow of events that a user must partake to be able to use the system effectively Bittner and Spence 2008. The elicitation of requirements may generate thousands of features. However, a feature offers little understanding of how a system will work. Since use cases place all the features and requirements in context by gathering the related features under a goal oriented umbrella that provides meaning to the specific actor in the specific use case, a stakeholder gains a high level understanding of how the system works Bittner and Spence 2008.

4.3.6 Mental Model

Understanding the mental model of how a system works is an important part of creating a system that will work effectively Bittner and Spence 2008. The mental

model is the model of how a user expects the system to work based on their current understanding of how a task should be performed Bittner and Spence 2008. This understanding is derived from the user's prior experience of how they complete a task. If a user were to throw away a piece of paper or really anything, then it is quite likely that their mental model is that they should throw it away in a trash can. They have likely done this their entire life. So if one were to design a system and had a feature that allowed a user to delete something, then it would make sense to throw things into a trash can. On most desktops of different operating systems, there is a trash can. The trash can icon is used to delete items on the computer. Having a user delete items by using an icon of a trash can is consistent with the user's mental model. If a designer believes that an efficient manner to eject a disk image is to drag it into the trash can, then this can cause problems as it did for Apple Computer. Users thought that dragging the disk image into the trash can would delete all the files on that particular disk image instead of simply ejecting the disk. This created a support nightmare that had many customers calling Apple for help. Apple computer was able to solve this problem by issuing an update that had an eject button on the desktop Hackos and Redish 1998.

Understanding the mental model of the users will allow the designers to conceptualize the system or solution easily Bittner and Spence 2008. This understanding can then be presented to each of the stakeholders and then each of the stakeholders can understand this system at a high level. A key to this is to think of a system as closely to what the physical representation of the system is supposed to do as possible. Physical representations are powerful in that they give a tactile real life understanding. Object-oriented programming offers a profoundly popular way to take something as abstract as code organization and maintainability and place a

physical representation around it Bittner and Spence 2008. For instance, if one wanted to build a travel website then one could envision what one would need to do to be able to travel. One would have to:

- Know where they wanted to travel.
- When they wanted to travel.
- Input their travel dates.
- Select the best flights that fit within their budgets.
- Pay for their flights.
- Then receive a boarding pass to be able to get on their flights.

Each one of these is a sequence of events that represents a physical calendar for travel dates, cash for paying for a flight and a piece of paper for a boarding pass. Obviously, the modern travel site can virtualize each one of these steps. Understanding the physical world facilitates the comprehension of the virtualized world. Use cases can help one understand this mental model since the use cases focus not only on the flow of events but on the interplay of the actor with the system through the story that is being told in the use case description. The simpler the mental model, the easier it is to understand the system Bittner and Spence 2008.

4.3.7 Use Case Structure

The structure of a use case typically is described with 10% of the use case as a diagram and 90% as the use case description Bittner and Spence 2008. This does not mean that all use cases need to be long and involved. If a given use case requires very few features and a few flow of events, then it would have a simple use case. On the other hand, if a use case has multiple flows of events and has many

steps involved then it would make sense that the use case would have more complexity. Use cases must represent all functional requirements. However, it is not necessary for all use cases to be treated to the same level of depth. Use cases can be represented in as few as one page to as many as thirty pages each. Use cases only need to be detailed enough so that the given functionality can be developed and tested Bittner and Spence 2008.

4.4 Problem Analysis

There are three main problems that use cases help to solve.

There is an elaboration problem. This happens when a user is not fully aware of issues or problems until the problem domain is discussed further. Issues can be uncovered such as legal issues, or environmental issues that are not uncovered until elaboration has taken place.

There is a clarification problem. During the elicitation process the features are clarified. Then users will decide to change their minds upon more reflection. These changing ideas can cause problems as they need to be synched with the other already existing problems.

There is an evolution problem. This happens when a user is seemingly satisfied with a given solution but needs to change their mind because there have been new market conditions or changes in technology. These shifts signal that the old solution will cause problems that need to be solved.

4.5 Economic Context

Designers can understand the socio-economic environment by looking at the constraints on the system. Constraints limit what can be done by the system. One example of this is the regulatory framework that the system is placed in. If there happen to be laws such as gambling, in which it is illegal in a state or a country, then a gambling app is a non-starter despite the overall need for the app. Another example would be what would happen if a development team was required to use either an out of date or future technology that has yet to be widely accepted. These constraints cause issues because they make it difficult for the technical team to either learn these technologies or have any available support for the tools that they are required to use. Another major constraint could be how quickly the project is required to be done. This could be highly disruptive in both the quality of the system created and in the efforts of the development team that must develop the product Bittner and Spence 2008. Recording the constraints that the system faces is an important part of the vision document.

Once it is determined that the constraints are not too restrictive and that there is an economic environment favorable to the system that is being built, it is important to attack the problem itself. A serious analysis is required to determine if this can be built.

4.5.1 Actual Need

Software engineers should not build for its sake and the software should not be built unless it is truly needed. The main reason for this is that unless this product

is truly going to solve a problem in a way that the intended users need, then it will likely go unused. With every new software product, there will always be a learning curve that is required to begin using it. This product must be made aware to the end user, and the end user has to make a conscious effort to integrate it into their lives. The point being is that the product has to overcome enough internal user inertia so that the benefits of changing behavior will outweigh the effort to change Bittner and Spence 2008.

4.5.2 Actual Problem

Designers can attack the problem in many different avenues. Focusing on usability will help, plus marketing, tutorials, and price. However, the most help is to create a system that will solve the problem at hand. Therefore, we should ask four questions.

- Is there a problem here that needs to be solved?
- What are all the different ways to solve this problem that is not software based?
- Is software one of those solutions?
- If so, then after a competitive analysis, will your solution solve the user inertia problem?

Focusing on the first question will force all stakeholders to gain a clear understanding of what the actual needs are and pull them away from the fantasy of the brilliance of the solution. Solutions are exciting, and problems are boring and irritating otherwise, they would not be problems.

4.5.3 Importance of Non-Software Based Solutions.

Once all the stakeholders agree that a problem needs to be solved it is important to try and determine all the different ways to solve this problem that is not software based. Obviously, thinking about a solution that is not software based seems counter intuitive since one is trying to build a software solution. However, there are two important reasons to think about it this way Bittner and Spence 2008.

The first reason is that users and stakeholders are likely solving this problem already, albeit likely in an inefficient manner. Learning what people are currently doing to solve a particular problem will allow stakeholders to know what they are up against. The stakeholders will be able to know how inefficiently or efficiently the problem is being solved. Obviously, if the efficiency of solving the problem is greater than the user's inertia to learn the software solution then it is likely that they will not be using the system. Beyond understanding what is currently being done it is also very important to know what non-software solutions can be figured out as well. The reason for this is that using a software solution requires setup steps. In order to use a software solution, one must log into a phone or computer. If a current solution is non-software based, then logging into the phone or the computer is going to be foreign to their current solution. With that being said, if a software solution exposes some change in perception or some tweak that a user can apply to their non-software solution then the user's internal inertia drops and using the software becomes palatable. Rafique et al. 2012.

The second reason that learning all non-software solutions has benefits is that software should imitate what is already being done but in a more efficient manner.

If it is the case that the most efficient possible non-software solution requires a lot of time and effort and these tasks can be automated, then a software solution is a likely candidate. If it is a likely candidate, then software must be designed to mimic those tasks.

This then leads into the third question, and that is to determine if software is indeed one of those solutions. As said before software should be a solution only if it is able to solve two main problems. The first is that it solves the user's learnability problem and the second is that it drastically improves on the most efficient non-software solution or at least improves enough to solve the first problem.

4.6 Problem Domain Importance

Before a team spends ample time in the solution domain, time should be spent in the problem domain. The amount of time spent in the problem domain should be long enough to determine if the solution domain is even warranted.

4.6.1 Problem Tools

Some important tools in analyzing the problem domain are to create a problem statement and require all stakeholders to agree on it Bittner and Spence 2008. Thus, this is another example of why stakeholders need to be directly involved in the process. They workout problem analysis and if they do not agree on the problem but are instead only interested in the solution, then it is highly likely that the solution that they intend will not solve the problem that they intended.

4.6.2 Problem Scope

Another important aspect to understanding the problem is to understand its scope. The scope of the problem encompasses more than just the users of the system. Bittner points out that each of these stakeholders has needs that directly creates problems. Unless these needs are understood, then the scope of the problem is not understood. The scope of the problem encompasses all of the needs. Bittner's suggestion to determine the scope of the problem is to have each of the stakeholders describe their needs in "solution-independent fashion". Each of the stakeholders must describe each of the needs that address the root causes and why they are important. The root cause is the only thing that is important since this is a problem-oriented task and not a solution oriented task Bittner and Spence 2008.

4.6.3 Moscow Method

It is important to list all of the needs and then utilize the Moscow method to rank the importance of each of these needs. The Moscow method is detailed below:

- Must have (Mo)
- Should have (S)
- Could have (Co)
- Want to have but will not have this time around (W)

Using the Moscow method will facilitate prioritization of all the needs. This will help to determine what will actually be built within the development constraints of the system Bittner and Spence 2008.

Another important piece is to be aware of the solutions that the stakeholders are expecting Bittner and Spence 2008. There are many ways to solve a problem and often there can be two competing yet equally effective solutions. When this is the case, it is important to have the stakeholders involved so that the solution is acceptable.

A misconception is that the use cases and the supplementary docs are enough to build a viable solution. This is not the case, and it is likely to have a very negative financial impact if this assumption has been taken. One can develop a given solution that represents what all stakeholders intend and the system could work flawlessly with all the essential nonfunctional requirements in place and working correctly but still end up with a system that has some exterior factors that make the system useless. If the economic climate makes the system unaffordable, then there will be no users to use the system. If there is a government law that does not allow the solution to even exist, then there will be no users. If the technology that the user is required to have is either outdated or does not exist for them, the system is not likely to have any users. It is important to understand this climate so that one can truly build a viable solution Bittner and Spence 2008.

An important question to answer is, do use cases allow one to find a solution to a problem or do they allow one to discover a problem? It seems that if one were a customer who wished to have a solution built and this customer came up with an exhaustive list of features to be built then it would stand to reason that this customer must know what the problem is that they are trying to solve. However, this is not usually the case. A stakeholder's idea a problem can quickly generate into a list of a thousand features to solve this problem. Since a large feature list that is expressed with little context generally creates confusion, it is easily concluded that

the underlying problem can get buried and distorted within this massive feature list. So by organizing and sequencing the feature list along with the flow of events in story form, the use cases can begin to illuminate the problem that is trying to be solved. Creating stories that represent flows of events can generate questions for the customer to try to further question what is supposed to be built. Shared understanding is one of the greatest values that use cases provide and is the inevitable result of the synthetic technique that is use case modeling Bittner and Spence 2008.

It is important to remember that the entire point of use cases is to generate understanding amongst stakeholders. They must not be overly complex and should be as simple as possible Bittner and Spence 2008. They do not have to be perfect, but one should make sure to write things down Bittner and Spence 2008.

4.6.4 Use Case Flow of Events

To gain a greater understanding of use cases it is beneficial to review the important parts of a use case from a more fundamental level. For instance, use cases are a formal technique Bittner and Spence 2008. This means that use cases are made up of a specific technique and have a specific way in which they are put together. Use cases consist of three types of flows. The basic flow, the alternative flow, and the sub-flow Bittner and Spence 2008. Use cases consist of the use case model to express the functional requirements and the supplementary documents to express the nonfunctional requirements.

In expressing the functional requirements in a formal way, it means that you will express the flow of events exhaustively. However, if there happens to be a case where formal methods are not necessarily required, then an informal approach can

be taken. This approach would focus on only those methods that have the greatest impact on the use case Bittner and Spence 2008.

One of the most important aspects of a use case is the flow of events. The flow of events are where the story takes place between the actor and the system Bittner and Spence 2008. Like any two stories it is equally possible to get from the beginning to the end in drastically different ways, so is the case with use cases. In use cases, it is important to first layout the "happy path" communication that an actor and system will take. After this has been laid out, the next steps are to define the alternative flows. A use case encompasses all of the flows.

4.6.5 Vision Document

Use cases are typically a high-level document that spans both requirements and design. However, they are not something that a development team can go straight to creating. The reason for this is that use cases can often be too detailed for all of the given stakeholders to participate in actively. All stakeholders encompass a vast array of technical understanding and beliefs as to what the system should be when it is finished. Due to this disparity, developing an understanding that everyone can accept and agree upon can be a challenge. The solution to getting all the stakeholders on the same page is to create a vision document Bittner and Spence 2008.

A vision document is the culmination of all the documentation that is required so that every stakeholder has a basic understanding of what the system is trying to do. This document is very high level and it focuses on what needs to be done and not how to do it and why it needs to be done which helps establish a bedrock for

each of the stakeholders. The goal is that they agree, that the system is on track to meet their goals, and the system is something that truly needs to be built Bittner and Spence 2008.

What needs to be done to have a solid vision document? The first thing that needs to be done is to define who the stakeholders are. Stakeholders are an important part of the development process because they are the source of all the requirements of the system that is being built Bittner and Spence 2008. Without having the stakeholders in the process early and throughout the development process it is likely that the system that is being built will only either partially solve the problem or not solve the needs at all.

4.6.6 Stakeholders

If one were to think about what a stakeholder is, it would be very likely that they would think about those who use the system once it is built, but this a too narrow of an understanding. Bittner defines a stakeholder as – "An individual who is materially affected by the outcome of the system or the project(s) producing the system" Bittner and Spence 2008. It is true that users are definitely materially affected by the system but so is a vast array of other people who are also materially affected by the outcome of the system. For instance, requirements engineers need to know are who is sponsoring the system, such as "business managers, financiers, shareholders, champions, department heads, sellers, marketers, steering committee members" Bittner and Spence 2008. Albeit these people are not directly affected but they are essential in supporting the system and having the system reflect the business needs that are required. The development team is another im-

portant stakeholder of the system and they encompass not only the developers of the system but also the "Project Managers, System Maintainers, testers, support staff, designers, technical writers, production staff" Bittner and Spence 2008. Two other main stakeholders are the authorities and customers of the system Bittner and Spence 2008.

If one were to try to determine who is materially affected by a particular system one could easily see that stakeholders would vary drastically from one solution to another. Important questions and answers must be gathered to determine whom the actual stakeholders are going to be. Stakeholders can be further defined based upon the answers that were recorded. Below are some examples that are meant to specifically define the person that is materially affected by this system.

- Who will be affected by the success or failure of the system?
- Who are the users of the system?
- Who is the economic buyer of the system?
- Who is the sponsor of the development?
- Who else will be affected by the outputs that the system produces?
- Who will evaluate and sign off on the system when it is delivered and deployed?
- Are there any other internal or external users of the system? Bittner and Spence 2008

Identifying the stakeholders is very important but clearly one could not possibly ask everyone who is a stakeholder to be a part of the development process due to the sheer numbers of stakeholders a system would pertain to. Therefore, it is important to find a stakeholder who will represent the identified stakeholders in the

previous step. Ideally, these representatives will be available for the duration of the development process. This means they will be able to:

1. Faithfully represent the views of their stakeholders
2. Take an active role in the project
3. Participate in requirements and other project reviews
4. Participate in the assessment and verification of the product
5. Attend workshops and meetings
6. Do independent research
7. Champion the project to the stakeholders they represent Bittner and Spence 2008

Once the requirements team knows whom they will need to represent, it is not necessarily the case that those who will represent the stakeholders are going to be a good pick. Someone who is a bad pick can very easily derail a project and make it tough to have a system that will meet goals of the stakeholders that were defined in the vision document. Typically having a small group of people of say 2 to 5 who are committed throughout the duration of the project is a very effective way to get the continual feedback that is required to develop a project that truly meets the needs of all the stakeholders Bittner and Spence 2008.

4.6.7 User Thoughts

Who knows what the user wants? There are two sides. To try and query the user to get their requirements or give the user what they want based upon the expertise of the people creating the product.

The first approach assumes that the user knows what they want. If this is the case, then creating a usability design would be a simple matter of gathering their understanding of the product and then delivering what they say that they want. If for some reason they have fickle sentiments or an incomplete understanding, it is quite logical that the result of what is produced will be incomplete. Also depending upon the size of the user base it may be difficult to get all the sentiments of the users.

Chapter 5

TASK MODELING

Task modeling is another form of requirements engineering that spans into design. While use cases are mostly textual, task models are mostly diagrams that depict a breakdown of tasks that will achieve a given goal. A diagram that is used to depict a task model is known as a concur task tree, which is displayed below.

Figure 17. Example of a Task Model in the CTT Notation
"W3-taskmodel" n.d.

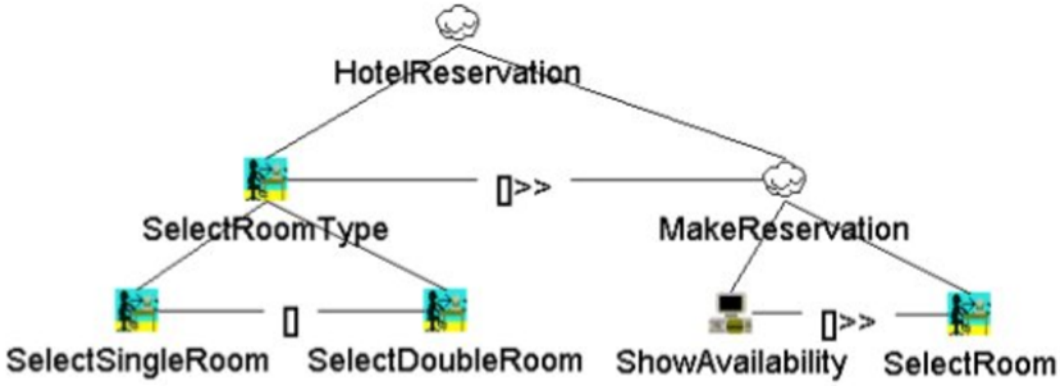


Figure 1. Example of a Task Model in the CTT Notation

Task models are a part of a larger area known as task analysis. Task analysis is an area that attempts to detail HCI – human-computer interaction problems with the primary focus being that of work performance Diaper and Stanton 2004. There are two types of entities that task modeling produces. These are things and the relationships between these things. Things are subdivided into tangible and intangible. Tangible things are those that are a part of the physical world such as computers and keyboards and intangible are those things that are a part of the mental world. A concur task tree can be considered an intangible thing. It is impor-

tant to note that the central concept of task modeling is the performance of work Diaper and Stanton 2004.

Task models are used to detail and outline all of the steps that a given user will undertake to achieve a given goal Diaper and Stanton 2004. "Task models are used to improve the understanding of how users may interact with a given user interface for carrying out a particular interactive task" Stanton and Young 1998.

There are different types of task models. Some of the major task models are hierarchical task analysis; goals, operators, methods, and selection rules (GOMS), groupware task analysis, concur task trees, methode'analytique de description de t^aches (MAD), task knowledge structure, diane+, method for usability engineering, and task object-oriented description. Each of these modeling methods is meant to do the same thing which is model a user's behavior Diaper and Stanton 2004.

Use cases and task models differ because each has different scopes. Task models are more detailed and rigid. It is difficult to have a dialog with a stakeholder because the task models are rigid and each model requires a learning curve to understand it. This knowledge is not available to all stakeholders. Since use cases are mainly text, they lead to more elaboration because everyone understands natural language. Use cases can be edited, revised and understood by all stakeholders. As a result, use cases tend to straddle software requirements and software design whereas task models tend to be closer to detailed design and UI design. Another reason for is because task modeling refers to human-computer-interaction and use cases generically refer to actors and systems meaning that actors do not have to be initiated by a human. Actors are any stakeholder type that initiates an action to the system to achieve a goal.

Chapter 6

COLLABORATION

6.1 Introduction

In software development and the corporate world, different functions are often performed by different people. Broad functions throughout the entire software development cycle are often split up into teams. There are sales teams, development teams, integration teams, implementation teams, support teams, system operation teams, network operation teams, database teams, account managers, plus many others. Each team performs a crucial role in the creation, deployment and maintenance of software.

There is a tendency that since each group plays an important role, they can see their specific role as the most important in the software cycle. The sales team thinks that if it were not for them, there would be no product for anyone to use. The development team feels like if they don't create the product, there would be no product for the sales team to sell. The integration team feels that if it weren't for them, customers would not be able to integrate their products. The account managers feel that if it weren't for them, there would be no one to build a long-term relationship with the customers. These self-important justifications, while possessing some validity, fail to see that the whole picture of each team plays a crucial role in the outcome of having a successful project. Therefore, it is vital that different teams should have a say in the initial creation of a product. If they collaborate in the beginning, their team's interests will reflect at the outset.

However, this narrow and insular view are not all bad, provided it is exploited in the right way. When one sees their job, and by extension their team as being overly important, it is likely they are arguing the positives of their position, and if pushed, the necessary positives of their position. In the same vein, it is likely they are also pointing out the negatives, and if pushed, the absolute negatives of every other team's jobs. Although their viewpoint is distorted and likely exaggerated, it can offer valuable information when coming to a conclusion on how software can go from an idea to implementation in the most thorough way possible.

Now let us explore different ways in which these teams can collaborate.

6.2 Collaboration Methods

There are many ways to collaborate. Teams can collaborate online with text and video using such technologies as GoToMeeting, Webex, Skype video, Google Hangouts or over the phone without video. These are powerful forms of synchronous communication. There are also asynchronous ways for teams to collaborate. Some of these are email, Skype, Jabber, along with many document sharing solutions. Some of these solutions are File Cloud, DropBox, Google Drive, and Huddle. These are powerful because anyone can access the documents from anywhere and the file sizes are not as limited as they are with email.

Let us take a brief moment and talk about asynchronous and synchronous communication.

6.2.1 Asynchronous and Synchronous Communication

Another important part of collaboration is how the collaboration occurs. Two of these ways are synchronous and asynchronous forms of communication. Asynchronous solutions are valuable in that they allow for different people to communicate at times that are convenient to their workflow. This is true for both the sender and the receiver. Synchronous solutions are valuable in that they allow for people to not only communicate in real-time but it also allows them to discern intent which enables problems to be worked through more efficiently. Often synchronous solutions are employed when asynchronous solutions are not working or become too time-consuming. Let us explore some of the advantages and disadvantages of these two forms of communication.

Synchronous collaboration has the advantage of being able to resolve issues by quickly talking them out. Decisions can be made quickly, and information can be gathered from other members easily. Sometimes, problems need data from more than one source to come to a solution. Having the right people on a call to provide the data that is required and also to perform certain tasks to be able to supply further data for a given solution can be valuable. Some problems can be solved this way quickly.

However, there can be significant downsides to synchronous collaboration. The downside of synchronous communication is a lack of documentation that is created. Often meeting notes are not taken, and actionable items will sometimes get lost unless another member has some incentive to push for it. Another negative is that schedules can conflict and sometimes getting together can be a chore because each meeting can have a different importance for each member. A meeting that

does not have a great deal of importance can only get partial attention by some members that join. Verbal agreements do not have the same binding as do written agreements and therefore assigned tasks do not necessarily have the same follow through. There is also an inefficiency to synchronous collaboration. Synchronous collaboration is very efficient for those who are involved in a particular problem but it can be very inefficient for those who are in the meeting but are only needed for a small portion of it. What happens in this scenario is that there is a large chunk of that person's day that is not productive. Also, synchronous communication only works when everyone is prepared to solve the problem that the meeting is meant to solve. This does not only mean that people need to be prepared for the meeting, but there are also times in which issues arise during the call that people were not prepared for prior to the meeting.

Asynchronous communication has many upsides. One of the upsides of asynchronous communication is that work can be done in the order that is most efficient for the stakeholder who has to perform that work. This means that it might make some sense to start one project and then if one gets stuck one can start another, or even one can do the work that's required at the time of the day that best suits that stakeholder. This flexibility in schedule for the stakeholder is very appealing. For some people, it may take longer to think about the appropriate response to someone's communication than if they were pressured to do it synchronously by being in a meeting. This is important because use cases themselves are mostly written documents. As has been said earlier in this thesis, diagrams make up a very small part of an actual use case.

However, there are disadvantages to asynchronous communication. One disadvantage to asynchronous communication is a question of priorities. It may be dif-

ficult to prioritize which projects should be completed at what time. Synchronous communication, such as a meeting, forces those participants in the meeting to spend a specific amount of time addressing the issue at hand.

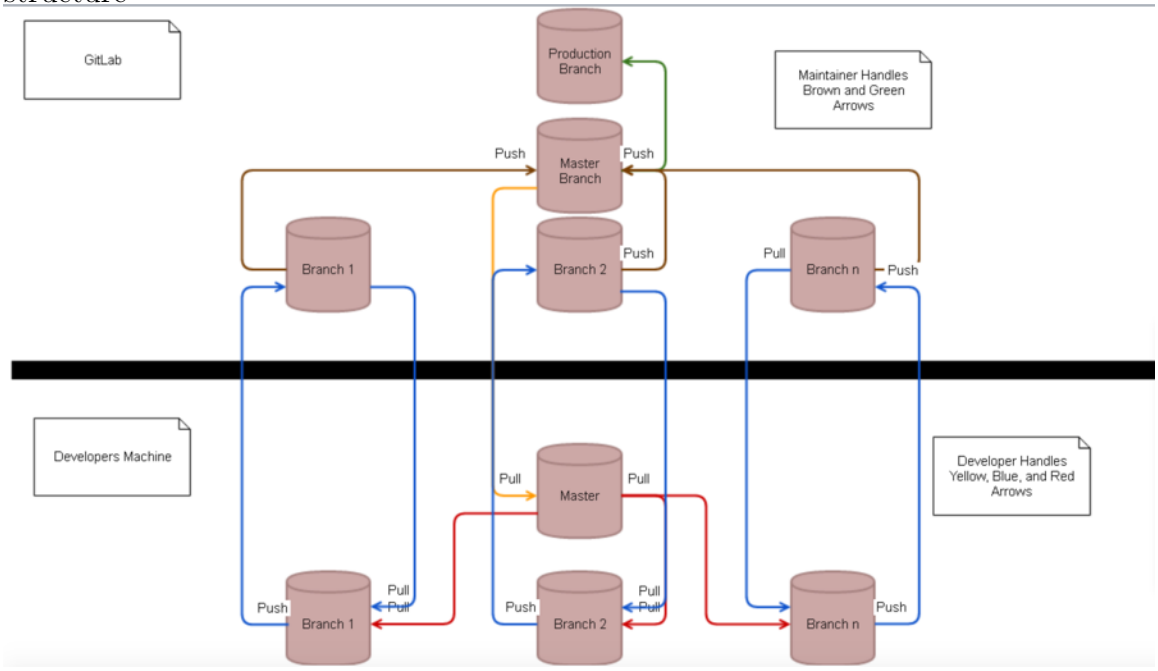
6.3 Git and Git Workflows

Git is a distributed version control system that allows for sharing and maintaining of a common code base. Distributed version control means that the entire history of a given code base is stored on any developer's machine that happens to clone or pull down the current project. This is contrasted with a version control system such as svn which requires a central repository to hold the given code base. Since git does not require a server to store the code base, there are two ways to which developers can share code. The first way is that git can use a centralized repository simply for backing up the code and handling access rights. The second way is to use a peer to peer approach in which all push and pulls are made directly between each developer's computer using ssh. The most popular form of using a central repository is using Github. Github's purpose is simply to establish a repository that promotes social coding with the underlying idea that collaborative development is better than singular development. How this collaboration occurs is more of a social contract than a technical requirement. This is true because of the flexibility that git offers. The rules in which people ultimately decide to share the code are entirely up to them. However, below is a diagram that describes a typical workflow using Git.

Since git is distributed and the entire history of a given project is loaded on the developer's system, any version of a given file can be recreated. All changes to

given project are saved and any developer can revert to anyone of them. Subversion has a central repository and it does not share a project's entire history. This means that any changes that are made in the code base that needs to be committed must be made over a network, so there is often latency when done this way and the given restriction that a developer cannot make a commit on their local computer in the same sense as git. Git allows for the commits to be made on a local file system and when the repository is pushed to the central repository, the entire commit history is preserved. If one chooses to make local commits on their computer using svn's front end then when the files are finally pushed to the central repository all of the local commits get squashed to one large commit and from svn's point of view there was only one commit.

Figure 18. This diagram gives a high-level overview of a basic git repository structure



At the top of the diagram, there is the role of the maintainer. Typically, the

maintainer is the project owner and is the one who makes the final decisions on which code is merged to the master branch and what code is not. The maintainer maintains the master branch and the production branch. A branch is complete when a feature has been finished and tested. A developer can now create a merge request. Even in this scenario, it is still the responsibility of the maintainer to run all tests prior to committing any code to the server by testing, accepting, and merging the code to the master branch.

An alternative approach for merging finished code into the master branch is to not have a maintainer at all. This is often the case for dev ops and continuous integration. These topics are out of the scope of this thesis but I bring it up to show that there can be great flexibility with the git workflow and to stress that the workflow is more social than technical. However, dev ops and continuous integration is basically an environment where when the code is submitted, the code is run through various tests and then is deployed into production. If the code is good, then it moves to production and is controlled by a concept as a feature toggle. If the code fails the tests, it is sent back to the developer who fixes it and then the developer pushes it through until the code is run without errors and is pushed to production.

The master branch serves as a branch for all the features for a specific release plus all of the features of the prior releases. It is also feasible to have more than one master branch serving a particular release that is meant to remain unchanged. For example, you could have a 1.0 branch that remains unchanged while a 2.0 master branch is actively worked on.

The production branch stores the latest release and the master branch merges its code into it. This code, by and large, is meant to be untouched unless there

are any hotfixes that need to be changed. A developer should not have access to either the master branch or the production branch. When one has direct access to the code base, one has a very powerful tool available to them and the production environment is too sensitive to an environment for mistakes to occur. However, mistakes do occur and sometimes the production branch should be accessed. It is important that only bugs are fixed and no new code is introduced that is not directly related to the bug at hand.

The developers will follow two simple rules. The first rule is that they can only pull from the master branch and never push to it. This is because only the maintainer can push to the master once the feature has been accepted. If a developer pushes directly to the master branch, the branch will be polluted with unaccepted code and that code may or may not be in the current release. This can throw off the other developers because they may be working off of different assumptions about the code base. The second rule is that they can only push to the current branch that they are working on. Each of these branches that a developer is working on should be reflected on the server. This means that if a developer is working on branch A on the developer's local machine there will be a branch on the server that is also labeled branch A. This is a typical collaborative workflow.

6.3.1 Resistance to Use

If it is the case that git is such a great solution, then why is there often resistance to its use for sharing documents? The main reason is a learnability and psychological one. Overcoming a learning curve is only worth it to someone if the difficulty of learning, it is less than the benefits that the new technology will offer.

Since the most common way for people to share documents is through email, the concepts of git push, git pull, branching, merging will be a challenge. This is also true for designers because even though they design software, it does not mean they know how to use version control at that level.

Software adoption is greatly aided by taking real world processes and mimicking them in the software. This was addressed earlier in the thesis when talking about the mental model. In the physical world if I was working on a design and I wanted someone else to work on the design then I would simply hand the design to them to work on and review. As such, email makes a lot of sense. A designer works on a design, then the designer puts the design in an email and then it is electronically handed over to the person the designer wishes to get input from. It follows a mental model that makes sense. As was described earlier this is not the case when using git. Git requires one to realize that any changes that are made are recorded as their own layer. When a design is sent over email after changes have been made, then only the design is sent over with potentially very limited versioning attached to the design.

6.4 Existing Tools and Methodologies

6.4.1 Github

Github is the foremost website for social coding using git. Github was founded by Chris Wanstrath, PJ Hyett, and Tom Preston-Werner in on February 8, 2008, and as of September 25, 2016 Github has over 38 million repositories and over 15 million users n.d. Github is extremely popular because its goal was to not only

host code but also to create an environment where social collaboration was easy. Issues can be raised by developers for other developers to discuss. The projects host can have a wiki to write a description and how to use it. Developers can communicate with each other by working on a piece of code and then when it is finished they can create a merge request so that the project's maintainer can merge the completed code.

For example, let's look at the laravel github page. Laravel is a PHP web MVC framework that is one of the most popular PHP projects on Github. There are so many projects on Github that it is tough to determine which projects one would want to use for your project. The best projects to choose from are those that have a lot of contributors and recent commits. Laravel boasts 365 contributors, 5,088 commits, and a recent commit as early as four days ago

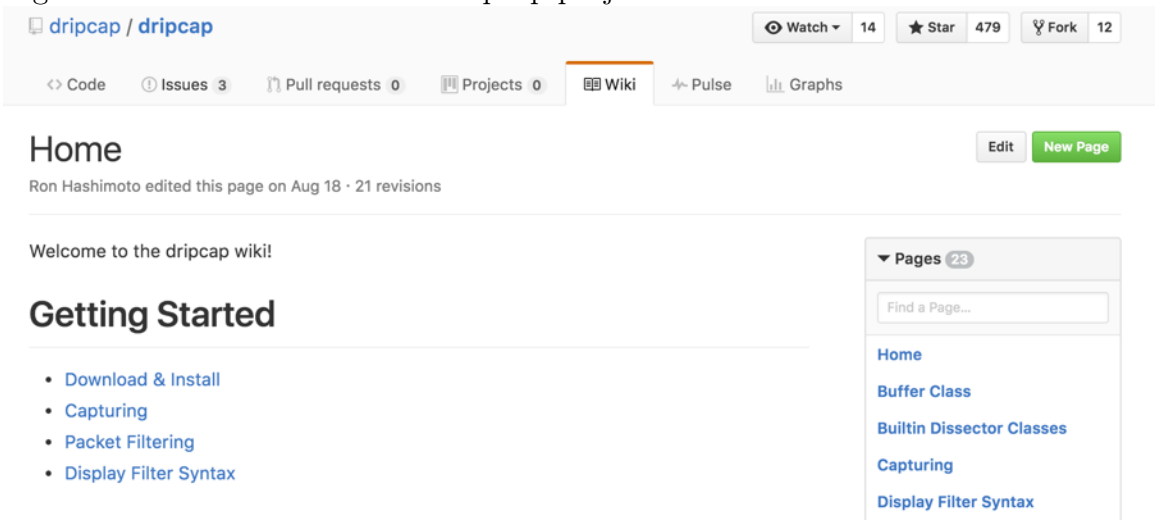
Figure 19. Shows the number of commits and contributors to the laravel project on github

n.d.

Laravel has an official site for its documentation and does not use a wiki but dripcap a packet analyzer project uses a wiki to show people how to get started and begin to use it.

Github is a powerful tool that is used by millions but it is mainly used for low-level coding and it is not used much for natural language. For our purposes, we need a tool that will allow for the sharing of use cases which deal with natural language.

Figure 20. Shows a wiki for the dripcap project



There are a few projects on GitHub that encompass natural language, but even those require a bit of technical knowledge that is out of scope for most stakeholders. Github requires a knowledge of git and its concepts to be able to use it.

One such project that is on Github is the HoTT book that was created by many mathematicians. The project currently has 66 contributors Program 2013. Since this is a math book, latex is used to create the chapters. Even though they have a website that gives you links for learning git and learning latex Program 2013, both of these technologies require some technical knowledge to implement correctly. For someone who wants to contribute to this book and who has does not have any technology background, it will be a challenge to contribute. There is a tool that uses git at a higher level.

6.4.2 Penflip

Penflip is an interesting tool in that it tries to use the power of git and use it for natural language. It is meant to be used like GitHub but for writers. It was

created by Loren Burton and he describes it in great detail in 2013 while it was still in its early stages n.d. He describes how he wants to use the power of git and the GitHub workflow to create a site that nontechnical writers can use to collaborate and version control their work. He has both free and paid versions of private writings. This tool offers a great way to bring writers who are nontechnical minded and be able to combine the power of git.

However, a downside is that all the code is being required to be written in markdown. Markdown is an abstraction of HTML. It is a way to write HTML in a more descriptive way. However, this library still requires people to write using markup that is a WYSIWYM instead of something that is like Microsoft word that is a WYSIWYG. Most writers are used to WYSIWYG. When one writes with paper that is what is happening and if one were to recreate this for the computer then it should be as close to real paper as possible. Another downside to Penflip is that it does not offer any time management functionality to a given use case. It also does not offer a mechanism to hand a design off to another stakeholder when it is completed by a prior stakeholder.

6.4.3 Google Docs

Another tool out there for working on documents in a collaborative way is Google Docs. This is powerful in that it allows many people to work on a given document at the same time. The idea is that one can contribute as if one were using a word document that is on their desktop even though it is hoste no the cloud. People can contribute using a familiar interface that does not require learning any code or managing any branches. Google Docs also is version controlled. One can

go to a prior version of a document and see the changes that were made by other contributors.

Using Google Docs is such a powerful tool that it nearly meets the requirements of the tool that is outlined in this thesis. However, there are a couple of reasons as to why Google docs are insufficient. One of the main reasons why it is insufficient is that it does not put any time constraints on the completion of the document. There are no notifications around impending deadlines. A reason for this is because of the second reason why Google docs are insufficient for our needs, and that is because it is too general. Google Docs is meant to be a collaborative tool for any document that a team wishes to create. It is not tailored to creating use cases and as such it does not offer the important management tools that are inherent in this use case tool.

6.4.4 Existing Tools Comparison

These three tools are very close to the functionality that would be required to implement a collaborative use case tool. The least appropriate tool was shown first the most appropriate tool was shown last. As has been pointed out in each of the summaries of the three tools they all suffer from not having management capabilities and being too general. This means that each one of them does not have any time dictated notifications to remind a given stakeholder to finish their contributions on time.

They do however offer many of the features that are valuable for the use case tool that has been created. For instance, they all offer version control to varying degrees. Penflip offers a much more user-friendly version of GitHub but it suffers

from still having to use a markup language to write the documentation. Google Docs is very powerful in that it is a version control word processor on the web. The only downside being that it is too general and does not offer the time management tools of this use case tool.

So, with all of this being said one can see that a use case tool could be very beneficial for creating use cases across stakeholders. This tool is unique because users will use it both sequentially and in parallel. Version control as was outlined earlier is very powerful when it comes to asynchronous communication. This tool allows an unlimited number of people to work together on a project. Now let's explore this tool in a little more detail.

Chapter 7

PROBLEM DEFINITION

7.1 Distribution Problem

Ultimately the problem stems from the concept of a document being a container for information. From a sense that our computers are made up of many different types of files, this idea is not entirely wrong. The real world of file sharing is represented in the way these files are being shared among many people. When files are shared using version control the concept is much more nuanced.

Designers use email because it matches our mental model of sharing. If we were to create a design work product like use cases we would write the text in some word editor and then put it into an email and share it. In everyday life, we would grab a sheet of paper, write some text and then hand it to the next person to edit, so email seems natural. But this greatly inhibits collaboration because of the number of work products it is possible to create. At my job, documentation is limited to three or four people at the most.

The next obvious question is why don't we have just one use case in the cloud worked on by everyone? This idea is used by several file locking cloud-based systems. The main point to file locking is data integrity. If you have more than one person working on a document at the same time, then merging can be difficult, and data loss is possible.

However, the obvious downside to this is scalability. Having only one person work on a document limits productivity.

So why not have a cloud-based software system that is one document which more than one person can work on at the same time?

Systems like Google docs achieve this powerful feature, allowing more than one person to work on the document simultaneously, or at different times. The downside is, designers cannot have their own, independent branches. Moreover, there is a potential for data loss when multiple designers are simultaneously working on the same thing.

7.2 Flexibility Problem

Flexibility refers to how easy a design can be split apart or put together. Use cases, for example, are often manipulated. Sometimes one use case is two use cases and sometimes two use cases are too narrowly focused and should be merged into one. Having the ability to do this can be tricky if multiple versions are out there, difficult to track down.

7.3 Scalability Problem

The traditional way of sharing documents limits scalability because it limits the number of developers who can contribute. Cloud-based file locking allows only one person at a time to contribute. If the Linux project on Github were done via email, it would be an utter nightmare. Consider that the number of potential developers could be more than 1000. If each of the 1000 developers commented once for every commit to the Linux project, there would be 1000 contributions per commit. One Linux project currently online has 634,959 commits. Attempted via email, the total

number of documents generated would be 1000 times 634,959, totaling 635 million documents. Not only is it unimaginably cumbersome to work on the Linux project this way, if only one document could be worked on by one person at a time, but productivity would also screech to a halt.

Chapter 8

SOLUTION

What seems like an obvious solution to the problems of flexibility, distribution, and scalability in collaborative design is to move away from documents altogether. A further step would be to exit the mental model of the physical world and employ a more efficient collaboration method. The generation of documents would only be optionally needed and then only at the end of the process. Thinking of a design work product as simply information viewed by more than one person, one is freed up to focus on the data itself and not its container.

This thesis proposes a new methodology built around efficiently defined roles and policies, along with versioning, manipulation, and scalability. When defining roles and policies, there should be a certain amount of flexibility to change with each building block.

Across the industry, there are many job titles given to various roles in software development. For example, there are designers, solution managers, architects, development managers, quality assurance, integration engineers, systems engineers, account managers, support reps and others. Although each of these roles has a particular function in the software development lifecycle, these need not be static. In some use cases, a person will have the role of a designer while on another use case, that same person could have the role of a development manager.

There should also be some flexibility in how each use case is distributed among selected roles. In project management, there are four different types of task dependencies. The purpose of task dependencies is to determine when to begin each role.

Task dependencies are described in more detail a little later, but for now, a design building block structures the order and timing of each role. This definition would be known as the policy of the design building block.

Another important aspect of the methodology I am proposing is versioning. Versioning means that each role should be able to contribute whenever and however they choose. Because of the likelihood of disagreement among roles, the designer is usually designated as the final word on an individual use case or building block, and it should be the designer who makes the final decisions on what will ultimately be on the use case or building block. Versioning allows the designer to review all of the comments in order, pulling out key ideas from each of the roles.

Versioning must be easy to use and review. There are applications which provide versioning, but the ability to review input can be a challenge. Applications such as Word provide built-in application versioning, but it is a messy process, and it's difficult to see the different versions of a particular document. One way to make versioning easy is by using something that is specifically built for versioning, like Git. Browsing history, merging, and deleting are all seamlessly done with Git, and branching with ease is one one of its most powerful features.

A new design methodology must be able to manipulate the building block. The building block must be split up, merged, and easily edited. This means that something like Git will have to be used in a unique way. With normal use, one works on a document until a feature is completed, then the feature is committed to the branch. If the developer wants to try out something different, they can create a branch, test out their idea and merge or delete it, depending on its success. This is a pretty straightforward way to utilize version control. The new methodology I propose will take a different approach. The new methodology will have the unique

ability to use separate branches for different parts of the building block. This means that if a use case had to be split up, there would need to be a way to manage the branches under the parent design building block master branch. This way the project will still be managed as one but also treated separately.

Versioning will also have to be done differently when merging building blocks. When merging two building blocks, each building block having its use case, the master branches of each building block will have to be merged as if they were a part of the same building block. This would be like combining two separate projects, something not typically done in the normal workflow of a distributed version control system like Git.

As stated earlier, scalability is a real challenge when one is forced to use technologies such as email or file locking cloud-based solutions. A new design methodology must be able to handle scalability. Using email makes managing the documents, as they multiply exponentially, nearly impossible. Cloud-based file locking solutions inhibit scalability because only one person can work on a document at the same time. This is another place where distributed version control systems can really shine: not only the building blocks are shared but also the building blocks entire history. As a result, anyone who wishes to contribute can have a clone of the repository on their own system and simultaneously keep it up-to-date with other people's repositories. Unlike documents that are sent through email nothing multiplies on someone's computer as future changes are made.

Files in the cloud are locked to maintain data integrity. The issue is this: if person A and person B check out the same document, and they both try to commit, it would be the last commit wins scenario. When this happens, the first person's

commit will likely be lost, causing real problems. Using versioning all changes are saved, and data is not ever lost because every commit keeps the data for posterity.

The new design methodology proposed here resolves multiple documentation problems and the cloud problem. The cloud stores the design building block and each branch of all the various roles. When people sign on to a web-based solution, they can manage all of the building block design work products in the cloud. Each of the branches for each of the roles will exist in the cloud in the same way that they would exist on their machine. This solution allows for there to be seemingly one design building block that everyone is working on. That is the purpose of having one document in the cloud: combining the apparent ease of sharing one document with the ability for everyone to have their branch in the cloud as if they were working on their own repository on their machine.

The last important piece to this methodology is scalability. There must be a way in which any number of people can work on the building block simultaneously so all input can be included. In the following chapter, we will discuss a proof of concept tool for the design methodology just outlined.

Chapter 9

TOOL

9.1 Overview

Now that we have discussed the two main pillars of this thesis, different approaches to high-level requirements and collaboration, we can now begin to explore a tool which combines the two of them. This tool is a proof of concept and an example of the methodology that was described earlier.

I am introducing a design tool which demonstrates the ability to create use cases in a collaborative way. Use cases allow for high-level requirements to be translated into low-level specifications. This is often a difficult task for one person. Under ideal circumstances, it is beneficial for all the stakeholders to able to contribute to the generation of the use case. As was pointed out earlier in the thesis, different teams have different requirements for a given solution. There are certain things that one team needs to do their job that another team does not need. Each use case is a stronger solution when multiple teams contributions are reflected in it.

The focus of this design collaboration tool on five internal stakeholders, each tied to the creation of a given use case: the designer, the solution manager, the architect, the development manager and the quality assurance manager. Unlike other stakeholder groups, this tool will assume that each these team will have a stakeholder representative. The tool will be handed to just one person from each team.

The workflow of my design collaboration tool begins with the designer. The

designer is the one who initiates the use case. When the use case has been initiated, it is handed off to the solution manager. The solution manager will have a specified amount of time, during which comments are allowed for the given use case. While the solution manager is adding the relevant comments, other stakeholders can add their comments to the use case. When the use case is in the solution managers time frame, whatever the solution manager accepts or rejects will be assigned to the master. This means, once the completion date has passed, all comments from the solution manager will not be a part of the solution managers branch. They will be on the master branch.

Because there must be an enforcement of business deadlines, after the use case has been passed onto the next stakeholder (in this instance the architect), anything that the solution manager adds will not be a part of any final use case. The tool will enforce business deadlines for a given use case. Under other collaboration tools, the deadline for completing a project with multiple contributors is external to the work product. What this means is, there is someone outside of the contributors pushing them forward and reminding them when something is due. My design collaboration tool will push people to finish based upon a date predetermined by the designer. After the use case has made its rounds and the final stakeholder has made the relevant contributions, the use case is passed back to the designer. The designer is only able to see the master branch and has final say on whether a use case is complete.

Another innovation of this tool is that it eliminates inefficient means of communication. Without a tool which allows multiple people to work on the same document in version control, one is limited to traditional ways: emailing the most recent copy, having verbal meetings, online meetings, or face-to-face meetings using

post-it notes. As was covered earlier, sending documents through email makes viewing history of a given document an extraordinary chore. Some companies produce documents with the only recorded changes written at the bottom of the documentation. These notes have to be summarized by the person who made the changes. The problem is, they may not accurately reflect the actual changes that were made. Having a version control system on a document allows one to walk back through the actual changes made. This is important because if a designer is interested in what a particular stakeholder had to say, that designer could to go through the revision history and find all of that stakeholder's changes and contributions. The beauty of something like Git is that all committed changes are never actually lost.

9.1.1 Workflow

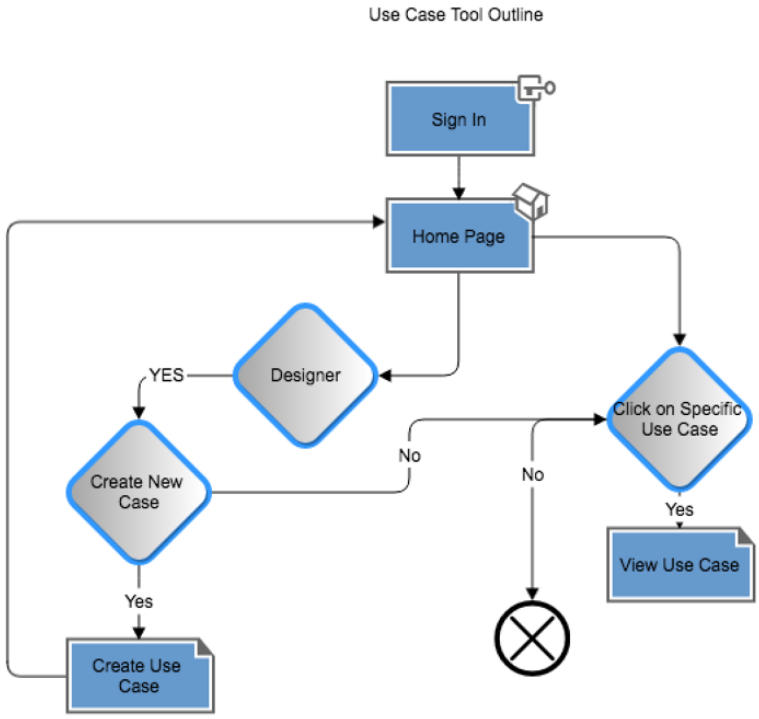
The diagram below shows the high-level workflow of this tool.

9.1.2 The Seed Case

9.2 Task Dependencies

Project management has a concept known as sequence activities. "Sequence Activities is the process of identifying and documenting relationships among the project activities". Understanding this process is important in order to achieve the greatest amount of benefit by organizing the tasks as efficiently as possible Project Management Institute 2000.

Figure 21. A process diagram for the tool

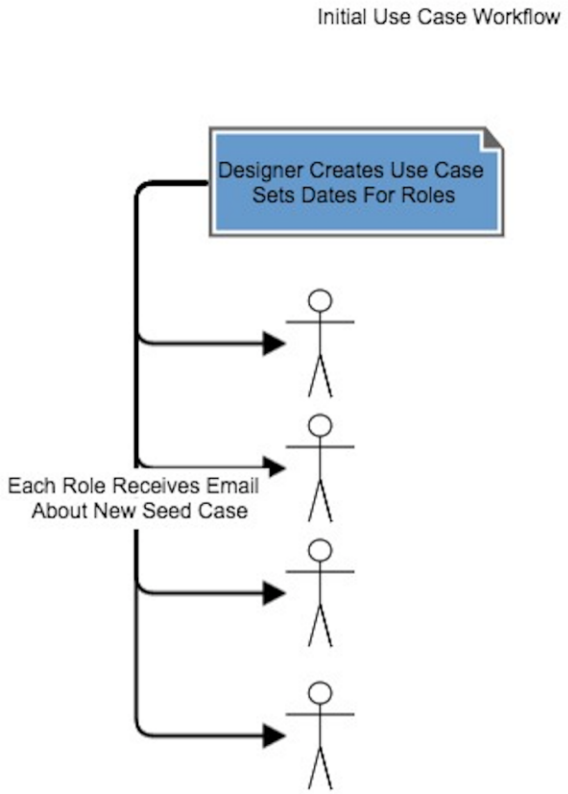


One way to organize these tasks efficiently is to use the precedence diagramming method (PDM). PDM's are used to construct a schedule model to understand the sequences of activities. Logical relationships are used to map out the activities. There are four logical relationships, sometimes referred to as task dependencies, and they are detailed below. Project Management Institute 2000.

9.2.1 Finish to Finish

The relationship here is that a person (B) cannot finish until another person (A) has finished Project Management Institute 2000. A real life example would be a manager at a restaurant cannot go home until all of the employees have finished up and are ready to leave (see diagram below).

Figure 22. Showing the beginning of the seed case



9.2.2 Start to Finish

An alternate task dependency is that a person (B) cannot finish until another activity has started Project Management Institute 2000. A real life example of this would be a runner in a relay race being unable to finish running until the baton has been handed off to the next runner (see figure below).

Figure 23. A finish to finish diagram

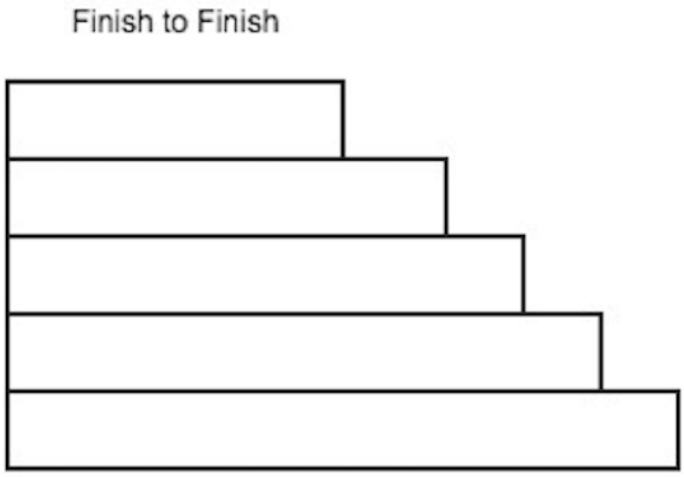
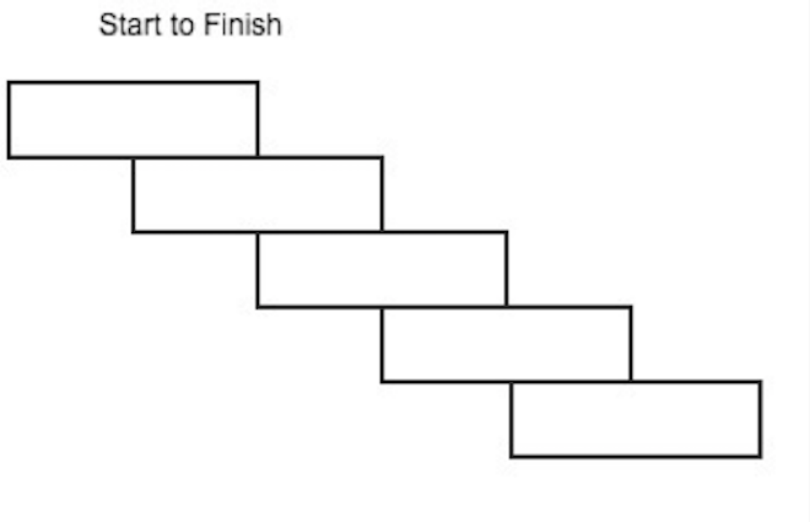


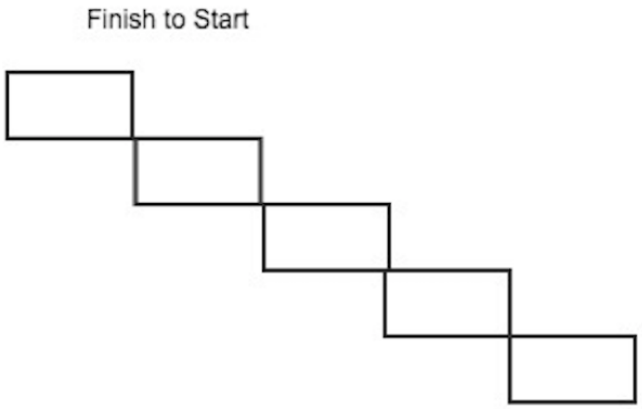
Figure 24. A Start to Start diagram



9.2.3 Finish to Start

Another task dependency is that person (B) is unable to start until a person (A) has finished. This can be visualized by someone being unable to publish a book until the book is finished Project Management Institute 2000.

Figure 25. A Finish to Start diagram



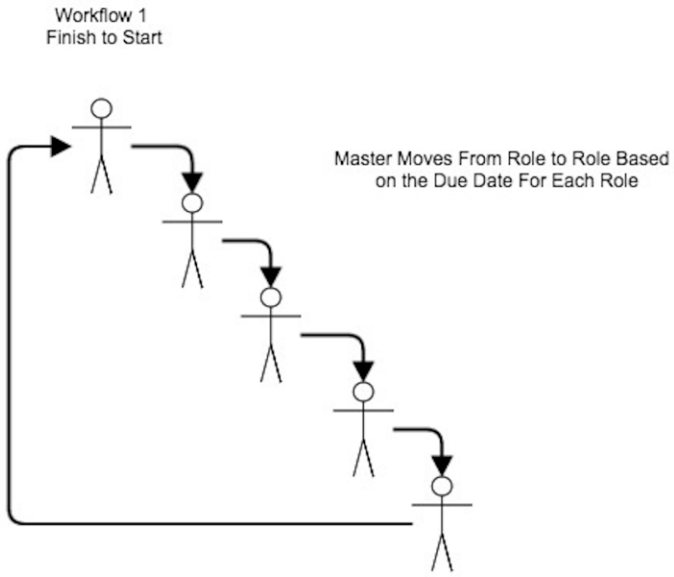
The diagram below describes the finish to start task dependency as it applies to the app. In this scenario, the master branch is only worked on by one role at a time, and each role must wait until the prior role has finished. To prevent contributors from having to wait until the last minute, this workflow is essential. It also enables the subsequent role to have more of the needed information.

9.2.4 Start to Start

Start to start is a task dependency for collaboration in which all roles can start at the same time, but they can finish at different times. An example would be any race in which everyone starts at the same time, but finish at different times. Every role depends on another role to start before they can start Project Management Institute 2000.

Below there is an example of how users in this app are utilizing the start to start task dependency. In this scenario, all four roles are unable to start until the

Figure 26. A Finish to Start diagram as is implemented in my app



designer has actually created the use case. Once they have begun they can finish at any time, but mainly after their master branch timeline has finished.

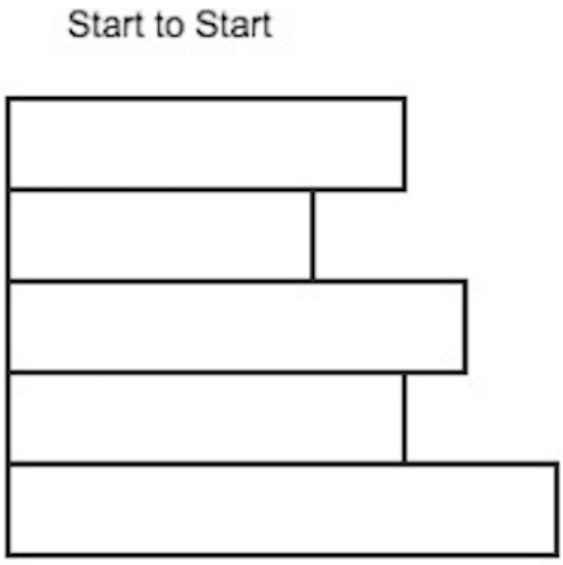
9.2.5 Three User States

Each user will go through one of three states when they work on a use case: the before, during, and after states.

The "before" state refers to the time before a role has a turn to work on the master branch. In this state, they are working on their branch. When a user works on their branch, all of their changes can be seen by all users except for the designer. The designer can only see the master branch changes.

The "during" state is when the user is working on the master branch. At this

Figure 27. A Start to Start Diagram



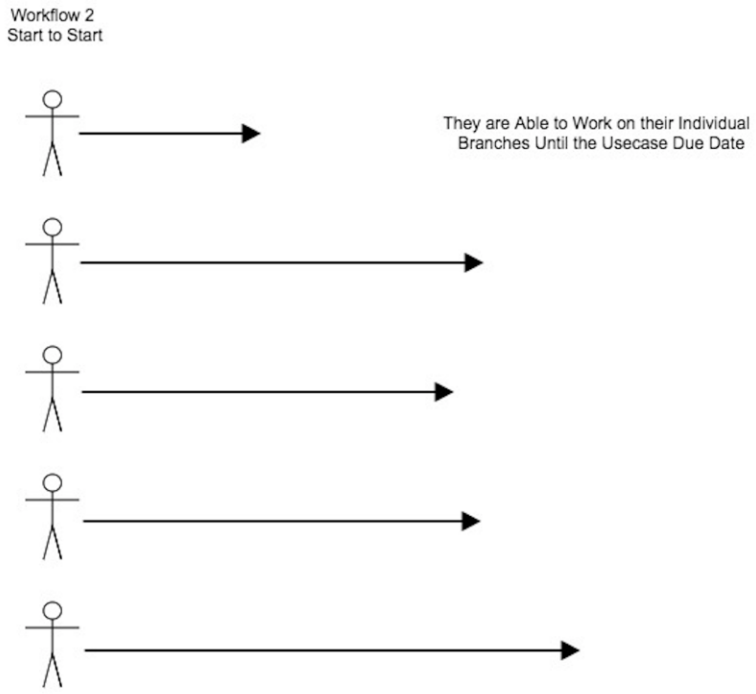
point it is important that a user finalizes as much input as possible because these are the only changes the designer will see.

The "after" state is when the users due date have passed. In this state each of the users can still contribute but they must do one of two things: they can wait until the next version to include their information, or they can use someone down the line that may have access to the master as a proxy. If the proxy user wishes, they can then make the requested changes to the master.

On the surface, this design collaboration tool is relatively straightforward. A user will register themselves and in doing so will select the role that they play.

There are five roles that a use case gets passed too. However, there is one difference which sets the designer apart from the other four roles. The designer can

Figure 28. A Start to Start diagram as implemented in my tool



create a use case, and the other four roles can only edit a use case. Once a user has been registered in the system, they will be sent to the homepage. The homepage has two different views. If a designer has logged in, they will have the ability to create a use case.

Also, the designer will see columns that are pertinent to the role. For instance, it is important for the designer to know when a use case is due and who is currently in possession of the use case.

If a stakeholder signs in under a different role, they will see columns that are pertinent to that role. As one can see from the illustration below this role has a start date and an end date.

Figure 29. The three states a role takes in the app

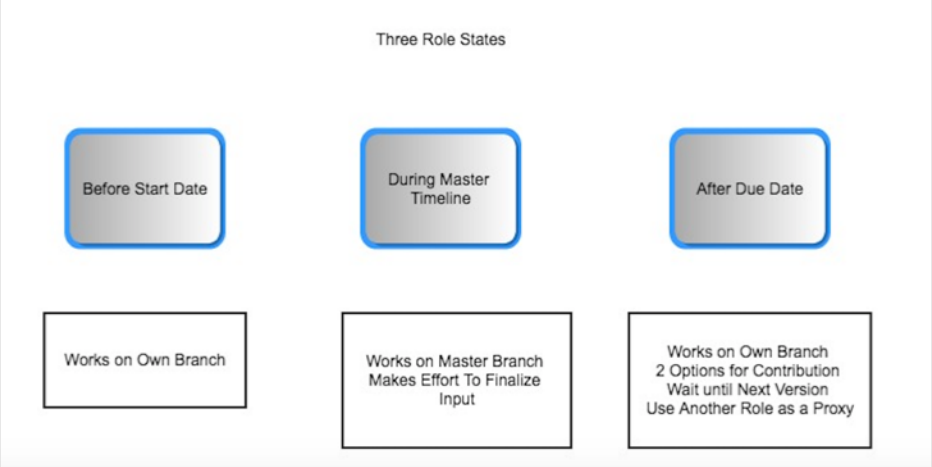
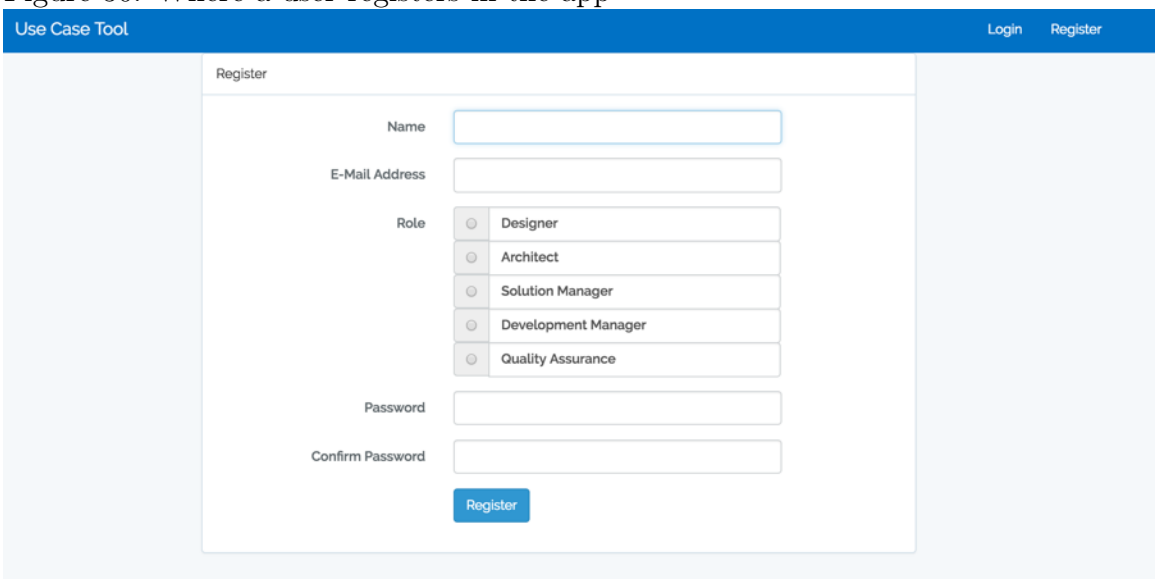


Figure 30. Where a user registers in the app



Because each role has control of the master branch only during a given period, these dates are pertinent to all the roles except the designer. Called the "sequential" part of the tool, it gives an incentive for each role to actively contribute to a given use case. Only changes merged into the master branch will ultimately be seen by the designer, meaning that for any given segment of time, each role will have the

Figure 31. Where a designer sees their use cases

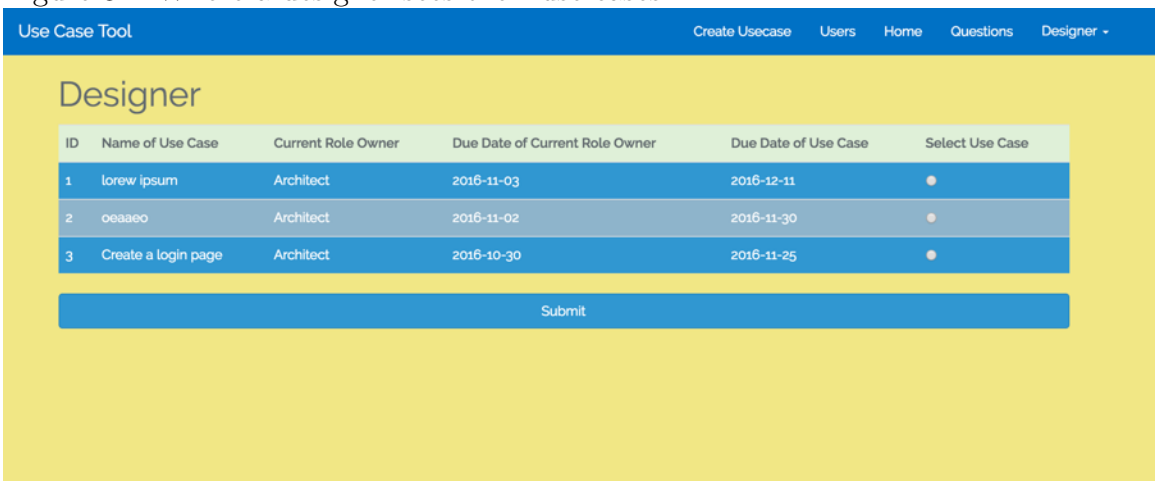
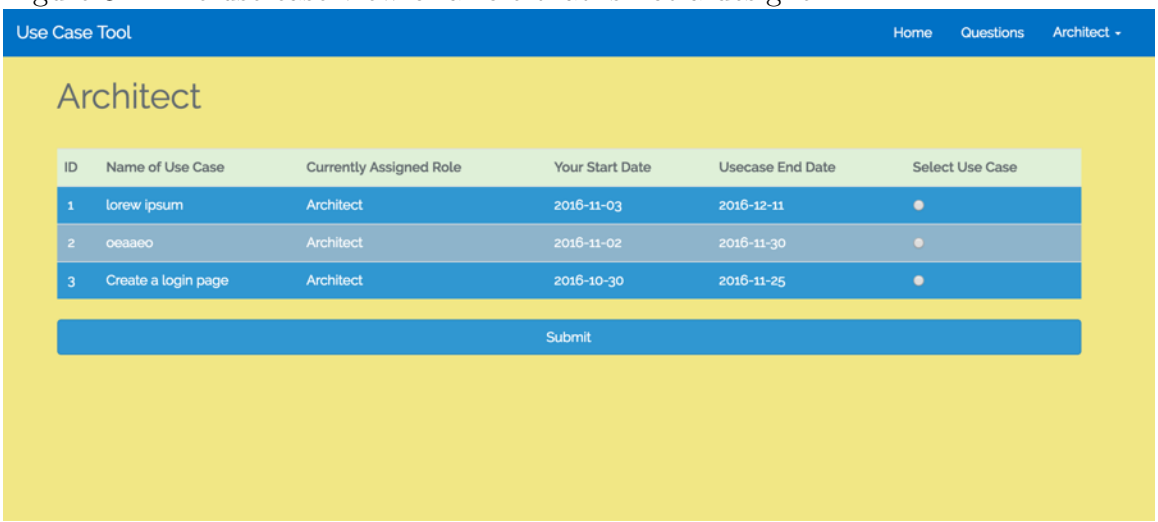


Figure 32. The use case view of a role that is not a designer

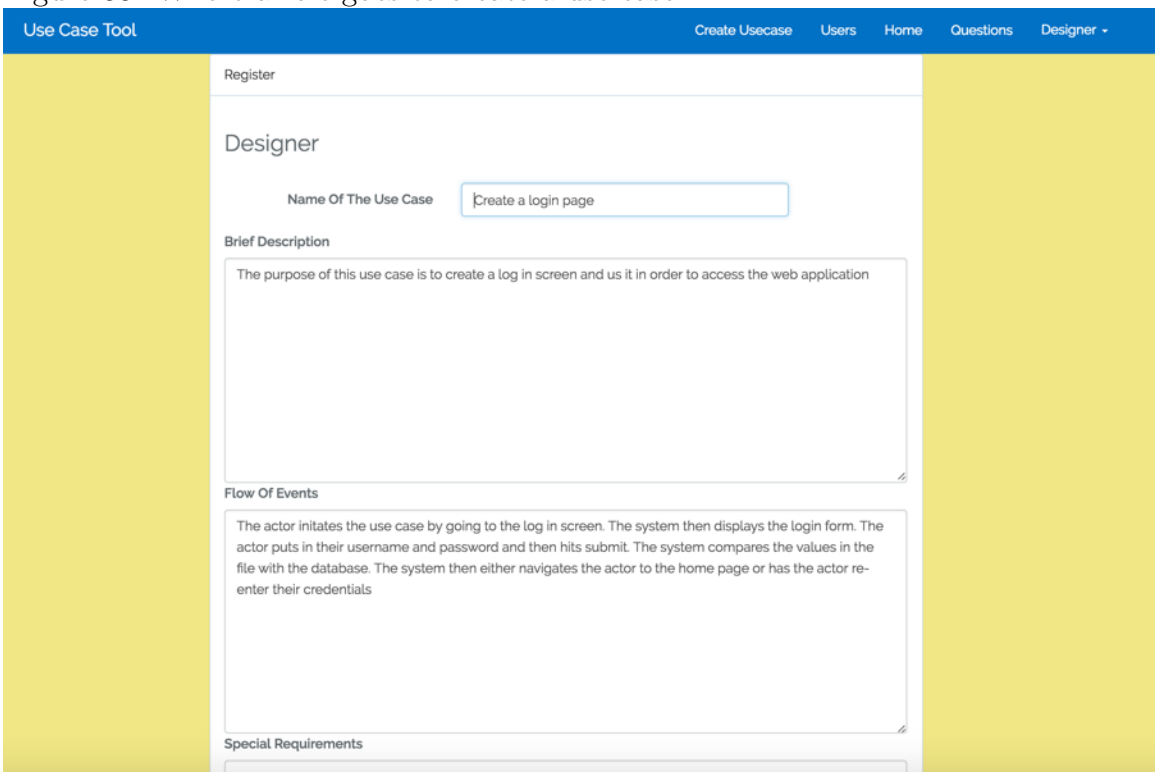


power to decide what goes into the master branch. Before we discuss this in more detail, let's look at how a designer adds a new use case.

Once the designer clicks on create a use case, the designer will be sent to the createUseCase page.

This page has text boxes for the eight key aspects of creating a use case. These are Brief Description, Flow of Events, Special Requirements, Preconditions, Post Conditions, Extension Points, Relationships, and Diagrams. Each of these aspects

Figure 33. Where a role goes to create a use case

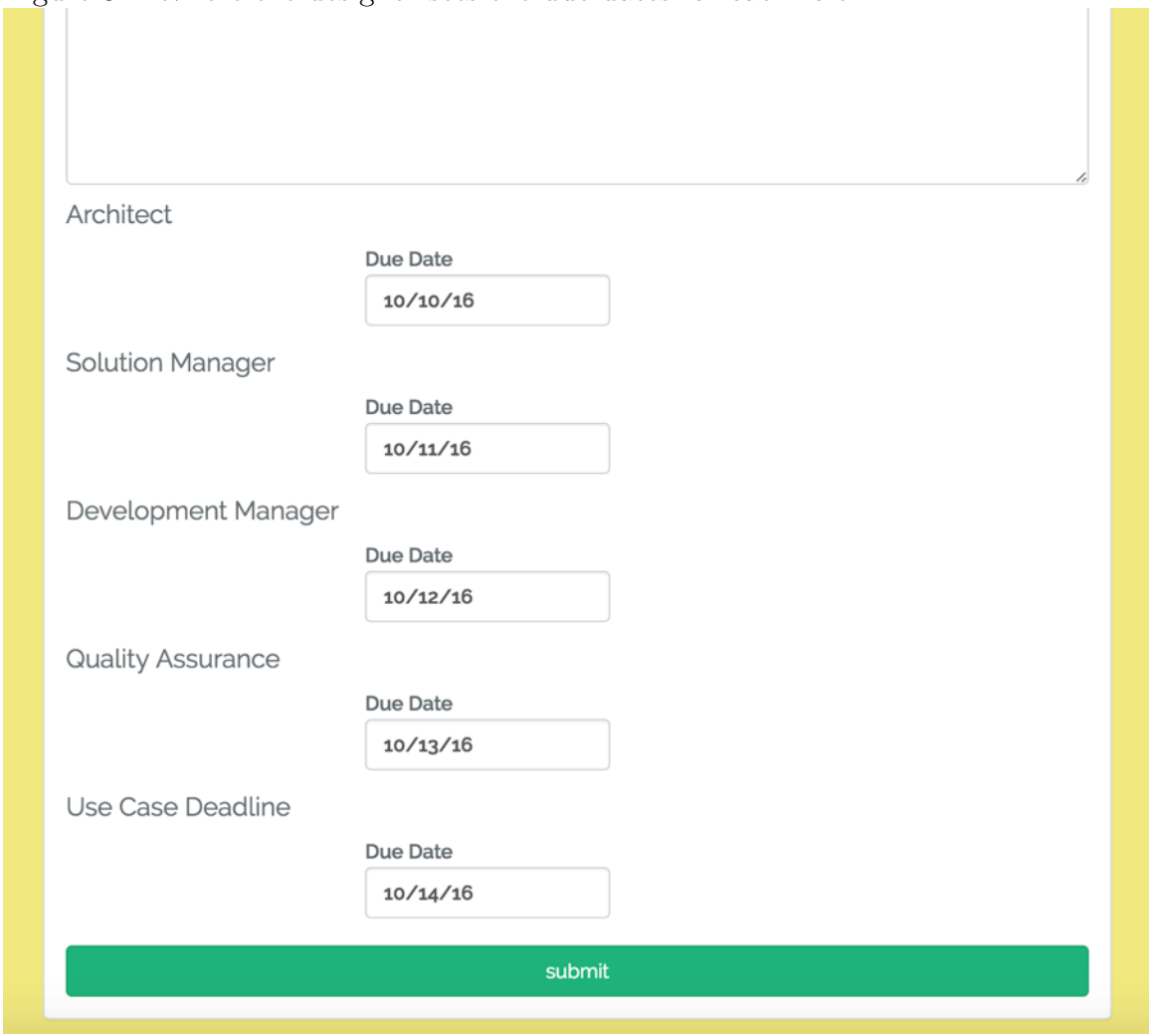


makes up the main flow of a use case. A use case also has alternative flows and non-functional requirements. When a use case is created, only the main flow is created because alternative flows and nonfunctional requirements can be added later.

At the very bottom of this page are the due dates for each of the other roles. These dates are set by the designer.

The dates refer to the last day that a given role has to add their comments to the master branch. A given role's start day is one day after the due date of the prior role. The roles are in order according to who gets the use case first. Once the use case has been created, the designer will be sent back to the homepage, which will display all use cases, including the most current one.

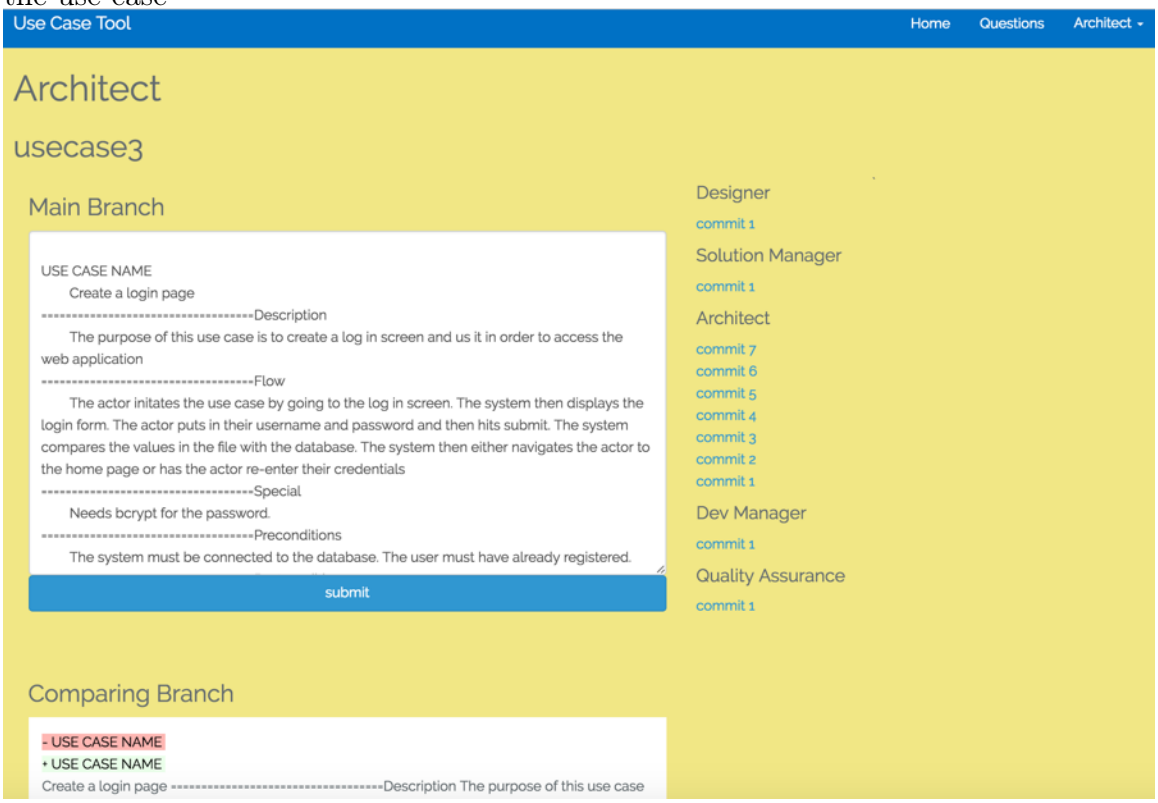
Figure 34. Where the designer sets the due dates for each role



Once the designer or any of the other stakeholders clicks on one of the use cases on the homepage, they will be sent to a use case editing page.

There are three main sections to this page. The first section is the top text box. This text box will hold either the master branch or the current user's branch. If the user us within their allotted time to edit the use case, the text box will hold the master branch. However, if they go to this page outside their allotted time, they will only see their branch. However, the user will not know whether they are

Figure 35. What a role sees such as an architect when they want to manipulate the use case

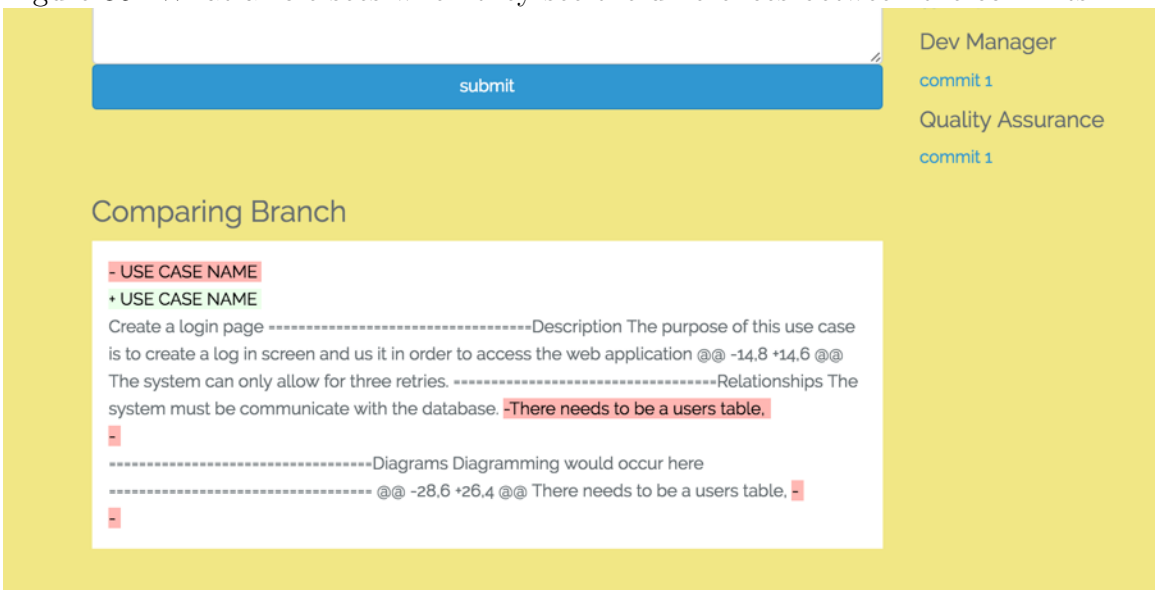


on the master branch or their own branch until the changes they submit during their allotted time can be seen by the designer. Whether on the master branch or their own branch is immaterial to the user. This application should abstract that technicality from them.

The next box is the bottom text box, which will only contain the branch the user chooses to select. It is in this textbox that the user can view anyone else's branch and all the changes that other users have contributed. Git commands are made on the backend, which switches branches and displays the information.

The comparing branch above shows lines highlighted in red and lines highlighted in green. The box is read-only and shows what has changed from the per-

Figure 36. What a role sees when they see the differences between the commits



spective of the clicked commit. In other words, those lines highlighted in red main branch changes and those things in green are what is different from the comparing branch.

Finally, let's look at the column on the right. This column will be where a user can look at all of the contributions of each other user. There is a three-tiered hierarchy in this sidebar. The top tier is the name of a user. Once one clicks on a user, one will see all of the commits made by that user. Each of these commits makes up the second section of the hierarchy.

9.2.6 Merges

Merging in the design collaboration tool is handled manually, primarily because each user determines what they want in their branches. While their main branch is open, the user clicks on another branch that interests them, then adds the changes

they see in the comparing branch to their main branch. They will then push submit which will allow them to commit the newest changes.

Chapter 10

TESTING

Now that we understand the tool, it is now time to validate it.

10.1 Methodology

This app is validated by presenting this application to a design class by Dr. Gaffar. Before this class, I created 15 separate use cases. I was the designer initially. I then handed out 11 sheets of paper where there were four slots and one role was assigned to each slot. Each of the students would put in their student IDs and names. They were then instructed to go to the website and register as one of the roles. The idea was to have them each work on a specific use case and get a feel for the collaboration. When each role made changes to a use case, the other roles would be able to see those changes reflected in the application. Once this was complete, they would go and answer a survey website.

1. How do you like this tool?
2. How easy do you find this tool to use?
3. Do you think this tool will be helpful for designing software projects?
4. What feature do you think would improve this tool the most?
5. Overall from 1 to 10 how would you rate this tool?
6. From 1 to 10 how do you rate this tools potential?

10.2 Results

The responses to this tool have been overwhelmingly positive.

How do you like this tool?

84% of people found it useful

How easy do you find this tool to use?

72% of people found it easy to use

Do you think this tool will be helpful in designing software projects?

76% of people said yes

Overall from 1 to 10 how would you rate this tool?

7.58 Average

From 1 to 10 how do you rate this tool's potential?

8.64 Average

What feature do you think would improve this tool the most?

Received a large set of answers

Chapter 11

FUTURE WORK

This tool is meant to be a proof of concept tool. There are many features that can be implemented to improve the tool so that the outlined methodology can be realized. Below is a list of features.

1. Role Hierarchies

   There are times in which teams work together under a given role. If a role such as the development manager decides to have developers contribute to the manager first and then the manager decides to contribute to the master branch, then this role hierarchy must be considered.

2. Advanced policy management

   Advanced policy management accounts for flexibility in how the design product will be implemented. Each design building block should have the flexibility of having its policy associated with it. If one design building block wishes to be FF and SS and another wishes to be SF or FS, then this should be allowed.

3. Preferred notification methods

   Each user that signs up for this site should be able to have the choice of how to be notified when a change has been made such as a new change in the use case. The user should be able to choose from email, SMS text, or other push notifications.

4. Incorporate multiple collaborative design artifacts

   This tool implements only one design building block, use cases. However,

there are many other design building blocks that designers use. Therefore this tool should offer those design building blocks as well.

5. Multiple use case structures

   There should also have different versions of the design building blocks. This tool only uses the structure that was outlined by Bittner. There are other use case structures the designers should be able to choose from. This tool should offer all of the different varieties.

6. Custom uses case structure

   A designer should be able to create their use case structure. This allows the designer the flexibility to create a use case structure that is specific to a goal of a given use case.

7. Editable collaborative graphs

   The editable collaborative graphs tool should also allow for graphical manipulation. Many of the designs that designers create are graphical. Therefore these graphical designs should be editable by all of the different roles assigned to work on the use case.

8. Manage all in one framework

   There are many different tools that are tailored to each design. This tool would have a lot of power by being a one-stop shop for all tools to support a given project.

Chapter 12

CONCLUSION

Software design is a collaborative effort. This part of the software lifecycle requires contributions from many different teams. It requires knowledge about all the other aspects of software engineering. The reason for this is the design must reflect all that is necessary for a piece of software to reflect the software requirements with high quality. Each designer does not have the time or the expertise to do this all on their own. So, it is important to get as many people to contribute as possible. For this to happen, this needs to be done in a way that is secure, flexible, and scalable. Our work has shown that traditional ways of comparing textual contributions are not adequate. My work has shown a methodology that will replace post it notes, Word, Excel, and other general tools that are currently used for design. This document has tested the proof of concept tool that has been overwhelmingly positive. This thesis has shown and proven that there is a better way.

REFERENCES

. n.d. https://github.com/about.

. n.d. https://github.com/laravel/laravel.

. n.d. https://twitter.com/madebyloren.

Allen, F. E. 1981. "The History of Language Processor Technology in IBM." *IBM Journal of Research and Development* 25, no. 5 (September): 535–548.

Bittner, K., and I. Spence. 2008. *Use case modeling.* Boston MA: Pearson Education, Inc.

Bourque, P., R. E. Fairley, and eds. 2014. *Guide to the Software Engineering Body of Knowledge, Version 3.0.* IEEE Computer Society.

Business Process Management (Hrsg.), European Association of. 2011. *Business Process Management Common Body of Knowledge - BPM CBOK: Leitfaden für das Prozessmanagement herausgegeben von der EABPM (European Association of Business Process Management).* European Association of Business Process Management, April.

Diaper, D., and N. A. Stanton. 2004. *The Handbook of Task Analysis for Human-Computer Interaction.* Mahwah, NJ: Lawrence Erlbaum Associates, Inc.

Dijkstra, E. W. 2010. "On the foolishness of "natural language programming"." https://www.cs.utexas.edu/users/EWD/transcriptions/EWD06xx/EWD667.

Griswold, W. G., and K. Shonle M. Sullivan. 2006. "Modular software design with crosscutting interfaces in IEEE Software." *IEEE Software* 23, no. 1 (January): 51–60.

Hackos, J.T., and J. C. Redish. 1998. *User and task analysis for interface design.* John Wiley & Sons, Inc.

Herrera, E. 2015. *The BPMN Graphic Handbook.* Kindle Edition.

Humphrey, W. S. 2000. *The Personal Software Process (PSP).* Technical report. 5 Eglin Street, Hanscom AFB, MA 01731-2116: SEI Joint Program Office, November.

Johnson, P. M., H Kou, J. Augustin, C. Chan, C. Moore, J. Miglani, S. Zhen, and W. E. J. Doane. 2003. "Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined." In *ICSE International Conference on Software Engineering.* Corvallis, Oregon, May.

Panagacos, T. 2012. *The Ultimate Guide to Business Process Management: Everything you need to know and how to apply it to your organization.* CreateSpace Independent Publishing Platform. Kindle Edition.

Pressman, R. 2005. *Software Engineering: A Practitioner's Approach 6th ed.* New York, NY: McGraw-Hill.

Program, Univalent Foundations. 2013. "Homotopy Type Theory: Univalent Foundations of Mathematics." https://homotopytypetheory.org/book.

Project Management Institute, Inc. 2000. *A guide to the project management body of knowledge (pmbok guide).* 5th ed. Newtown Square, PA: PMI.org.

Rafique, I., W. Jingnong, W. Yunhong, M. Q. Abbasi, Lew P., and X. Wang. 2012. "Evaluating Software Learnability: A Learnability Attributes Model." In *Systems and Informatics (ICSAI), 2012 International Conference on.* Beijing, China, June.

Rolland, C. A. 1998. "Comprehensive View of Process Engineering. International Conference on Advanced information Systems Engineering." In *Lecture Notes in Computer Science,* 1–24. Italy: Springer, 1413.

Ruckert, M. 2015. *The MMIX Supplement: supplement to The art of computer programming, volumes 1, 2, 3.* Upper Saddle River, NJ: Pearson Education, Inc.

Scacchi, W. 2001. "Process Models in Software Engineering." February. http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf.

Stanton, N., and M. Young. 1998. "Is utility in the eye of the beholder? A study of ergonomics methods. Applied Ergonomics." 29(1):41–54.

"W3-taskmodel." n.d. https://www.w3.org/TR/task-models/.