|  |  |
| --- | --- |
|  | Assignment 2: Code Complete Review |
|  |  |
|  | Syed Muhammad Dawoud Sheraz Ali-111417-BESE-5B  Software Construction – Sir Fahad Ahmed Satti  4/23/17 |

***Originality Certificate***

# **(This review will cover chapter 2-34, with exception of Chapter 30 and Chapter 33)**

# **SOFTWARE CONSTRUCTION**

“Software Construction is the detailed creation of the working, meaningful software through the combination of coding, verification, unit testing, integration testing and debugging.” [1]

As the name implies, Software Construction is the creation of the software. But the term holds a lot more than merely construction. Software Construction is a late stage of Software Development Life Cycle (SDLC). Reason being it requires a lot of things before the actual construction can begin. Such things include problem definition and categorization, requirement gathering, requirement verification, planning, designing and design verification. These are some of the things that should be taken care of before starting the actual construction process.

There are many ways by which the process of Software Construction has been defined. But the one best defined is in the **IEEE SWEBOK (**Software Engineering Body of Knowledge**) 2004 version (Section 4: Software Construction) [1].** The whole process of Software Construction has been broken down into small parts, with each part contributing a great deal of meaning into the overall process. For the review of **Code Complete 2nd Edition (provided on LMS NUST),** the explanation will be relating the chapters to the overall software construction process, as given in the guide. The overview of the process is:

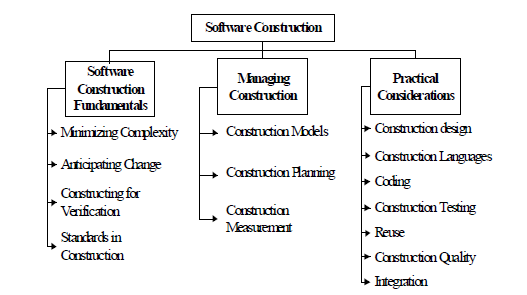


Figure 1: Breakdown of Software Construction Process [1]

# **MINIMIZING COMPLEXITY (ref chapter 2)**

“Rome wasn’t built in a day”. This particular line says it all. Software aren’t built in just one day. There is a lot of pre-processing required to deal with the creation of software. It is just like constructing a building. For the construction, the basic skeleton of the building is made. Based on skeleton, the workload is divided into pieces. After all the careful planning, the strong buildings are made. Same is the case of the software construction. This section will deal with all the chapters, as mentioned in reference that explain how reducing the complexity is a part of software construction.

## ***Chapter 2:* Metaphors for *a Richer Understanding of Software Development***

This particular chapter is all about the usage of analogies. Often in software industry, new software emerge from the previous ones. The creation of such pieces involves the use of analogies and heuristics. By the analogy, one thinks about a system from the existing system. Based on analysis, one predicts that particular features are there in the old system. The new system will have similar features like those. This calls for an analogy. By analyzing the old system, features for the new system are developed.

Before using the analogy, there is great deal to understand the relationship between the two software. Make sure that they come under the same category. The complexity and time will be approximately same. Not only that, but the resources and management would be nearly same. E.g. building a Dog House is a lot difference that building an eight-story large building. They don’t have much in common, apart from fact that both are some sort of buildings. Same is the case for the Software Systems. There should the match between them.

Analogies are like the metaphor. Metaphors relate to original things, but they aren’t original. Such metaphors doesn’t tell the original answer. Rather they show how to approach the problem. This is the complexity reduction step. Instead of approaching the software as a giant workload, make models and metaphors. Analyze any existing familiarity. There is always a high probability that you would find something. Such findings will tell you what and how to do.

Analogies are not any algorithm. They are heuristics. They would never tell how to do. Instead, they act as guidelines to achieve the results. Software construction uses such heuristics and metaphors to get to the final result. Some examples include comparing software construction to Growing up crops on a farm. To grow crops, there is great deal of work to be done. Deciding what to grow is important. In Software term, it is the problem definition. Choosing the appropriate land and the conditions are next step. This means the requirements, restrictions and design constraints are involved here. Then, sowing the seed, watering them accordingly, removing the weeds and reaping the final product are involved. In software terms, the method comprises of coding, debugging, testing, integration and system check.

This is just an example of metaphors. There are others as well, but the idea of all is that they tell how to approach the software construction process. What matters is the choice of metaphors. Since these are heuristics, many can be combined into one piece. To use such metaphor, careful planning is required.

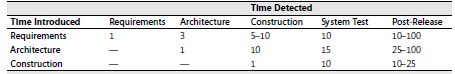
# **ANTICIPATING CHANGES (ref chapter 3)**

Software systems are meant to change. Customers change the requirements a lot. Handling the change request is one of the hard things in construction process. If the designed system doesn’t has capacity to accommodate changes, the system can face the rejection by users. McConnell discusses this issue in chapter 3.

# ***Chapter 3:* Measure Twice, Cut Once: Upstream Prerequisites**

As the name implies, measure twice before making the decision of cutting. Once cut, nothing can be done. That’s same thing for software. This not only applies for original product, but for the part where we introduce the changes. McConnell suggests that before starting the construction, one must know the system by heart. Each and every possible detail should be there. When someone knows the system that well, they can anticipate the change request and where it can come.

Besides the maintenance, the major portion is about the defects that are caused by not knowing the system. It is because if one is unclear about the system and still codes it through, one time will comes when the coding will not proceed and system will require the change. That change will cost a lot. The following figure shows the various stages where the change can be detected and the cost to fix it:



**Figure 2: Average cost of fixing a bug due to incorrect specifications**

As the figure depicts, the cost of fixing a bug, due to incorrect system specifications is a lot if discovered in later stages. That’s why it is necessary to understand the system before beginning the process. Not only the system, but the category under which that system comes. Some categories are more critical than others and require a lot of thinking before beginning the work such as safety critical systems or mission critical systems. These systems are more dangerous than any regular system in sense that changes aren’t easy to handle. Any bug detected can lead to a lot of losses.

Not only system goes through change due to incorrect specifications, but also due to customers’ change request. It is common saying that customers don’t really know what they want. At first, they would give one information and after a while, they might change the whole thing. This is one reason to understand the whole system properly as one wrong action can lead to many problems. McConnell tells some guidelines to deal with the changes asked by the customers as:

* Make a checklist of new requirements and integrate them after the initial system development
* Calculate the cost and risk of the new requirement. If there is high risk (regardless of cost), integrate it first. Otherwise, one can keep medium and low risk requirements for end. If there are multiple high risk requirements, assign priorities and act accordingly.
* Make Change-control teams that analyze the change requirement and determine its feasibility. That team is responsible for telling customer how much that change can cost.
* In the end, if your customer is coming up with new requirements every day, just dump the project. The customer who brings new requirements each day isn’t really sure about the system and there is great chance that he/she will be unhappy with final product. Better dump it in start rather than at end, where all your effort will go pointless.

Software are prone to changes. The best way is to make space to fit those changes.

# **CONSTRUCTING FOR VERIFICATION (ref Ch.: 8, 20, 21)**

Constructing for verification means that building the software in such a way that errors and faults can easily be detected and removed by the developer while coding. Not only that, but any other error should be easily detectable after integration. Anyone with technical information should be able to find out the error. Following are the views of McConnell in different chapters:

# ***Chapter 8:* Defensive Programming**

Defensive programming doesn’t mean that developer becomes defensive about the code and not accepting any argument against the code. Instead, it says that you don’t know what others can do with your system. Make your code to handle the exceptions. This means that don’t leave any opportunity for the error. If there are any errors, they should be easy to locate. McConnell describes some coding techniques that can be adopted to locate and fix the errors.

First thing is the handling of invalid inputs. Consider the users of your system as dumb beings. They will always poke around the system, providing invalid inputs. For that purpose, always do Fuzz Testing of the system. Fuzzing actually works on idea of providing the invalid input to the system.

Use assertions in your code. Assertions provide a way for system to check itself. If assertion is true, that means system is working fine. If assertion is wrong, there is some error in the system and system will stop at the point where assertion became wrong. Now-a-days, programming languages provide a built-in support for assertions via unit testing. Programmers can always detect the error easily using the assertions.

Handle the errors in the code. There are many ways an error can introduce into your system, hence, there are different ways to deal with it. If there is any error during a function call, return some default value (on which system can work). If there is no return value, stop the working. Always log the errors. Log the error, the time it occurred, the location, preconditions and post-conditions. Show any error message to the user. Stop the further execution if necessary. There are many ways to handle the error. It all depends on the sensitivity of the area where error arises.

Programmer can also provide with the error checking routine. The routine can check what type of error it is and try to fix it. If not, crash the program gracefully. The important thing is that if program crashes, make sure the current data is backed up.

# ***Chapter 20:* The Software-Quality Landscape**

In this chapter, McConnell focuses on how to measure the quality of a Software. For a software being developed, there are various quality attributes. These include correctness, reliability, usability, efficiency, accuracy, performance, adaptability and integrity. Users will search for these features in the software system. Most of these characteristics should be there in the final system.

To achieve them, define some quality standards. These standards can vary from organization to organization. These can include testing, inspection, formal reviews etc. The idea is to achieve most of the quality attributes. But the problem is one attribute can either encourage other attribute or discourage it. E.g. efficiency can cause bad effect on accuracy. When such situation comes across, maintain a balance between the attributes and see which ones are important than others.

As the main purpose of quality assurance includes the defect handling, there are many techniques for this. Some of these techniques include Informal design reviews, formal design reviews, unit testing, debugging, Prototyping, Integration Testing, Regression Testing, beta testing and modelling. Each technique has its own merits or demerits. It is better to combine more than one technique to achieve better results. By such combinations, the percentage of defects detected increases. After the bug detection, there is need to analyze the cost to fix that defect. It is advisable to reduce the bug rate by testing system with nearly all the defect detection techniques. This builds up the confidence in the system.

# ***Chapter 21:* Collaborative Construction**

This chapter deals with the process of detecting and removing bugs by the idea of collaborative construction. This process uses the idea of collaborating with others while programming. Study shows that simple collaborations results in better debugging. For example, Ahmed is having problem with the code about linked list. He has coded everything correctly. But there is always NullPointerException. He visits his co-worker Rana and tells him about the problem. He is explaining to Rana about code that I have coded this thing and that too. Oh wait, I forgot to initialize head node. Thanks for your help. Rana didn’t said a word. But Ahmed was able to find the problem. McConnell has discussed the technique of pair programming and how it helps for better quality code.

Pair programming involves two persons, with one coding and other watching the code and pointing out any problem. This technique requires a lot of practice. That’s why don’t make two programmers sit together who never had experience of pair programming. In the pair, at least one person should have knowledge of pair programming by having done it before. The person watching the screen shouldn’t just stare it. Instead, he/she should be active to anticipate what should be coded next after the current line.

Pair programming isn’t easy to handle. But it does comes with a lot of benefits like better quality code, earlier development, development stress reduction etc. But to achieve success, many things should be kept in mind. Define the standards for the pair programming. Switch the pair after a while. Don’t make team of people who don’t like each other. Assign one member as team leader who would guide the other through the process. Some things will be hard to get. But in the end, the result will be better product.

There are other collaborative construction techniques like Formal Inspections, Code Reading, and Walkthroughs etc. Each technique has its own charms. Formal Inspection comprises of team of different people who inspect the code of developer and find out any mistake. The goal is only to find error, not correct them. The developer doesn’t review his/her code. Instead, the developer only explains the problem and the way it was approached. Code reading is another technique. In this technique, two or three people read the code and if they find any error, they meet with developer and tell him/her about the error. They also tell a possible way to fix it as well.

Hence, there are different ways to handle the errors in the code. It all depends on what the organization chooses.

# **1.4. STANDARDS IN CONSTRUCTION (ref chapter 4)**

Everything follows some sort of standard. Without any standard, there is no way to check the integrity of the thing being made. In a section of chapter 4, McConnell discusses the importance of standards:

## ***Chapter 4:* Key Construction Decisions**

This chapter consists of discussion about programming languages and the standards in construction. The part of programming languages will be discussed later in section 3.2 on Construction languages. This particular part is about the standards of the software construction. There are no particular standards for the software construction. There are some generalized features, but it depends on the team working on the project. The team can have its own set of standards and they can work accordingly. McConnell discusses this point that even if they make the standards, the process of creating those standards should be done ahead of any construction. The team should follow those standards and can’t change them in middle of construction process. That’s why it is very important to choose the best standard that one can follow.

Besides that, there are some general standards such as:

* Standard coding conventions like meaningful variable names, comments etc.
* Quality Assurance by testing and debugging
* Code control among the team members
* Revision control tools

The idea of standards should be to deliver the best product.

# **3.2. CONSTRUCTION LANGUAGES (ref chapter 4)**

The choice of which programming language to use for the system construction is very important. Each language has its pros and cons. One must know the system very best to decide what programming language to code in. This is the most important thing. Wrong programming language can lead to frustration and problem during the development. In a part of chapter 4, McConnell discusses the choice of programming languages for the development.

## ***Chapter 4:* Key Construction Decisions**

In this chapter, McConnell has emphasized a lot on the choice of programming language. Deciding the language will lead to productivity and better product. If the language chosen isn’t in the best interest, the development will be very hard. There will be problems regarding the features that the language can offer. This is why it is important to choose the language that suites the development team. Study has shown that programmer working with the familiar language have 30-35% more chance of productivity. In today’s era, people are comfortable in programming with High-level languages like C++, Java etc. Not many people feel comfortable with Assembly Language or any low-level language. McConnell discusses some of the programming languages as:

* **Assembly Language** is the low level language. In this language, each statement corresponds to one process instruction. Due to this reason, it is very specific to the system architecture. Portability is an issue here
* **C** is the mid-level programming language. It does have high level language features like structures, machine independence and control flow.
* **C++** is the object oriented language. It is compatible with C language. Besides that, it offers a lot of OO features like inheritance, polymorphism etc.
* **Ada** is the high level language and it is used a lot for the embedded systems. It allows the features like abstraction and encapsulation. It is named Ada after the **Ada Lovelace,** the mathematician, who has been considered as world’s first programmer.

Besides these, there are other programming languages like Java, C#, FORTRAN, Python, COBOL, JavaScript, PHP, SQL, Perl and Visual Basic. There are different choices which depend on the system and the team making that system.

# **3.7. INTEGRATION (ref chapter 29)**

Integration is the activity in which different modules of a software are brought together to make a single system. This is the part where the final system development actually starts. The problems at this level are unknown. Separate modules can be tested to make sure they are working correct. But when they are combined, there is always a great chance of error. Such problems are usually termed as “Emergent Properties”. The view of Steve McConnell is given in the chapter 29.

## ***Chapter 29:* Integration**

According to McConnell, the process of integration depends a lot on the construction sequence. The way in which whole process of software construction is followed affects the final integration. We can only integrate what is available and fully tested. Integrating anything that has high rate of error will lead to inefficient behavior of the system. The final system can be stable. But if things aren’t built correctly, system will crash before even reaching into the final stage.

Integration is seen in many other fields. In building construction, the integration of small parts lead to a strong building. One could benefit a lot from the careful integration such as small error rate, less documentation, better code control and most important, the experience. But carrying out the integration is very difficult. Following are the two ways in which integration is carried out mostly:

* Phased Integration
* Incremental Integration

As per McConnell, the phased integration is just like regular integration. You create the modules separately, test them and remove any bugs. When all modules have been coded, integrate them all together. This is problematic as all those modules will experience the other modules for the first time. The cause of the problem would be hard to determine as each module has been tested individually.

On the other hand, Incremental Integration is very different. It says that rather than combining all the modules at same time, build one module and integrate it with the final system. Then build another module and integrate with existing module. If there are any errors, fix them before further integration. This process continues until the final system is established. McConnell gives an example of snow ball rolling from a high snowy mountain. As it rolls down, some ice comes along with it. When it reaches the bottom, it is one giant snow ball. The benefits of Incremental Integration include:

* Early Error Detection
* Early Success
* Careful Testing

There are different ways to carry out the Incremental Integration. These ways include the Top-Down Integration, Bottom-up Integration, Risk-Oriented Integration, T-shaped orientation and feature oriented integration. These are different patterns that depend on what type of system is being built. With the proper technique, builds are made every day to update the system. The process of continuous integration happens each day. This is the beauty of Incremental Integration. McConnell mentions that a lot that continuous integration improves the system quality, builds up confidence in the system and make the system fault-tolerant.

# ***References:***

1. ieeexplore.ieee.org/ielE/4425811/4425812/04425813.pdf (page 64)