CSC440/540 Homework 3 (Design factors)

- Name: Mike Huber
- Deadline: October 4, 2017 (Wednesday)
- Submitted: October 4, 2017 (Wednesday)
- Time analysis

	Q1	Q2	Q3	Q4	Total
Estimation	01:00	02:55	00:45	03:00	08:00
Measurement	00:57	03:07	00:42	02:41	07:27

1.3.1 Revise and add writings in your "SE book"

[11:55 2017/09/26]

	Chapter 1	Chapter 2	Glossary	Total
Estimation	00:20	00:20	00:20	01:00
Measurement	00:31	00:16	00:10	00:57

Note: The time stamps have been removed from the chapters in this submission because they are confusing to look at along with the submission itself, since I bounce back and forth between doing someting and noting it in the submission. These time stamps are in the book on BitBucket.

Mike Huber - SE Book Chapter 1

This chapter is about the definition of Software Engineering. Since there is no single perfect definition that has been totally agreed upon yet, there are multiple versions. My version is the following.

My definition of SE

Software Engineering is the craft of creating software systems from the ground up. This starts with the idea, which leads to the design, which leads into the creation of the system. After this system is made, it is then finally modified and maintained to stay updated with the needs of the users.

Others' definitions of SE

1. "The systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software"
   The Bureau of Labor Statistics - IEEE
2. "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software"
   IEEE - Standard Glossary of Software Engineering Terminology
3. "An engineering discipline that is concerned with all aspects of software production"
   Ian Sommerville - Software Engineering 8th Edition
4. "The establishment and use of sound engineering principles in order to economically obtain software that is reliable and works efficiently on real machines"
   Fritz Bauer - Software Engineering, Information Processing
5. "A branch of computer science that deals with the design, implementation, and maintenance of complex computer programs"
   https://www.merriam-webster.com/dictionary/software%20engineering

Comments on Definitions

1. I agree with most of this definition and especially like the tie-in of both scientific and technological knowledge. The part I would maybe modify is the end. I would include the idea of upkeep for the software as well. Maintenance is just as important as implementation in my opinion.
2. Again, the end of this definition seems to be missing a part. In this case, I would argue that they are missing conceptualization and design of the software.
3. This one is interesting and a bit different and more vague than the previous two. While I do enjoy the ability to conceive whatever I like about the meaning of "software production", this is not a very effective definition because of the lack of clarity.
4. My comment on this definition is that it is missing a lot of the main parts on Software Engineering, at least the way that I see it. Sure, we want our software to be reliable and efficient, but there is no mention of design, or the process of implementing or maintaining at all. I don't like this definition much at all.
5. On the opposite side of the spectrum from the previous definition, this one is my favorite so far out of all I've seen. It describes the design process which others tended to miss, and includes the maintenance work

of the system. It also includes a very key word, which is *complex* computer programs. The only part of this one that is a little arguable is that Software Engineering doesn't necessarily have to branch from Computer Science, but it is accurate to say that the meat of Software Engineering does come from Computer Science.

Mike Huber - SE Book Chapter 2

Principles are peoples' personal goals and ways to keep themselves in line. Rules are enforced by a small group and can be argued against and possibly modified. Laws are absolute, no matter if you agree with them or not and are not modified. In this chapter, we discuss a few rules relating to Sofware Engineering.

Rules

- KISS is a rule that was discussed in class. It stands for "keep it simple, stupid" and actually originated in the U.S. Navy. This rule plays well into the challenge that software engineers face, which is complexity. Simple, being the opposite of complex, is what we strive for in our software. Further research from documents and videos on the internet suggests that this rule was also loosely based around Leonardo Da Vinci's concept of "Simplicity as [is] the ultimate sophistication".

Mike Huber - SE Book Glossary

Glossary of all definitions gathered throughout this book and the course of CSC440. First definitions in bold on each are my personal definitions.

Software Engineering

Software Engineering is the craft of creating software systems from the ground up. This starts with the idea, which leads to the design, which leads into the creation of the system. After this system is made, it is then finally modified and maintained to stay updated with the needs of the users.

- The systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software
  The Bureau of Labor Statistics - IEEE

- The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software
  IEEE - Standard Glossary of Software Engineering Terminology
- An engineering discipline that is concerned with all aspects of software production
  Ian Sommerville - Software Engineering 8th Edition
- The establishment and use of sound engineering principles in order to economically obtain software that is reliable and works efficiently on real machines
  Fritz Bauer - Software Engineering, Information Processing
- A branch of computer science that deals with the design, implementation, and maintenance of complex computer programs
  https://www.merriam-webster.com/dictionary/software%20engineering

Design

Design does not have a personal definition yet.

1.3.2 Watch video and make summaries

	Simple Made Easy	Making Architecture Matter	Agile Architecture	Total
Estimation	01:30	00:25	01:00	02:55
Measurement	01:32	00:28	01:07	03:07

1. Simple Made Easy

	Watch	Summary	Total
Estimation	01:00	00:30	01:30
Measurement	01:01	00:31	01:32

Summary

One of the quotes this speaker, Rich Hickey, starts off with is from Edsger W. Dijkstra, “We should aim for simplicity because simplicity is a prerequisite for reliability”. This segways into his point that the definition of simple is commonly mixed up with the definition of easy. The talk goes into detail about this as it is important to the rest of the talk. Rich explains that the origin of the word easy comes from the idea of being physically close, which makes sense because if something is close to you it is *easy to obtain. On the other hand, the origin of the word simple means intertwined. One other word to go along with these is complect. Rich defines this as the act of intertwining, so in a modern case, complecting means to take something simple and make it complex.

While complex software code can possibly make our experience writing that code terrible, it isn't the main answer Rich gives us to questions about the quality or reliability of our software. The real idea behind these questions is whether or not someone else down the road can understand and change your existing software. This includes being able to effectively reason about the program and know what will be affected when a change is made. A funny, but very true, statement Rich makes is "[every bug] ... passed the type checker, and it passed all tests" meaning without our knowledge of the program, there are bugs that can slip past our unit tests.

To point out some directly complex versus simple constructs, our speaker gives a good list of each which I will include some of here:

Complex

1. State
2. Object
3. Methods
4. Syntax
5. Inheritance
6. Vars
7. Loops
8. Conditionals

Simple

1. Values
2. Functions
3. Namespaces
4. Data
5. Polymorphism
6. Queues
7. Rules
8. Consistency

In his conclusion, Rich talks specifically about what can be done to build simple systems. A reason why I enjoyed this talk so much, because he offers ideas and solutions instead of just complaining about what is wrong with the way systems are currently built. One part of building this simple system is abstracting by asking questions like who, what, when, where, why, and how. The key concept here though is to not mix any of the two, or at least as few as possible. An example that was given is if you figure out clearly the answer to what, you can leave the how to someone else and solve later. Another step toward building a simple system is by choosing constructs that generate simple artifacts. The list of what simple constructs are good to use was given earlier and is the list given above. The last part of building a simple system Rich gives us is to simplify by encapsulation, which as he puts it is, “I don’t know, I don’t want to know”.

Comment: Seeing the list of complex constructs that Rich came up with, made me realize that I use a lot of those. I think I can keep this in mind as I progress as a software developer/engineer and try to use more simple constructs in the future.

[13:10 2017/09/26][14:03 2017/09/28]

2.2 Making Architecture Matter

	Watch	Summary	Total
Estimation	00:15	00:10	00:25
Measurement	00:14	00:14	00:28

Summary

This video was short and sweet, even within fifteen minutes, Martin Fowler was able to give some really good insight into software architecture. The two main parts of this talk were about what software architecture is and then why it is important. To start, Martin explains that he went back and forth with Ralph Johnson on how to define software architecture. What they came up with was that things that are hard to change and the overall understanding of the group were the important things. From there, Martin mentions that software architecture is just that, the important stuff.

Now the question stands, why is the important stuff important? Martin answers this beautifully with a small graph that he drew out on a slide in his talk. With internal design comes modularity and the ability to add more functionality later on in the project easily. His graph shows a curve that evens out quickly when there is no design, meaning adding functionality becomes harder and harder as time goes on with the poor structure of the code. On the other hand, the curve quickly goes up when there is effort on design. This means that it will actually become easier to add new functionality because the groundwork for the code is already there and is solid.

Again, I really enjoyed this talk because Martin was able to very effectively give me an idea of software architecture within a short amount of time. Explaining to an executive the importance of good internal design will be much easier now with a better grasp on the concept.

[14:17 2017/09/28][14:22 2017/09/28]

2.3 Agile Architecture

	Watch	Summary	Total
Estimation	00:40	00:20	01:00
Measurement	00:38	00:29	01:07

Summary

Honestly, I did not like this talk very much at all. Switching back and forth between talkers was confusing to say the least and made it almost seem like they were disagreeing on points that they were clearly agreeing on. Besides that, the start of the talk included Agile programming but it seemed to only be briefly mentioned, and then the discussion turned into either being about software architecture or how the company the speakers work for do something a certain way. This may have been the point since the talk was about both Agile and architecture, but early on, the talk stopped being about Agile entirely and wasn't mentioned at all later on other than in passing. That being said, I will summarize the bits of the talk that were understandable or coherent to me.

Starting off, Molly Dishman and Martin Fowler talk about Agile programming. From what I gathered, the waterfall process of one thing leading into another is combined into one from the documentation point of view. Molly mentioned that most see it as "working code with no documentation" but it was hard to follow if they really were including no documentation or if it was all together as the slide suggested. If they aren't using documentation, I fail to see how communication and understanding can still work in a software project. Another topic similar to this with Agile was the concept of responding to change over having a plan. Again, Molly states that most interpret this as "responding to change and not having a plan". This sounds as if the code is being changed dynamically in the effort to get the project in the right direction, but again I fail to see how this benefits the project. I may not have the right understanding of Agile programming from this talk, but these were the main points I took away from it.

The main points I received from other parts of the talk were collaborating to make sure everyone is in agreeance on what the hard things to change are, how the architects should look into what is actually being done in the code base, and using tools to help discern whether or not the project is going in the right direction. A part of the collaboration that Martin mentioned is that it doesn't always have to be a face to face group meeting. This can be the technical lead looking into the commits on the repository and making sure everyone is agreeing on what direction the project is heading. Another example that was given is that a lead would walk around and ask developers to describe the program with only four objects. If everyone is giving the same or similar objects, everyone is on the same page. If not, that lead would maybe consider having a meeting to go over exactly how the project should look so that everyone can get on the same page. Part of the team lead looking into the commits, as mentioned earlier, also helps them learn what works and what doesn't, as well as allow them to see what is being done by the developers. Martin talks about how he brought this idea to his wife who is a structural engineer and she said that all engineers should verify that the builders are doing everything the right way, because half the time, they won't. Lastly, another way to check into the direction of the project is with tools that can be set to look for certain parts that currently exist in the code base that shouldn't, based on where the team lead is invisioning the project going.

A lot of this video seemed like buzz words and information about things useless to both architecture and Agile programming. It was difficult to extract the high points in this discussion, more research and understanding will need to come in order to really get the concept of Agile and the previous short video about Martin's idea of software architecture was way more helpful than this one.

[14:51 2017/09/28][13:30 2017/10/03]

1.3.3 Definitions

	Design	Software Design	Software Architecture	Total
Estimation	00:15	00:15	00:15	00:45
Measurement	00:21	00:12	00:09	00:42

1. Design

Instructor Definition: Design is an order that demands an intended actions.

- To create, fashion, execute, or construct according to plan.
  https://www.merriam-webster.com/dictionary/design
  -> Comparison: The closest part of this definition to the instructor's is that it is based around a plan - in the instructor's definition, the intended action.

- Realization of a concept or idea into a configuration, drawing, model, mould, pattern, plan or specification (on which the actual or commercial production of an item is based) and which helps achieve the item's designated objective(s).
  http://www.businessdictionary.com/definition/design.html
  -> Comparison: This definition is similar to the previous in that everything points to the designated objective. So far, each definition leads to something specifically defined as a plan or objective - a goal really.

- To make or draw plans for something.
  http://dictionary.cambridge.org/us/dictionary/english/design

-> Comparison: Out of the others found, this definition is very vague which is what I normally don't like about definitions, because they don't seem clear. In this case the definition is not similar to the instructor's because it makes no mention of a direct objective.

[13:51 2017/10/03][13:52 2017/10/03]

2. Software Design

Instructor Definition: Identifies and defines "modules"(units) of the software implementation in enough detail that they can be coded and they can be tested individually.

- Software design is the process by which an agent creates a specification of a software artifact, intended to accomplish goals, using a set of primitive components and subject to constraints.
  https://en.wikipedia.org/wiki/Software_design
  -> Comparison: These two definitions are similar in that they describe the specifics of the software and that it is created with a goal in mind.
- The process of defining software methods, functions, objects, and the overall structure and interaction of your code so that the resulting functionality will satisfy your users requirements.
  https://sea.ucar.edu/best-practices/design
  -> Comparison: This definition looks at the overall structure while the instructor's mentions the idea should be in detail.
- Software design is the process by which an agent creates a specification of a software artifact, intended to accomplish goals, using a set of primitive components and subject to constraints.
  http://www.definitions.net/definition/software%20design
  -> Comparison: Our last definition is identical to the first, I included it because I think it's interesting that I found two sources that agree with one another. This one compares to the instructor's definition in the same way, that they both look towards a goal of some sort for the software.

[14:04 2017/10/03][14:05 2017/10/03]

3. Software Architecture

Instructor Definition: The goal of software architecture is to minimize the human resources required to build and maintain the required system.

- Software architecture refers to the high level structures of a software system, the discipline of creating such structures, and the documentation of these structures.
  https://en.wikipedia.org/wiki/Software_architecture

-> Comparison: The high level concept from this definition is agreed upon with the instructor. It is not mentioned in the listed definition, but from the class discussion, software design is seen as low level and software architecture is seen as high level.

- The set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both.
  Documenting Software Architectures: Views and Beyond (2nd Edition), Clements et al, Addison-Wesley, 2010
  -> Comparison: I like the part of this definition about the ability to reason about the system from the architecture. This fits nicely with the instructor's definition as making the architecture in a way that makes it easy to reason about the system, minimizes the human resources needed for building and maintaining that system.

- Software architecture is the defining and structuring of a solution that meets technical and operational requirements.
  https://www.techopedia.com/definition/24596/software-architecture
  -> Comparison: This definition and the instructor's are similar in that they both are focused on the requirements for the software system when talking about the software architecture for that system.

[14:14 2017/10/03][16:02 2017/10/03]

1.3.4 Read papers

	Foundations for the Study of SA	The World and the Machine	Total
Estimation	01:30	01:30	03:00
Measurement	01:11	01:30	02:41

1. Foundations for the Study of Software Architecture

	Read	Summary	Total
Estimation	01:00	00:30	01:30
Measurement	00:42	00:29	01:11

[16:44 2017/10/03][16:45 2017/10/03]

Summary

As a high level model of software architecture, the abstract of this paper describes three components that make up software architecture as they define it. These three components are elements, form, and rationale. The elements are the smallest pieces being the actual processing and data. Next, form is the connection between all of our elements. Lastly, the rationale is the understanding of the architecture with constraints in mind that usually come from the requirements provided.

In the introduction, this paper talks about the time table of software engineering. One big transition being in the 1980s, when software engineering moved towards the design and design process. The reason this is big is because it was the start of the mix between design and implementation. Setting up the future for software architecture is this paper's main goal, and is accomplished by starting with the ground work of the basic idea behind software architecture, then providing models and examples for how to implement this idea, and elaborating on a few of the benefits to software architecture.

The section that I took interest in was about the examples and notation for software architecture. In this section, two types of compilers are examined: a sequential one and one implemented by parallel processing and a shared internal system. Each part of the compiler is broken down into different parts. The lexer, which takes characters from the code as input, and outputs tokens (chunks of those characters to represent the code). The Parser, which takes those tokens as input and makes phrases out of them to output. The semantor, which pulls in the phrases from the parser and makes correlated phrases in order to keep the ordering of the code. An optimizer, which is optional but annotates the correlated phrases that come from the semantor. And the code generator which takes the correlated code with or without notation, and outputs the object code. The way the sequential compiler works is by checking to make sure that each of these parts are completely finished with their task before moving on the next part. This is what the paper calls a "classical" compiler architecture.

Now, the parallel processing compiler is similar to the sequential one but with an added feature. The important parts of this compiler are just the lexer, the parser, and the semantor. Characters are still passed to the lexer, which outputs tokens, and the parser still deals with tokens and phrases, as well as the semantor handling phrases and correlated phrases. However, the difference is that the output of these three parts of the compiler all goes into an internal system, which will be able to do all of the procedures needed to complete the tasks of these tokens, phrases, and correlated phrases concurrently. Therefore, each element must be synchronized as to not run into any race conditions. This is done by looking at whether or not there *will* be tokens, phrases, or correlated phrases left instead of looking at whether or not there currently are none in the element currently being processed. Downsides do arise with this type of compiler though. A major one being the added complexity, but the question is if it is worth the tradeoff.

[17:14 2017/10/03][18:53 2017/10/03]

2. The World and the Machine

	Read	Summary	Total
Estimation	01:00	00:30	01:30
Measurement	00:45	00:45	01:30

[19:38 2017/10/03][19:39 2017/10/03]

Summary

I want to start by mentioning that I really enjoyed reading this paper and I'm very glad that I did read it. To start, Michael A. Jackson declares that we need to keep a healthy balance between the world and the machine when developing software. He then leads into describing exactly why we can engineer software, when we cannot do the same with things like poems and mathematical theoruems, which software is often compared with. The reason is because we are describing and building a physical object to interact with the world and solve problems. Jackson refers to the relationship between the world and machine as four specific facets. The modelling facet; the interface facet; the engineering facet; and the problem facet.

When we make a machine to simulate the world, like data models and object models, we are of course in the modelling facet. Issues can arise from this. A big issue is that we cannot describe the machine and the world in the exact same ways. The mapping of object x to object y in the machine certainly isn't the exact same as how those two objects relate in the world.

Another point that Jackson makes is that the world is the problem and the machine we build is the solution to that problem. The interface fact is there to provide the interaction needed between the world and the machine. A great example is given in the paper about an elevator. When the elevator is on the third floor, there is a bit in the machine that is set to 1 for an element in the array *floor_sensors[3]*. If the button to go up is pressed, that is an event of the world. It is also an event of the machine since there will be some event signal sent to change the direction of the elevator to up and then make the elevator move.

The engineering facet is what translates the requirement to a specification. For the previous example, this would be translating the button press from the user in the world to make the elevator go up and the bit in the machine to change to a 1 in the correct array element. The example given in the paper is the realization that an airplane must limit the ability to provide reverse thrust. In the world, we know that we only want to allow the airplane to provide reverse thrust when the wheels are on the ground. Sensors are placed on the wheels to pulse when the wheels are rotating. We then translate the state of pulsing from the sensor to mean that the wheels are on the ground. This is because the wheels are rotation if and only if they are on the ground. The sensor is pulsing if and only if the wheels are rotating. We can conclude that the sensor is pulsing if and only if the wheels are on the ground from the transitive property.

Differentiating between what and how is the main idea of the last facet. The problem facet. Problems are varied and cannot be approached by using a hierarchical structure. Jackson asserts that many problems in the world act more like a parallel structure than hierarchical. A good suggestion for tackling these types of problems is to decompose them into smaller problems. The example given is with a CASE tool. Smaller problems of creating a CASE tool are things like needing the ability to edit text, having a GUI to assist users, or having an information system that shows work progress.

As to keep this summary fairly brief (there is a lot of valuable content to talk about from this paper) I will touch on the basics of the four denials that Jackson mentions. The first is denial from prior knowledge. This is essentially knowing what problems the world will provide already so as to not reinvent the wheel. Jackson calls the second denial by hacking. He means the obsession with using a computer, rather than breaking into them. This is the idea that engineers like seeing their product come to life similar to how a bridge maker enjoys seeing the bridge that was created. Looking into the details or the stress of the design process and things of that nature is not how most developers spend their time. The third is denial by abstraction. We forget what the symbols we use actually stand for. Lastly, we have denial by vagueness. When we mix up the description of the world with the description of the machine - defined earlier as an issue developers run into - we get vagueness that can confuse others.

To end this summary, I want to make a connection that was made in the paper. In the midst of the explaination of description, Jackson quotes von Neumann and then simplifies the quote by saying “There is no sense in being precise when you don’t know what you are talking about”. And to end the paper, Jackson proclaims his respect for mathematics and says “It [mathematics] has served physics and eningeering well. It can serve software development well, too, if we make sure that we know what we are talking about”.

[20:24 2017/10/03]

Executive Summary

Time Tables

The main one is the first from the top of the submission. Other, more specific ones are in the area of the section that was timed.

List of What Was Learned

- Definitions of software engineering, design, software design, and software architecture
- The difference between simple and easy, as well as complicated and complex
- Why software architecture is important and what others define it as
- Different models for the compiler architecture
- How to balance the abstract between world and machine

Feedback

This final version of the homework made it less stressful and less time consuming to finish, which helped me learn more from it and gave me a chance to really dig into the videos and papers to get the full content from them. Along those lines though, I do want to mention that I feel like there have been too many changes. It's understandable that things need to be modified to either fix mistakes or make deadlines able to be met, but too many changes and edits make things confusing. Another point beside the confusion of too many edits is the timing on the edits. Some changes have been made close to the due date of an assignment that make it a shorter assignment or extend the deadline. This may be unfortunately rewarding a procrastination mentality that is unhealthy for learning the content. For example, since HW3 was slimmed down near it's deadline, those who hadn't started on it yet now plan to have to do less, while those who have already started are now upset that they started early and now have done more than the others. Another example to note is that the naming format to our homework has changed as well, since there is a lot of stress put on the fact that it needs to be perfectly aligned with the specification of how it should be named. It started as CSC440-last-first-HW# and is now just last-first-HW#.

Other than the changes, I can't think of anything else in terms of what I don't like. I have been enjoying the class a lot so far and have learned a lot of information that seems useful in the real world versus a class that just makes everyone code up a simple problem in a program and then tell us that that's all there is to software developing and engineering.

Lastly to just say what's on my mind, I will admit that I'm a little worried about the midterm exam because I feel as though I'm not sure what to study in order to do well on it. That mostly comes with the territory of learning about the idea of software engineering and the fact that it's still sort of a debate even to define it. Regardless, I feel like I'm starting to understand the correct concepts and ideas so as to at least show what I know on the exam and hopefully that is what you're looking for.