CSC 478: Software Engineering Capstone

Module 3

Agile Software Development, Configuration Management, Requirements Engineering

Objectives:

1. Be familiar with the basics of several agile methods, and prototyping, and how these processes can be useful.
2. Be aware of some of the limitations of agile methods.
3. Understand what software configuration management is, and why it is necessary.
4. Be introduced to the following fundamental configuration management activities: configuration management planning, change management, version and release management, and system building.
5. Understand the importance of CASE tools with respect to configuration management processes.
6. Understand the difference between user requirements and system requirements.
7. Understand the difference between functional and non-functional requirements.
8. Be introduced to the fundamentals of requirements engineering.
9. Understand the importance of validating requirements.
10. Understand why requirements management is important.
11. Understand why it important to establish the boundaries (scope) of a software system.
12. Become familiar with some of the techniques commonly used to model requirements for a system.

Reading:

1. Sommerville, chapters 3, 25, & 4.

2. Web sections for chapters 4 & 25 (available at: https://iansommerville.com/software-engineering-book/static/web/web-sections/)

    a. Agile Changes (Chapter 25)
    b. Domain Requirements (Chapter 4)
    c. Requirements Standards (Chapter 4)
    d. Requirements Reviews (Chapter 4)
    e. Changing Requirements (Chapter 4)
    f. Traceability (Chapter 4)

Assignment:

Make one original post in the discussion forum for Module 3, and respond to the posts of at least two other students.

Quiz:

Take the quiz for Module 3.

3.1 Agile Software Development

One of the key criticisms of the software process models we discussed in Module 2 is that they all have difficulty dealing with change. During the development of a software project, many types of changes can occur, some of which include:

1. The customer or end user may demand arbitrary changes in the functionality. For example, customers may decide they want to add an extra feature, or they may want to change the way the originally specified functionality works.

2. The business rules of the organization requesting the software may change, requiring changes to be made to ensure the software satisfies the new changes. As an example, an organization may have requested software employing a Microsoft SQL Server database, but during development, the organization's management may have struck a deal to obtain licensing for Oracle's DBMS, and they decide to switch from SQL Server to Oracle.

3. New laws may be enacted, or existing laws may be amended, that may require changes in the software's functionality. A hypothetical example here might be that tighter information security laws adopted by the Department of Homeland Security change the way that certain types of information must be protected. If this wasn't anticipated during the requirements elicitation phase, changes must be made to comply with the new laws.

4. During development, it is possible that the chosen approach for solving a particular problem is found to be flawed, and a different approach must be taken.

5. Software and/or hardware dependencies for the software under development may change. The new software may need to be redesigned to work with the updated components.

6. A competitor with new insight into a problem or end user market may emerge that forces a development team to upgrade their own requirements and/or design in order to remain competitive.

7. The precise requirements for a project may be difficult to define, due to customer indecision or complexities inherent in the problem being solved, but the development team may be required to build a product anyway, possibly with a short deadline.

8. The marketability of a particular software application may change before the software is completed.

We talked about how bad the waterfall model handles change, but the fact is that none of the models we discussed in Module 2 are really good at handling changes. The spiral model is good at anticipating risks—and the chances that software requirements may change is one of the many risks that must be considered. The incremental model may be able to handle changes if they involve components that have not yet been developed. But even these models can be somewhat laggard considering the rapid pace at which certain aspects of the world change. Agile software development models attempt to maintain productivity in spite of change, by defining software engineering processes built around the belief that changing requirements are inevitable, so the best way to deal with them is to have a process that expects them and can accommodate them, as opposed to grinding to a halt when they occur. All agile processes have two common characteristics:

1. They prescribe rapid delivery of small, functional increments of software.

2. They are designed to accommodate continuous changes in requirements.

In this module we will discuss Extreme Programming (or XP for short) and Scrum, two of the most popular agile software development processes.

3.2 Extreme Programming (XP)

Kent Beck and Ward Cunningham , along with 15 other software engineers, developed the Agile Manifesto, in 2001. This manifesto can be found at http://agilemanifesto.org/ for those who are interested in viewing it. Extreme Programming is only one of many agile methods. Sommerville does an excellent job of summarizing what Extreme Programming is all about. However, I do want to re-emphasize some of the major principles XP uses.

1. There is always an on-site customer who can be consulted should any questions arise. The customer decides what functionality should go into each increment of the software.

2. Functionality is never added early. XP follows the YAGNI principle (You Aren't Going to Need It). This avoids the problem of introducing features early on that are later deemed unnecessary.

3. The time window for each increment is very small - on the order of a few days or a week. By contrast, many other agile and rapid development models allow several months per increment.

4. Tests are always written before the code. Code is written to ensure the tests pass. This is completely reversed compared to other process models.

5. Pair programming is employed. This involves two programmers sitting at one computer. One types, while the other proofreads, looks up material in reference books, and shares ideas for how to make the code work. At any point, the person who is proofreading can switch places with the person who is typing. Code is added to the repository only when it has passed all tests.

6. There is collective ownership of all code. This means anyone can pull code out of the repository at any time and improve on it.

7. 40 hour work weeks are used. Overtime is limited because of the potential negative effects it can have on performance.

8. Documentation is virtually non-existent. The source code essentially documents itself. The philosophy is, no one ever reads it anyway, so why waste time creating it.

Many of these principles sound great (especially the 40 hour work weeks), and when employed by a skilled Extreme Team can produce quality software very quickly. I don't know that I agree with writing all tests prior to coding. That works fine for user-based acceptance tests, but I'm not convinced it generates fewer faults. I also do not agree with the "no documentation" philosophy. I do agree that documentation can be overdone, and it may not be of much use for one-off projects with a known limited life span, but just try to maintain a project written by someone else with no documentation. In many cases you would almost be better off rewriting the whole thing from scratch!

3.3 Scrum

Scrum got its name from the rugby activity where a mass of players comes together to fight for possession of the ball. That doesn't mean people on a Scrum team fight, however. Scrum is primarily a management technique that obeys the following principles:

1. Small working teams are used to maximize communication while minimizing overhead.

2. Software increments are delivered frequently. Each increment is functional.

3. Both work and people are partitioned into low-coupling packets. Each packet is relatively self-contained and should not have to interact significantly with other packets.

4. Testing and documentation are performed continuously as the software is built.

5. The final product can be declared "done" at any time. For example, it could be declared done when a competing product is about to hit the market, in order to get the first share of the market. Or, it could be declared done if the customer decides they don't want to wait any longer. Since each increment is functional, the customer will always get a working product.

6. A "backlog" is always maintained that contains a prioritized list of project requirements that still need to be completed. The project manager updates this list as needed. Work in the backlog is charted for each increment over time, and the result is called the "burndown chart". The burndown chart is a high-level document that shows how much work needs to be done for the entire project.

7. Work is done in "sprints". A sprint is a work unit required to complete one or more requirements in the backlog. Sprints must fit within the time window specified for each increment (e.g., 30 days). The requirements in the backlog that correspond to a sprint are frozen until the sprint has been completed. No changes can be made to those requirements during the sprint. This allows the sprint team to work in a stable environment. When a sprint has been completed, it is removed from the backlog, and is no longer shown on the burndown chart.

8. Each day there is a scrum meeting. Scrum meetings are very short, lasting only about 15 minutes. During the meeting everyone on the team is asked 3 questions:

   a. What did you do since the last team meeting?
   b. What obstacles are you encountering?
   c. What do you plan to accomplish by the next team meeting?

In addition, software increments must be demonstrated to the customer for validation. Management's primary role in a scrum project is to remove obstacles that are hindering the team, not to direct the development of the product. This can include things like buying new software or hardware, providing more comfortable chairs, or even bringing in pizza for lunch so as not to disrupt a team that's on a roll.

For more information on Scrum (sometimes written as SCRUM, although the name is not an acronym), see Scrum.org, created by Ken Schwaber, one of the creators of Scrum; or ScrumAlliance.org, an organization also founded by Schwaber.

3.4 Configuration Management

Many of you have probably done some type of configuration management before. Some of you probably have not. Configuration management is crucial in larger projects, but less so in small projects. What exactly do we mean, though, by the term “configuration”?

A software configuration is the collection of all components necessary for the software to meet its requirements. Configuration management includes any activity that results in, or could potentially result in a new version of the software being built. As noted by Sommerville, configuration management is primarily concerned with controlling four activities: 1) change management, 2) version management, 3) systems building, and 4) release management. These are described further below.

Change management

Changes can occur in the following phases of development:

1. ***Requirements.*** Changes in requirements are changes in the desired functionality of the software, and can be submitted by the customer, requested by end users, or mandated by changes in business rules or laws.

2. ***Design.*** Changes in design can exist independently of changes in the requirements. The purpose of the design phase is to specify how the requirements will be met, but if there is more than one way to meet a particular requirement, it is possible to decide on one approach initially, and change to a different approach later. The requirements remain unchanged, but the way in which they will be satisfied changes.

3. ***Implementation.*** Just as there may be more than design that will meet a requirement, there will usually be more than one way to write the code that satisfies the design. More efficient code to solve a particular aspect of a problem may be discovered during the coding phase, for example. Both the requirements and design may remain unchanged while the code is altered.

Equally important to controlling what changes should be made to the software, is what changes should not be made. Just because a change has been requested doesn’t necessarily mean it must be implemented. Some change requests may be frivolous, and should therefore be ignored. Some changes will require such extensive revamping that they can only be practically included in the next major release version. Some changes may even be impossible to implement.

Version management

There are two separate software configuration tracks that are typically maintained for software projects: 1) development and 2) release. As software is changed during development, new development versions are created to keep track of the changes. This is vital, since if it is determined that a particular set of changes will not work as expected, there must be some way to easily roll back to a previous version and continue development down a different track, without having to start over from the beginning. In addition to different versions that result from changes made to the software’s functionality, there may be separate versions to support different languages (e.g., English, Japanese, Spanish, Arabic, etc.), or different levels of capability (e.g., Standard, Professional, Enterprise, Ultimate).

Systems building

Most complex software exists not as a single, monolithic, executable file, but as a collection of separate components. Systems building is responsible for assembling the correct set of components to complete a working version of the software. Systems building may or may not involve compiling and linking source code. The number of individual components for a given software application may be quite large, and each component may have multiple versions of its own, since a single component may be developed as a separate software project. Often, makefiles are used to keep track of component dependencies, and assemble the components in the proper order.

Release management

As software evolves after it has been released, upgraded versions are periodically made available to customers, containing new or modified functionality. Compatibility issues frequently arise when a software package has dependencies, each of which may exist in multiple release versions themselves. Although it is most common to think about software-based releases, computer hardware also exists in multiple release versions, and as such can contribute to both hardware and software release compatibility. Apple's OS X operating system, for example, is tightly bound to computer hardware configurations. A given Apple computer model will typically only be upgradable to a certain version of OS X. Further operating system upgrades require hardware upgrades as well. Most organizations routinely support several release versions of software, so it is essential to maintain compatibility matrices for each version in order to facilitate customer support.

3.5 Approaches to Version Management

There are several ways to approach version management during development. One simple method that works well for very small projects is to simply make a dated backup of all source files and components into a folder on a disk, prior to beginning any new work. Each folder therefore represents a different version. This makes it easy to find things chronologically, but you will need to include some sort of list that will clearly indicate what was done on each date, to avoid having to laboriously search through all the folders to find a particular piece of code. Another disadvantage to this method is storage space. The longer it takes to complete the project, the more space will be required to store each day's version. Additionally, depending on what work is done on a given day, it may be necessary to have more than one separate version for that date.

A second method relies on a numbering scheme for identification. A common numbering scheme uses three integers delimited by periods (e.g., 7.1.2). The leftmost number represents major changes in functionality. The rightmost number represents very minor changes. The middle number represents changes that are significant in importance, but which do not significantly change the functionality. Unfortunately, there are no good standards to define what distinguishes a major modification from a minor modification from an intermediate modification. The dividing lines are left to the discretion of the developers. The advantage to this scheme is that it is easy to tell which version is the latest—it will always be the one with the highest number for each section of the version ID. Often, though, developers will use more than three numbers in their versioning schemes, and may include additional identifiers such as "build" identifiers. The overall version number can grow to be quite complex.

For projects where storage requirements are strict, it may not be practical to store full copies of all components for every version. In those cases, the following storage rules can be applied to make for more efficient use of available storage space:

- If a component was not changed from its previous version, do not copy it; simply maintain a link to the most recent version of that component.
- If a component was changed from its previous version, instead of storing a new copy of the entire component, only store the difference between the new and old components. Files that contain these differences are known as **delta files**.

While this method may allow for more efficient storage, it adds an additional level of complexity to the management aspect. It is essential to have a software tool capable of keeping track of the differences, and building a complete component from all the delta files.

Version control for moderate to large projects is most easily accomplished using one of many available software tools that automate the process of keeping track of the different versions. Examples of such tools include: CVS, Subversion, Rational ClearCase (IBM), and Mercurial. Version control tools are invaluable assets for development teams, since they relieve the developers of the burden of maintaining and communicating version information amongst themselves. Many version control tools also provide file checkout features to ensure a particular file can only be modified by a single individual at a time. Only after a checked out file has been checked back in may another member of the team gain write privileges for that file.

3.6 Requirements Engineering

Suppose I asked you to build for me a software version of a particular game — Monopoly, for example. What would you do? Would you simply say, "OK", acquire a copy of the boxed game, and begin building a version for the computer? — probably not. How would you know what type of computer I have? What if I wanted the game board to be customizable, to match some of the variants that exist on the market (e.g., Star Wars Monopoly, Simpsons Monopoly, etc.)? What if you built the game for a monitor using 1280 x 1024 resolution, but my monitor could only handle 800 x 600? What if I didn't like the way you had provided for moving the player tokens on the screen? In short, if you had immediately begun building the software based only on my request, you would have been making a lot of assumptions which may or may not have been valid. You would have skipped the first phase of the software life cycle - the requirements phase.

The requirements phase of the software life cycle is arguably the most important of all the phases, because it is during this phase that you try to figure out precisely what it is that you are supposed to build. I emphasize the word, "what", because often when figuring out **what** you're supposed to build it is natural to also think about **how** you will build it. However, these decisions should be left for the design phase. So the general rule of thumb is:

1. Requirements Phase - **what** are we supposed to build?
2. Design Phase - **how** do we build it?

The dividing line between the "what" and the "how" is not always clear cut. For example, suppose one requirement states that the software must store information on employees in a company. This is clearly a "what" - something the software must do. There are many ways this information can be stored. In some cases, the customer may not care how the information is stored, so long as it is stored, and can be retrieved and modified as needed. In such a case, "how" the information will be stored is a decision that should be left to the design phase.

Sometimes, though, the customer may explicitly require how something should be done. Perhaps the company has an Oracle database management system already installed and running, so the customer may specifically request that your software must be able to connect to an Oracle database, and store employee information in an Oracle table. In this case, "how" the information should be stored becomes a requirement, and should be decided upon during the requirements phase. Experience is the best indicator for determining whether a particular detail should be part of the requirements or part of the design.

The decisions that must be made during the requirements and design phases combined can be viewed as a continuum that shifts from one phase to the other, depending on the amount of understanding the customer has about what they want the software to do. As the customer's understanding increases, more decisions will be made in the requirements phase, leaving fewer decisions to be made during design. A lower level of understanding on the part of the customer means fewer requirements, which in turn means more decisions must be made by developers during design.

3.7 Characteristics of Good Requirements

What defines good requirements? Differing opinions exist, but generally speaking all requirements should strive to achieve the following characteristics:

1. **Correctness**

   The requirements should specify the proper functionality. This may sound like a no-brainer, but depending on the complexity of the problem being solved, it may be difficult to ascertain the correctness of some requirements.

2. **Consistent**

   All the requirements should be consistent with each other; i.e., there should be no requirement that contradicts another. The more people that are interviewed during requirements elicitation, the greater the likelihood that inconsistency can creep into the requirements. Everyone must be on the same page, or what you will end up with is a requirements document that specifies several variants of a software solution.

3. **Complete**

   Completeness of requirements means that all possible entities, states, transitions, input, output, constraints, etc., should be accounted for and handled by one or more requirements. Any important negative requirements (i.e., things the software should not do) should also be included. In practice, completeness is probably never truly achieved, since to do so would require that the precise positions and sizes of all graphical objects be specified, for example. What is usually achieved in a "complete" requirements document is a complete description of all the requirements the customer really cares about. Those items about which the customer has no opinion generally don't make it into the requirements document. As an example, the customer may require that when saving a file, a file chooser dialog should appear to allow the user to select where to save the file in the file system. But the customer may not care whether this dialog box includes functionality such as the ability to rename any files and folders that appear in the dialog as they are browsing for where to save a file.

4. **Realistic**

   All the requirements should be feasible and practical, given the resources available. The feasibility and practicality of the requirements may be determined on the fly during requirements elicitation, or determined later during a feasibility analysis. There may be several rounds of requirements elicitation followed by feasibility analysis before the design phase ever begins.

5. **Necessary**

   Frivolous requirements should be avoided. Care must be taken here to avoid conflict between what the customer requests and what the developer feels is necessary. To the developer, a requirement may seem superfluous, but if the customer is adamant about having it in the final product, it should probably be included.

6. **Verifiable**

   All requirements should be capable of being tested. Otherwise, there is no way to determine whether or not the software is valid (i.e., it meets all requirements). This is not as easy to accomplish as it sounds. A customer may use vague or ambiguous terminology that sounds perfectly valid, but is difficult to test. For example, suppose the customer states that the software should be easy to use. The value of that requirement is obvious, but how can the developer ensure the requirement has been met? Usability tends to be subjective, so software that some people find very usable might be considered unusable by others; which group of opinions should therefore be used to judge usability?

7. **Traceable**

   It should be possible to trace every requirement to a function, and vice versa. Traceability is often overlooked because its value tends to be underestimated. But the only way to determine whether all the requirements have been implemented, and whether there are any requirements that are unnecessary, is to show the traceability of each requirement througout the software life cycle. Each requirement should be traceable to one or more design components, which in turn should be traceable to one or more sections of code. A requirement that does not eventually get translated into code is either unnecessary, or indicates an incomplete software project. Ideally, each requirement, as well as each section of code, should be traceable to one or more tests that are executed during the testing phase.

   It is usually very difficult to achieve absolute success for all of these characteristics. The degrees of completeness and correctness, in particular, will depend on how well the customer knows what they want, and on how well the underlying problem is understood. Quite often one begins with a set of requirements thought to be complete, only to discover later on that something was left out that no one knew about when the requirements document was drafted. Time constraints and market constraints can also influence the requirements. Some requirements may be intentionally left out, or replaced by other requirements, in order to get something to market in time to capture a certain market share that might otherwise go to a competitor.

3.8 Functional vs. Non-functional Requirements

Sommerville does an excellent job explaining the difference between functional and non-functional requirements. I do want to point out, though, that the distinction between what is functional and what is non-functional can sometimes be vague. For example, consider the issue of timing constraints that Sommerville mentions. It may be that governmental regulations provide certain timing specifications for software that controls moving mechanical parts. Suppose, hypothetically, that the regulations state the software must be capable of responding within a time window of 0.5 seconds. That would be a non-functional requirement. It could be made into a functional requirement, however, if the customer explicitly requests that the software should be capable of responding within a time window of say, 0.25 seconds. In this case, the customer is striving for better performance than regulations require, so the governmental requirement is superseded. Another example is a situation where the software being developed must make use of a database, and the organization for which the software is being developed has a policy that all databases must use Oracle, for example. The requirement that the database be an Oracle database therefore becomes a non-functional requirement. In other situations, however, using an Oracle database could be a functional requirement — if the customer requests it without pressure of a mandate.

3.9 Requirements Specification

Once the requirements have all been elicited from the customer, they should be organized into a single, written document that will serve to formally specify the requirements for the designers to follow. This document is referred to as the **requirements specification**. If only one customer is involved, it may be possible to create the specification at the time the requirements are elicited. But if multiple customers are involved, or if the requested software is going to be complex, it is more likely there will be a round of requirements elicitation, followed by discussion and feasibility analysis among the developers, followed by a discussion with the customer to gain further clarification of the requirements or explain why certain requirements are not feasible. This process can be repeated as many times as needed (time permitting) until the developers feel they have as complete an understanding of the requirements as they are likely to get, at which point design can begin.

There is an inherent problem in the requirements elicitation process, namely that customers are more likely to speak in terms of natural language due to a lack of background in software engineering, while software engineers need much more detail in order to translate what the customer wants into a workable requirements specification. The only way to resolve this natural language problem is to work with the customer to flesh out details the customer probably doesn't even realize are necessary. Sommerville provides a good example of this in figures 4.9 and 4.10. The categorized information represented in figure 4.10 represent what is known as a **structured requirement**. All the requirements in a requirements specification should ideally be in structured format. Although it does admittedly take time to document requirements in such a detailed manner, these details are vital if ambiguity and confusion are to be avoided.

If you note the level of detail in the single requirement described in figure 4.10, you'll notice several categories of information that are usually only found in comments for source code. So what is this information doing here? It serves to define, as completely as possible, what each requirement entails. The more information that is provided, and the more formal the representation, the closer the requirements document comes to looking like source code. This raises an interesting question: might it be possible to generate a program directly from the requirements specification, automating the subsequent phases of development. Theoretically, this should be possible, and there is active research ongoing in this area, but for now, only relatively simple software can be built in this way.

Many formats have been proposed for constructing a requirements specification. Each format has its strengths and weaknesses, and will be accepted differently by different developers. The IEEE and ANSI, known for their work in standardization, have proposed their own, joint standard for the requirements specification document: IEEE/ANSI 830-1998. For the group project, I have indicated you should use this standard to document your requirements, and I have included a copy of the standard on the Canvas website for the course. If there is another standard your team would rather use, let me know and I will consider it, but I want everyone to follow some standard for creating the requirements document. There is a common misconception that software engineering means coding, since there is a habit among many developers to eschew formal documentation and go straight to writing code. For personal projects that may be fine, though it isn't good practice. The benefits of agile processes notwithstanding, for professional projects, especially those completed as a group, documentation is essential. There is no way to know for sure how many developers may end up being involved in a project during its life cycle. Those who are intimately familiar with all the details may leave to pursue other projects or jobs, to be replaced by developers who at best may have only a vague idea of what the software does, but must undertake the project's completion, or maintain it after it has been released. For those developers, a lack of documentation will be a severe handicap.

3.10 Volatile Requirements

If a customer really wants to get under the skin of a software engineer, all they have to do is wait until development has begun and then start changing the requirements. There is nothing as frustrating as trying to build a system when the requirements are volatile. Some changes are essential, if they are made to correct errors in understanding, correct design flaws, or account for necessary features that were omitted or overlooked during the requirements phase. Often, however, customers will insist on many changes simply because they don't really know what they want, or they keep changing their mind about how they want the software to work. It is usually a good idea to force the customer to sign off on a set of requirements, at which point the requirements become frozen — they cannot be changed unless they have to be in order to make the software work. In signing off on the requirements the customer also acknowledges that additional changes, if accepted, will incur additional cost.

What happens when changes are necessary, even when the requirements have been frozen? When changes must be made to the requirements, they must be made in a controlled, managed way, which is why it is vital to be able to trace every requirement throughout the development process. In a complicated system, there can be dependencies in the requirements, which can be affected if a change is made to one of the requirements involved. Suppose you're in the middle of the coding phase, and you realize a particular component cannot be implemented as requested. You must be able to trace the code back to the corresponding design(s) and the design(s) back to the corresponding requirement(s), in order to ensure you correct the entire problem instead of only part of the problem.