Assignment 2: Code Complete Review

Syed Muhammad Dawoud Sheraz Ali-111417-BESE-5B
Software Construction – Sir Fahad Ahmed Satti
4/23/17

Originality Certificate

I, **Syed Muhammad Dawoud Sheraz Ali**, hereby declare that this report entitled "**Code Complete 2nd Ed. review**" submitted as an assignment for the course **SE312: Software Construction**, is a record of an original work done by me and that no part has been plagiarized without citations.

(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:

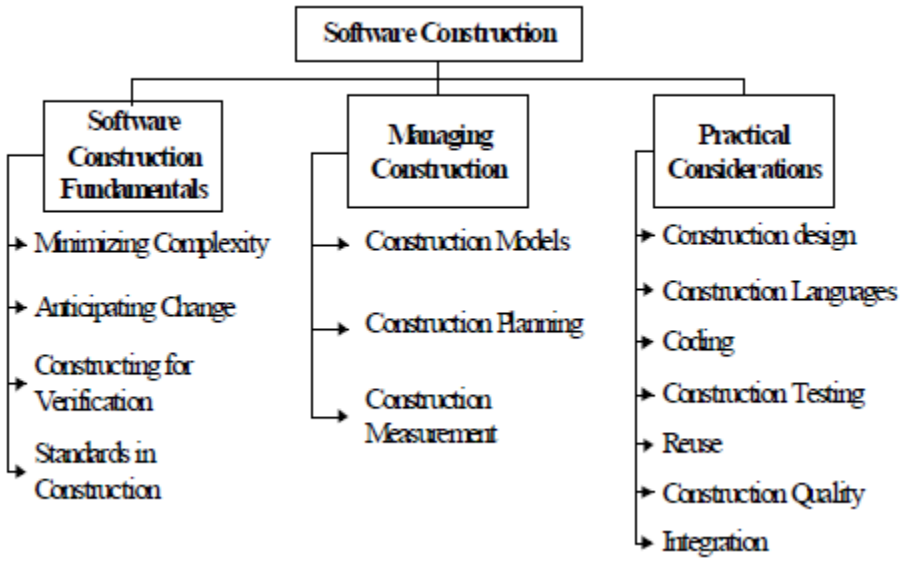

Figure 1: Breakdown of Software Construction Process [1]

1.1. 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.

1.2. 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:

	Time Detected				
Time Introduced	Requirements	Architecture	Construction	System Test	Post-Release
Requirements	1	3	5–10	10	10–100
Architecture	—	1	10	15	25–100
Construction	—	—	1	10	10–25

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.

1.3. 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.

2.1. CONSTRUCTION MODALS (ref chapter 2 and chapter 27)

Construction modals describe the development method followed to develop the product. In software engineering, there are various techniques for Software development like waterfall model, incremental model, scrum, extreme programming etc. According to McConnell, choosing the technique depends on the size of program. Ask the questions whether the program is small or large scale. What could be the target audience? How critical is the system? McConnell describes this in chapter 2 and chapter 27. Chapter 2 has already been discussed in section 1.1, where one uses the metaphors and analogies to decide on the development procedure. This section will follow chapter 27.

Chapter 27: How program size affects Construction

As discussed, the size of program matters a lot. McConnell argue that when person is working alone on the small-medium scale project, the ratio of error to correctness is always large. There is a lot of frustration experienced by the lonely programmer. But as team size increases, the percentage to deliver better product

increases as well. But this percentage is associated with the product scope and size as well.

Program size affects the number of errors that can occur. Following graph shows an idea of how size is linked to errors:

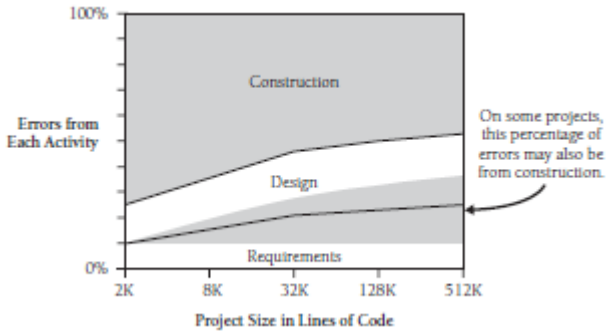

Figure 3: Program size and error percentage

As visible from graph, the small sized projects, with small team face a lot of errors in the construction. As the project and team size increases, the error percentage distributes to other activities as well. To reduce the error percentage, a lot of effort in needed to be done.

Another factor to high error rate is the wrong model choice. McConnell thinks that methodology is one of the important factor to the success of the project. Incorrect choice can lead to inconsistent behaviors. E.g. suppose your team is working on Word Processor. You are the team lead. After the analysis, you decide to follow the waterfall model. Waterfall model is the model which does an activity of Software Development Life Cycle only once. There is no going back once a part is done. You do the requirement gathering and follow the SDLC. In the construction phase, you find out the GUI (graphical user Interface) isn't very appealing. You ask your team to fix it and that fix introduces the bugs that weren't there before. You are near the deadline. You ask your team to rush the things and in the end, the bad looking product is delivered.

The problem in above case was the choice of development model. Only if you had chosen the Incremental approach, where you would deliver the small increments and get the user review after each increment and correct the system accordingly. That's why McConnell focuses to keep the product category and target audience in mind.

If you are not sure about what the target audience is (or you have very large target audience) and what is their preference, it is better to follow incremental

approach. One example of such projects is video games. You aren't always sure what people like. That's why gaming companies release the demos and get the gamers opinion. If you are sure about the audience size, you can follow waterfall method. Deciding what method to use depends on project size and category.

2.2. CONSTRUCTION PLANNING (ref chapter 27 and 28)

The success of project depends a lot on the careful planning. McConnell has discussed this issue in chapter 27 and 28. Chapter 27 has been discussed in section 2.1, where the program size the model choice was seen.

Chapter 28: Managing Construction

Managing the software construction isn't an easy job. The good management has direct relation with the construction planning. If the construction method is carefully planned ahead of construction, one can accommodate any irregular thing during the process.

First of all, encourage the good coding from start. One can advise the team to follow the good coding standards from beginning. Don't write the working code and then change it up into the standard code. During coding, ask other people to review one's code. Do the informal inspections to find out any standard violations. Give the appreciation to the good code.

Plan the change control in the construction process. As discussed in section 1.2 that requirements change a lot. Plan this activity ahead time and manage the process accordingly. Requirement change will lead to other changes as well. To handle these changes, one can adopt different methods like:

- Follow a systematic change control
- Assign groups to separate change requests
- Use version control software
- Estimate the cost of change and see whether it's worth changing.

Backup the code after each build. One is not sure about what can happen to code. Use version control software (VCS) and tools to monitor the changes. This

phase is also planned early. One can't just decide to incorporate the VCS in middle of construction.

Careful planning is the one root of success of software system. If one plans all the construction activities and anticipate where the changes can come, the whole process would be a lot smoother.

2.3. CONSTRUCTION MEASUREMENTS (ref chapter 25)

Construction measurements include activities that can be measured like code written, code change, code reuse, errors detected, errors fixed etc. In chapter 25, McConnell describes the code-tuning strategies that can be used for code measurement.

Chapter 25: Code-Tuning Strategies

Writing the good quality code is very important. A well written code always boosts up the performance of the system. But not everyone can write a good quality code. McConnell describes some of the ways to analyze the code and make the changes accordingly.

To see whether your code needs tuning, create the code according to the given standards. If the performance isn't above the scale, the code needs tuning. For this, first save the working code. Then, identify the potential spots where the bottleneck can occur. Mostly, these are due to deep nesting. But they can be due to incorrect design, weak algorithm etc. Apply one change at a time and see the result.

McConnell explains how to approach code tuning by the help of **Pareto Principle** or in general, **80/20 rule.** This says that 80% of the performance bottleneck is due to 20% code. This is a heuristic, but it has been tested a lot and it actually says the truth. There is one point of the code where all the processing and work takes the time. That is just the small code portion. McConnell explains that by this rule, one can narrow down the search to where change the code for performance increase.

Try to reduce the code size as possible. At first, the size reduction might not look promising. But it can help a lot in performance raise. If one piece of code is repeating a lot, make a routine or function of that redundancy. Also, don't try for optimization before any working code. First write the working code and then apply optimization techniques. But follow the standards of coding from start.

Maintain a balance between the I/O related operations and in-memory access. In-memory access are always faster than I/O operations. But one can't have all the memory for oneself. There is need to keep only required data in memory and rest should be saved in the file. If the file data is needed, read it and store it in the memory. Also avoid the expensive system call. Try to find the alternatives for the system calls. If one can't find the alternate, only then go for system call.

The choice of programming language is important here as well. This will be discussed in section 3.2. The idea of code tuning is just to increase the performance. Profile the code and see the potential change spot. Keep the 80/20 rule in mind.

3.1. CONSTRUCTION DESIGN (ref chapter 5)

In chapter 5, McConnell explains how the design can be considered as a construction activity.

Chapter 5: Design in Construction

Design isn't really considered to be construction activity. Design is created before the actual construction. Constructions is done according to the created design. This is correct for large scale projects. But for small-size projects, design is a part of construction. Most of the design is made when a person is actually coding. But still, the careful design procedure needs to be followed there.

Design is considered to be a **Wicked Problem.** These are problems whose solution can only be found after solving it completely or solving a part of it. In software, one might have an idea of how the thing would be designed. But it only be confirmed once it is created formally. When doing so, the possible errors are detected. If there are a lot of problems, there is need to change it. At the end, the design might look good. But the process of creating the design isn't well defined. Being a wicked problem, there is always chance of change while designing.

Design is all about preferences and optimization. One thing can be done on various ways. Based on the probable performance, one chooses among the possible selection. Design is needed as it clearly specifies the restrictions on the project. It tells what is possible and what isn't. There are things that can be included in design, but aren't necessary. These things are quality attributes. These QA also impose some sort of restrictions on the design.

There are some desirable traits of design. These include minimal complexity, Ease of change, reusability, loose coupling, minimum dependencies and portability.

These things can be achieved by the careful design process. First, design the overall system (first level design). Then, divide into packages or sub-systems (second level). On the third level, divide the sub-modules into the classes. Then, fourth level comprises of creating the data and functions inside the classes. In the end, define the interaction. By this way, any problem will be taken out before defining the interaction.

In the end, design in a heuristic. Even though there is a defined procedure, it can change based on the conditions. After the complete design, it should reflect the thing what is required. If not, design isn't consistent.

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.

3.3. CODING (ref chapter: 6, 7, 9-19, 24, 31, 32)

Coding is the important part of the construction process. Nothing will be constructed without coding. Coding is the essence of the construction. That's why Steve McConnell has allocated a great deal of portion to the coding techniques.

Chapter 6: Working Classes

In the earlier programming (1970s), the routine or function oriented programming pattern was followed. With the start of twenty-first century, the trend shifted towards the classes. Classes are collection of data and routines that share some behavior and responsibility. This concept is linked with Abstract Data Types (ADT). These are collection of types that work on data and operations. Classes do the same thing. The understanding of ADT is essential for the knowledge of Object-Oriented programming and classes.

With ADT and classes, one can hide the implementation details. Only required things are visible to others. Most of the details are hidden underneath the encapsulation layer. Changes in classes don't affect the overall program structure as changes are done within the classes. One can model the real world entities by concept of classes. By classes, one can combine the related data in one place. E.g. suppose one is developing program about employee information. Employee will have name, age, salary etc. There are different ways to model this situation. But the best

way is to create an Employee class and put the attributes there. In this way, there is logical relation between class and its attributes.

Another advantage is the encapsulation. By classes, one can minimize the accessibility of attributes and routines. Only those things that don't harm the abstraction are exposed. But some precautions are required for the classes. Only build those classes that have a logical relation between the attributes. Make it easily readable. The important thing Steve McConnell describes is the usage of friend classes. Steve says to avoid the friend classes as they violate the basic principle of encapsulation. Also, friend classes couple the classes among themselves. This puts a bad effect on the portability. It would be hard to move the particular class to other place for reusability.

Another issue to be discussed is the containment relation. Containment is when one class object is member of other class. This also cause the coupling. Use this as a last resort. Inheritance is better, but if logic doesn't allow inheritance, use containment.

In the end, Steve gives overview of why classes are important. Classes allow to model abstraction and real world objects. Complexity is reduced and changes are easy to do. Implementation details are hidden. And code is reusable. But avoid creating large classes. Optimal data member size is 7. If data members are increasing, consider splitting the class.

Chapter 7: High-Quality Routines

Routine is another name for the methods or functions. Functional programming has been in use for ages. Now, functions are part of classes. There is need to create functions. They help to avoid redundant code. Also, complexity is reduced as well. One can hide the details of what is happening. The change effect is limited by the creation of functions.

High-quality routines are supposed to provide with high cohesion. Such routines also have meaningful names. One should be able to get hint of what that particular routine is doing. There is balance of how many parameters are being passed. The parameter list isn't very big. Also, only that data which is required is passed. Any unnecessary detail isn't passed. The idea is to provide the required information only.

Chapter 9: The Pseudocode Programming Process

No one starts to code straightaway. There is the designing and planning involved. For the better results, the technique for Pseudocode Programming Process PPP is used. The idea is that instead of writing the code, write the main idea of how the thing would work. Write the pseudocode in the source file, but check it manually whether it is correct or not. This applies to all aspects of the programming.

Pseudocode can be good or bad based on how one is writing it. Use the English like, informal language. Don't use programming syntax in pseudocode as it spoils the fun of pseudocode. But write it in such a way that creating code from it should be very easy. By writing good pseudocode, one can turn it into the comments, which reduces the overhead of writing the comment. In other words, PPP makes the coding very easy. Another benefit is that one can change it easily. As it isn't coded, so one can iterate till the correct version is obtained.

Chapter 10: General Issues in Using Variables

Variables are the data containers. They hold a single value at a time. One needs to have an information of variables and their data type. After that, one can handle the data. Since variables are so helpful, there is need to use them properly. Steve discusses some guidelines to avoid the variables related issues.

Declare the variables as one types them. Initialize all the variables. That's one main issue when an uninitialized variable causes the system to crash. Use good names that convey the intent of the variable. Initialize variable close to its declaration. Avoid using the global variable. Try to have minimum level scope of variable. High level scope means there are more places where that variable can change. In the end, use one variable for one purpose only.

Chapter 11: The Power of Variable Names

As explained earlier, the meaningful name conveys the information of what is happening. In any program, there are a great number of variables. It is best to give each one of them a name that explains the intent of that variable.

Consider the example:

X = (y+z)/2

If x>10{// Do Something}

Else {// Do other stuff}

What is the purpose of x,y and z? It is not visible from the above code. On the other hand,

average = (firstOHTMarks+secondOHTMarks)/2

If average>10{// Do Something}

Else {// Do other stuff}

This is a lot better as it is clear what each variable means. In general, variables should fully represent what variable is for. Use the naming conventions that differentiates between local, class and global data. One can use small abbreviations of the real world object that is being modelled as a variable name. The verdict is to use conventions, make meaningful names and don't use the similar looking variable names.

Chapter 12: Fundamental Data Types

In this section, McConnell describes some of the fundamental data types:

- Integer are the data type that hold simple Integer value like 1,2 -1 etc. No decimal places are allowed
- Float allows the holding of decimal value like 1.22 etc.
- Character hold a single non-integer value like 'a', '$' etc. They can also hold integer values, but they aren't treated as integers
- Strings are combination of characters such as "Data", "type1" etc.
- Boolean are used to check truth value. Boolean cab only be 0 or 1.
- Enumerated are used when one knows all the possible values for a variable. The variable can be assigned any known value at any time
- Named constants are defined to use a meaningful value throughout the program
- Arrays are the group of data of same type. It uses indexing to get/set data.

Chapter 13: Unusual Data Types

In this chapter, McConnell discusses some of different data types. These vary a lot from the data types discussed earlier. Some of new types are:

- Structure are the data types that consist of various data types. They are available as struct in C and C++. They resemble classes, but only that struct have default accessibility scope as public. They are used when one single variable can't describe all the contents.
- Pointers aren't usually considered as data type. Instead, they hold the address of a variable residing in the memory. Pointers are references. They refer to other variables or collection of variables or any structure. Pointer handling is needed to be done very carefully. Any irregular access can result in crashing of program. Try to minimize the locations with pointer related operations.
- Global data is the type of data that is available to every statement of the program. Global data use is being avoided. It is because they violate the principle of information hiding by exposing all the information.

Chapter 14: Organizing Straight-line Code

This chapter is all about the statement centered programming. For some program to work, there is need for the sequence of events to occur. McConnell presents his idea that if this is the case, structure code such that all the dependencies are clearly visible. One should be able to figure out the sequence. The better way is to make routines of the dependencies. The parameters should identify how there is dependency between two routines.

There are also statements where order doesn't matter. In such case, try to keep the related statements as close as possible. For such cases, McConnell suggests that read the code from top to bottom and understand what is happening in the first attempt. If not, there is problem in the ordering of the code.

Chapter 15: Using Conditionals

In this section, the usage of if-conditional and case-conditional has been discussed. If statements involve the statements like if, if-else, if-else if- else if – else. There is nesting involved as well. If statements are used when there is decision of path based on the conditions. E.g. if (age > 30 && limit > 10); this is showing the condition inside if and based on what the truth value is, the path is taken accordingly. McConnell focuses on chaining and small level nesting to test the conditions.

Case statements are used when decision is to be made based on one variable. Such things can be done with if statements, but switch/case statements are better. In case statement, keep one action/case. Don't overdo the action of a particular case statement. Use break after each case. Provide some default values if no case is matched. In the end, McConnell says that each statement has its own flavor. Choose the one that suites the situation best.

Chapter 16: Controlling Loops

Loops are the iterative control structures. In loops, processing is done in some iterations. There are different types of loops. Following is the table that McConnell used in the book to show different loops types:

Language	Kind of Loop	Flexibility	Test Location
Visual Basic	*For-Next*	rigid	beginning
	While-Wend	flexible	beginning
	Do-Loop-While	flexible	beginning or end
	For-Each	rigid	beginning
C, C++, C#, Java	*for*	flexible	beginning
	while	flexible	beginning
	do-while	flexible	end
	*foreach**	rigid	beginning

Table 2: Different loop structure

One should be careful in the choice of the loops. Following are the cases that tell choose what loop in which scenario:

- Use while loop when it is not sure how many iterations will the loop take. This is the case when there is some condition and one doesn't know when that condition will occur.

- Do-while is similar to while loop. Use do-while when someone wants to execute the body of loop at least once.
- For loop is used when there is need to do the iteration for fix number of times. It is known ahead of time of how many iterations will take place.
- Foreach loop is used when some specification operation is to be performed on each element of a list or array.

The choice of loop depends upon what the situation is. Just make sure that loop will always end i.e. there is the breaking condition. If not, an infinite loop will keep on iterating that can lead to issues.

Chapter 17: Unusual Control Structures

This chapter discusses the unusual control structures. These are structure that change the flow of program, but are different from other control structures.

- Multiple returns from a routine are such type of structure where if answer is found, that answer is returned. Otherwise, some default value is returned. This is seen in the case of routine call. Suppose a routine checks whether a number is prime or not. The return is either TRUE or FALSE depending upon number. Use such routines when you are sure of what expected output is.
- Recursion is the case when a routine calls itself. In the routine, there is call to itself with some reduced data set or variable conditions. In recursion, a smaller version of a problem is solved. Use recursion when one is sure that problem can be broken down into smaller versions. Also, make sure that there is one base case from where there will be no more recursive calls.
- Goto statements aren't used very much. They are known for breaking the sequence of the program. They still exist in some programming languages. But their use is discouraged.

Chapter 18: Table-Driven Methods

In table-driven methods, one looks up the values stored inside a table, rather than using the conditional statements to find one. McConnell has claimed that virtually everything possible with conditional can be done with the table-driven

methods. He addresses two issues in table-driven methods. One is how to lookup the data in table. One can use some data to access it. But here he tells the second problem of what sort of data/key should be used. The key should be unique so that each record is unique. There are different ways to access the table which are:

- Direct access means one pick up directly what is required. Based on the table structure and key knowledge, pick the underlying value directly.
- Indexed access brings one piece of check before getting the actual data. First, one picks up the key from an index table. Then, using that key, pick up the actual data from the table.
- Stair-case access is another approach. In this approach, the keys are in form of some range with same output against each key. By the range, one can reduce the space required for the table.

Chapter 19: General Control Issues

In this section, Steve McConnell address some of the general control issues that are faced by the programmers.

- Use Boolean variables for the Boolean conditional test. Some programmers use the integers as 0 or 1 to check it. But McConnell says this approach isn't correct. Use the data type that is there for 0 and 1.
- Separate each logical block or compound statement. Use opening and closing brackets or proper indentation to separate out the different blocks. This help in better understanding and readability of the code
- Be careful of **null** values. There is special need to take care of null values in conditional and loops. e.g. in while(array[index]!='\0' || index<lengthArray), the null case will occur as array length check is done later in condition and memory access is done earlier. Remember that AND and OR are not commutative in most programming languages.
- Avoid deep nesting of loops and conditionals. Deeply nested conditions are always bottleneck to performance. It also implies one isn't sure about the program logic. If nesting level is deep, refactor that code by if-else statements or case statements.

Chapter 24: Refactoring

This section is about the refactoring which is the code change process. For any software, code change process happens every time. There is either change request by customer or there is need to improve the code. McConnell discusses the second reason, which is refactoring.

Refactoring can be done for many reasons. A particular code is duplicated, or one function is too long, or there is some performance issues in a particular routine, routine has a large list of parameters, coupling between the modules, improper organization and many other such reasons. All these issues don't allow the system to perform as it should. McConnell suggests that refactoring should be done in such cases. There are various refactoring strategies that can be applied accordingly.

- Data-level Refactoring where all the changes are related to one variable. That variable is either global or being used a lot or other such reasons. If it is global data, try to reduce the accesses to it.
- Statement-level refactoring involves the change of more than one statements. These statements can be consecutive or in same routine or in same class.
- Routine-level refactoring are changes at the function level. These can include splitting one large routine into small routines, modifying the parameter list, using a simple algorithm in routine or combining similar routines into one routine.
- Class-level refactoring relates to changes in class like data member addition or removal, converting two classes into one or vice versa, removing the deep level inheritance etc.
- System-level refactoring is about the interactions within classes. In this phase, use the correct relation between classes. Provide the error handling for any possible error that such interaction can give.

Refactoring is a difficult procedure. It is better to save the original code before any change. Make check points when a successful refactor code is made. In the end, McConnell advises to use the refactoring to make the code quality better.

Chapter 31: Layout and Style

The way code is structured depends a lot. Writing a well-formatted code is indeed a hard thing to do. But when done so, it makes life easier, not for oneself but for others as well. The first thing for good code is to layout it properly. Separate each

line. Don't mix the comments and real code. Make the code structure look good. One should be able to distinguish the different parts of the code.

Whenever coding is being done, only a small part of the code is to be meant to be read by computer. Major portion comprises of fact that code should be readable by humans. The good layout achieves the logical structure of the code. A person can easily understand what is going on in the picture.

For better quality code, use the whitespaces to separate the content. Don't put stuff all together. Whenever there is change of logic, show it properly. Use the grouping and proper indentation to show the parts of code. For conditional statement, use the spaces between the conditions. Use opening and closing braces properly to separate the multiple conditional statements. Use one statement per line. This is very good for readable code. Separate the comments with spaces and tabs. Use the whitespaces to separate out the functions. This might look silly, but in the end, one can ensure that code is readable.

Chapter 32: Self-Documenting Code

Documenting the code is the necessary part. It explains what is happening in the code. Even though code readability is focused a lot, code documents explain the logic. There are two types of documentation: **Internal and External.**

Internal documentation is done inside the code. This is done with the help of comments. Comments explain what is happening inside the code. They act as document which one can read to understand the code. But in practice, comments aren't considered to be internal documentation. The real internal documentation is by programming style. In layout section, readability was discussed. That readability includes the logic as well. One's code should explain automatically what is going on. E.g. doStuff(x) isn't conveying any information by itself. One can put the comment to explain what it is doing. But better version is like calculateBonus(extraDays). Its telling what to do and on what data. Sure there is more writing involved, but the final result is better. The good programming style can explain the code without even writing the comments (McConnell focuses to write comments).

Next part is the discussion about the comments. There is always dispute whether to write comments or not. Study shows that some programmer thinks it is waste of time to write the comments while some thinks it is better to write the comments. The thing is if someone is writing the comments, make sure that

comment explain or assist the code. Don't write misleading comments. If there is inconsistency between comments and the written code, don't follow what comment says. Comments can be of various kinds. It includes repeat of code (comment same as the code), explanation of code (explaining the statements), summary of code (overview of what code is about) and intent explaining comment. Use comment style according to the condition. Explain the things that code isn't able to explain.

Comments contribute a lot for the internal documentation. Now, a little bit about the external documentation. These are type of documentation that isn't written inside the code. It's outside the code. Such documents are the detailed explanation of the code. Usually, such documents are auto-generated based on code structure. Some programming languages provide syntax solely for outside documentations. Javadoc is an example, where details about the code are written in special format and one can generate the external document about it.

3.4. CONSTRUCTION TESTING (ref chapter 22, 23)

Whenever some product is being made, there is need to test it whether it is working fine or not. Testing checks whether standards are being followed or not. Same is the case with construction testing. Whenever someone is in the construction process, there is need to verify that the product being made is correct or not. In chapter 22 and 23, Steve McConnell explains the techniques of Developer Testing and Debugging respectively.

Chapter 22: Developer Testing

A person who codes a particular piece of program knows it better than anyone else. He/she can detect the problems in that part better than anyone (most cases) by the testing. Testing is of various types, including unit testing, module testing, integration testing and system testing. These are the main categories. Developer testing can come under any of these types. The better way is to use the combination of above mentioned techniques to get the better results.

The purpose of testing is to find the errors. A positive test is the test that breaks or crashes the system. The approach for testing is to check each requirement. Open up the Software Requirements Specifications, and see which requirement has

been implemented. If implemented, test it with various inputs. See the behavior of the system. Log any error and remove it.

There is need to take the testing seriously. Following graph shows the percentage of developer testing in complete software development life cycle:

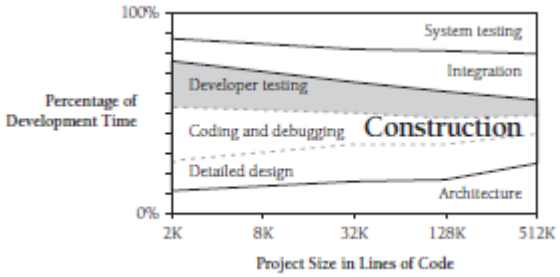

Figure 4: Developer Testing and project size

As graph indicates, developer testing take a lot of time for small projects. It is because team size is small and an individual takes a lot of time for testing. There are different ways to test the particular module. There are **Test First** or **Test Last** techniques. Test first says that develop the test cases first, and program accordingly. Test last is that develop the module and then write test cases. Both are useful in their own way. But test first is used a lot.

Even though developer testing is important, there are some limitations to it. First most, developers are emotionally attached to their code. They don't like to say that their code is buggy. They tend to write the clean test cases, test cases that don't crash the system. Because of this, the complicated cases that can down the system aren't tested. Only few programmers test their code with breaking test cases.

Testing doesn't assure that the system is bug free. It is because for a particular piece of input, there are many possible combinations. Out of these combinations, only few are breaking cases. When testing, one should keep that in mind and write the cases that provide new information. Don't test with redundant test cases.

Most problems occur when there are a lot of conditional statements. E.g.

```
If (age >40 || salary > 50000|| experience >=10)
        // DO something
```

This particular piece is testing three conditions. But there is OR between the statements. If age is greater than 40, the code inside **If block** will execute. If someone wants to access the information, they can do so easily. If anyone provides invalid

values for salary and experience, but provides age >40, there is no way to check for that invalid input. That's why design of test cases is very important for such statements.

Another testing technique is **Equivalence Partitioning.** This method comprises of defining the valid and invalid classes for the inputs of the system. Valid classes define the criteria acceptable by the input and invalid classes tell those which are not acceptable. For example, there is input for an Employee Age at a particular point in program. The valid classes will include age>18. Invalid can include age <=0. By such partitioning, the inputs are provided to check the system behavior.

Other techniques include **Boundary-Value Analysis.** This is an important testing technique. It checks how system behave by the values which are:

- Near the Boundary
- On the Boundary
- Outside the Boundary

This technique is used a lot for numerical inputs. Studies have shown that programs depict unexpected behavior under the boundary-value testing.

As test cases are made by humans, so there is chance of error in the test cases itself. Practices have shown that erroneous test cases can hinder the development a lot. One wastes a massive amount of time figuring out the problem when there is no problem. It is better to always check the test cases manually to see that they are not wrong.

With the increased automation, testing is being automated as well. There are tools available that can test the code automatically for the errors. Automated-Data generators based the test data on the given constraints. They also define what the expected outcome is.

The last thing McConnell discusses is the **Retesting or Regression Testing.** It means that whenever system is gone through some change, test it with the existing test cases to see they are working correct or not. This helps for fast testing as one doesn't need to make new test cases for change. The old test cases (most of them) should give the correct result on new piece of code.

Chapter 23: Debugging

Debugging is the process of finding the root cause of a bug and fixing it. It is different from testing in a sense that testing is the detection of bugs and debugging is

the correction of bugs. Debugging is one way to figure out and remove the defects in the system. Even though debugging is a common term, not many programmers are well aware of how to use it. Only the most experienced programmers can find out the errors quickly. Following tables shows this:

	Fastest Three Programmers	Slowest Three Programmers
Average debug time (minutes)	5.0	14.1
Average number of defects not found	0.7	1.7
Average number of defects made correcting defects	3.0	7.7

Source: "Some Psychological Evidence on How People Debug Computer Programs" (Gould 1975)

Table 1: Debugging and programmer experience

McConnell emphasizes that even fastest programmers take about five minutes to find bugs in the known code. That time margin is the reason to discover about the debugging. Steve McConnell considers the bugs as an opportunity to learn. It gives developer a chance to learn deeply about the system. One learns what type of error that person is making. Not only that, but how to fix that problem.

Defect finding isn't an easy job. Locating the hotspot where the issue can be is serious work. Whenever an error is detected, try to record the conditions in which the error happened. Try to repeat the situation with similar data. E.g. in one's code, the input of 29 is causing issues. To learn more, one must give inputs like 30, 29.1, 29.5 etc. to see whether issue still remains or not. If it exists, record the path followed by the input. See what is happening along the path and try to locate the bug point.

After finding the error, there is need to fix it. The fix should be like that it doesn't affect the other parts of the system. For the correctness, understand the overall system and then the part one is trying to fix. The complete knowledge of the system is required to make the change. If one isn't getting at the solution, try to relax. Relaxation gives mind a way to think about possible ways. Also don't fix the symptom, but fix the problem. E.g. in above example where 29 was causing issue, don't make statement that **If val==29→ end/ return.** This is wrong approach. What if the program doesn't work for 43 as well? One can't keep on adding special cases. Try to find the root cause. When fixed, check the new code. Make sure it is working correct.

There are various tools that can be used during the debugging. One tool is Version Control System. If the fix breaks the system, one can retreat to the code by the VCS. Maintain the versions while debugging. Make use of compiler warnings.

Compiler issues warnings because someone else had faced that problem and therefore, programmed the check in compiler. Don't skip such warnings.

3.5. CONSTRUCTION QUALITY (ref chapter 23)

Construction quality is the standard to check the quality of code. The quality measures that how well written the code is. Well-written means that error rate is very low. McConnell describes the quality check with the help of debugging. That has been discussed in section 3.4.

3.6. 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.

Overview (ref chapter 34)

This section is the overview of various things that have been discussed. It will focus on reducing complexity to building a high quality software.

Chapter 34: Themes in Software Craftsmanship

This part is the summary of what have been discussed. The first most step was the complexity reduction. No one can build extensive software in one day. There is a lot of thinking and planning involved. The concept is to reduce the complex

picture into easier partitions. Breakdown the system into smaller sub-modules. Look for reusable code.

Complexity reduction also matters in the coding. Don't use deep inheritance or deep nesting. In a great depth, the maintenance and debugging provides a lot of trouble. Don't use the goto statements as they break the logical sequencing. Use meaningful routines and variable names that others can understand. Define the error handling procedures to handle invalid data. Use advanced concepts of abstraction and encapsulation.

Another major concept is the correct choice of the model, team and the process. Use the model according to the system category. By category, decide what development method should be used. Also, choose the team that goes well with each other. Don't select members that are more likely to conflict with one another.

Coding standards is another important point. Write good quality code. Write the code that others can understand. Computer can understand whatever one writes. The ideal code is one others can get idea of without any external help. Make the code readable.

Choose the programming language that interest the team. Program into the language. If some techniques aren't available in your desired language, try to create those things yourself. Don't choose language that has extensive support for different techniques.

Maintain the quality of the code by the testing and debugging. Don't be afraid to change the code. Program according to the problem domain. Keep the requirements in mind. Don't code what isn't required.

This was just an overview of what was discussed. Good quality systems don't come into existence on their own. There is great deal of procedure involved, that one should follow to create the high quality software.

References:

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