CS 425 Midterm Review Ver. 2

Amelia Qux

November 2004

1 Introduction

1.1 Professional software development

Generic products are products that are created for general consumer use. Think of everyday software: Microsoft Office 2003, e-mail (electronic mail), Adobe Photoshop.
Customized software is software written bespoke for an individual or organization. Think of software like the DMV system, United States electronic voting booths, and Scantron reading software. Customized software is usually encountered in the workplace or provided by federal and state institutions.

1.1.1 Good Software

1. Functionality: it does what the user needs
2. Performance: it meets a user's speed/resource expectations
3. Maintainability: it can be updated easily
4. Dependability and security: it's reliable and safe
5. Usability: it's easy to learn and operate

Overall, good software is flexible. It expects change and its ultimate goal is to serve the user.

1.1.2 Systematic approach to software

1. Specification: Customers and engineers define the software
2. Development: The software is designed and programmed
3. Validation: Software is checked against customer's requirements
4. Evolution: The software is modified as per changing customer and market requirements

1.1.3 Computer Science vs. System Engineering

Computer science concerns itself with the theoretical and methodical aspects under computers and computer systems
System engineering concerns itself with all aspects of software development and evolution of complex system

1.1.4 Issues

1. *Heterogeneity*: The more widely used software becomes, the more general purpose its architecture must be, i.e. it must be able to run on varied devices and be backwards compatible with old systems
2. *Business and social change*: The process for writing software must be fast enough to adapt to the current social and industrial climate
3. *Security and trust*: Risk of malicious activity, especially via commonly used software, should be as low as possible
4. *Scale*: Software ranges from small, local systems to large, cloud based systems

1.2 Software engineering ethics

Software engineers are expected to uphold *confidentiality*, *competence*, and *intellectual property rights* in their practices

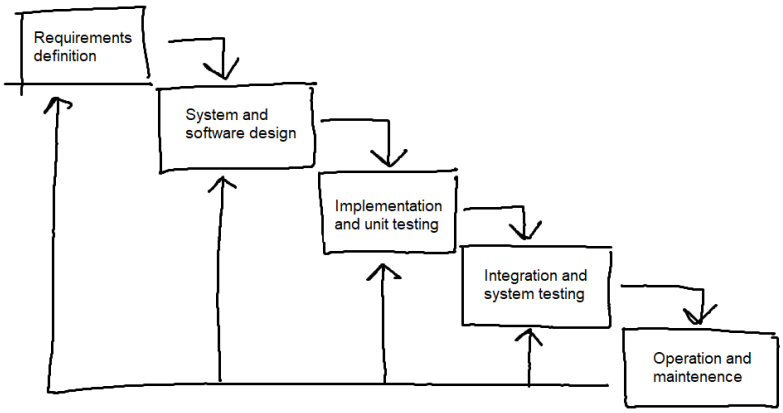
1. PUBLIC — Software engineers shall act consistently with the public interest
2. CLIENT AND EMPLOYER — Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest
3. PRODUCT — Software engineers shall ensure that their products and related modifications meet the highest professional standards possible
4. JUDGMENT — Software engineers shall maintain integrity and independence in their professional judgment
5. MANAGEMENT — Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance
6. PROFESSION — Software engineers shall advance the integrity and reputation of the profession consistent with the public interest
7. COLLEAGUES — Software engineers shall be fair to and supportive of their colleagues
8. SELF — Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession

2 Software Processes

2.1 Software process models

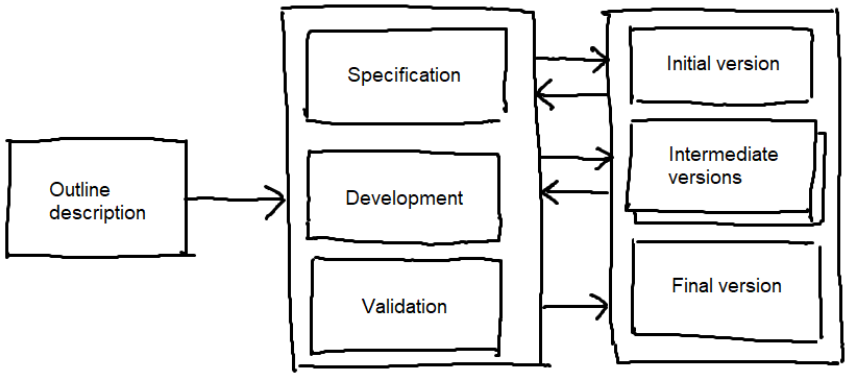
2.1.1 The Waterfall Model

The waterfall model has different phases that cascade to each other. The developer may only return to a previous phase after the last one. The waterfall model works for clear-cut software that won't experience many changes throughout development. The model still allows for the team to return to a previous pool and then cascade back down to Operation and Maintenance, but the team won't rely on these.



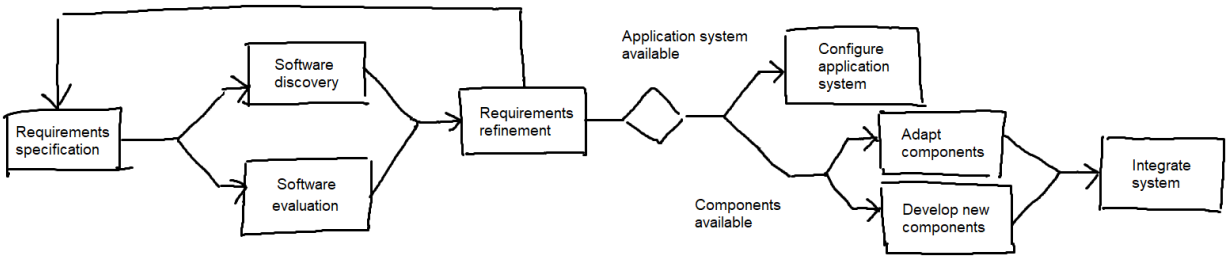
2.1.2 Incremental Development

Incremental development begins with the outline of the software, after which it moves onto a cycle of Specification, Development, and Validation. The software is released as a certain version and returned to the intermediate phase until its final version is released. This model works well for software that's new in concept but not completely refined in what it'll entail. Since customer input is still essential, releasing in incrementally updated versions is ideal in this case.



2.1.3 Integration and Configuration

Integration and configuration differs from the previous models as it relies on a generic, component driven development. This model of development is ideal when the requested software can easily build off of work done previously by others or if it's able to be built using many components of an established system.

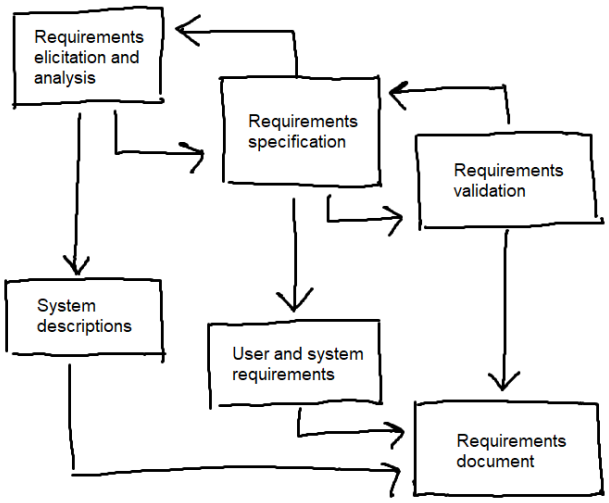


Note: Just because a process builds off of previous work isn't a bad thing. If someone has done it well before, why spend time re-making it? Modern software is built on the shoulders of giants.

2.2 Process activities

2.2.1 Main activities

1. Requirements elicitation and analysis: The derivation of a new system based on existing ones and interviews with potential clients
2. Requirements specification: The translation of the elicited requirements into a detailed requirements document
3. Requirements validation: The checking of the aforementioned requirements document for realism, consistency, and completeness followed by the making of corrections



2.2.2 Design activities

1. Architectural design: The identification of structure of the system and its principal components, their relationships, and their distribution
2. Database design: The design the system data structures and their relations in a database
3. Interface design: The definition of the interfaces between system components
4. Component selection and design: The search for reusable components and subsequent design of unavailable ones

System implementation consists of design/implementation, programming, and debugging followed by **verification and validation**

2.3 Coping with change

Change is inevitable in large software projects due to

1. Business changes
2. New technologies that improve implementations
3. Changing platforms

Prepare for change by using **system prototyping** as well as **incremental delivery**. These principles follow the **Fail Fast Principle**, which states that's it's better to fail as fast as possible so that course correction costs significantly less than if the failure occurred later on. By using prototypes, customers can let the design team know what they like and don't like about a system, far before any resources are spent on a feature that the customer potentially doesn't want. With incremental delivery, customers are eased into their product and can ask for changes along the way.

2.4 Process improvement

Improvement can be found through the

1. **Process Maturity Approach**, which focuses on improving process and project management and introducing good software engineering practice
2. **Agile Approach**, which is an iterative development approach, focusing on the reduction of software process overheads

2.4.1 Improvement activities

1. **Process measurement**: The measurement of one or more attributes of the product which form a baseline that helps decide if the process improvements have been effective
2. **Process analysis**: The current process is assessed to identified process weaknesses and bottlenecks. Process models (or process maps) that describe the process may be developed
3. **Process change**: The proposal of process changes to address identified process weaknesses

Process metrics include time taken, resources required, and number of occurrences.

3 Software Development Practices

3.1 Scrum

Scrum development is a system in which there are 2-4 week sprints during which a team tries to complete a certain set of tasks (stories). A backlog of new feature and bug stories, each with a story point value, are kept in the backlog and new ones are selected every new sprint along with rolled over stories from the last sprint.

3.1.1 Scrum Roles

1. **Developer** is responsible for the code
2. **Product Owner** is responsible for their priorities (checkout flow)
3. **Scrum Master** is responsible for re-tasking team members and facilitating stand-up meetings

3.1.2 Scrum Events

1. **Daily Scrum**s include stand-up updates and discussion of blockers
2. **Sprint Review** discusses the demo deadline
3. **Sprint Planning** manages re-tasking mid-sprint

3.1.3 Scrum Artifacts

1. The **Product Backlog** holds checkout priority change
2. The **Sprint Backlog** holds individual task assignments
3. The **Increment** is an addition of functionality within an existing feature

3.2 Agile

3.2.1 Manifesto

1. **Individuals and interactions** over processes and tools
2. **Working software** over comprehensive documentation
3. **Customer collaboration** over contract negotiation
4. **Responding to change** over following a plan

3.2.2 Principles

1. **Customer involvement** : Involve the customer; it's their product, their feedback is key
2. **Incremental delivery**: Frequently check in with your customer. It's easier to course correct now rather than later
3. **People not process**: Use your team to your advantage. Don't prescribe: everyone has their own method
4. **Embrace change**: Expect the system requirements to change and develop accounting for that
5. **Maintain simplicity**: Keep things simple. Complexity doesn't necessarily mean quality

3.3 Extreme Programming

New versions may be built several times per day, delivered to customers every 2 weeks, and all tests must pass for every build as quality assurance.

3.3.1 Practices

1. **Incremental planning**: Requirements are recorded on story cards included in a release, and are determined by the time available and their relative priority. The developers break these stories into tasks in their backlog
2. **Small releases**: The minimal useful set of valuable functionality is developed first. Frequent releases incrementally add functionality to the first release
3. **Simple design**: Enough design is carried out to meet the current requirements and nothing more
4. **Test-first development**: An automated unit test framework is used to test new functionality before it's released
5. **Refactoring**: All developers are expected to refactor the code continuously and as soon as possible, keeping the code simple and maintainable

3.3.2 Practices, Cont'

1. **Pair programming**: Developers work in pairs, checking each other's work and providing support
2. **Collective ownership**: The pairs of developers work on all areas of the system, maximizing the developers' expertise
3. **Continuous integration**: Work is integrated into the system as soon as a task is complete and all unit tests pass
4. **Sustainable pace**: Large amounts of overtime are not acceptable as it often leads to reduced code quality and medium term productivity
5. **On-site customer**: A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. They are responsible for their requirements

3.4 Writing Stories

Stories should be 'what they say on the tin'. Here are some stories:

SCRUM-145: Hotfix USB driver connection detailed in bug report `2004.3.8c1f9ae`

SCRUM-163: Rewrite `Recipe` class to utilize the Builder pattern

SCRUM-381: Write driver's license generation algorithm

SCRUM-501: Fix bugs in `ameliaq/werewolf_locator`

SCRUM-583: Fix bugs in database query menu

SCRUM-584: Fix bugs

It's clear that the quality of these stories degrade as the list goes on. The first is excellent: It describes what must be done, provides the sha of the bug report that it's associated with, and has its priority built into its description (hotfix). The second is another good story. It tells the responsible party what they must do and where. The third is still fine, but could be more specific. The quality of this story would depend on the size of the project and how organized the team was in other aspects. The fourth isn't great: stories should be specific and localized. 'Bugs' is very general, and while the story lists the branch the bugs are on, someone looking at it would have no idea what was going on. Even more so in the fifth story: there isn't even a pointer to where the bug is except for a vague reference to the menu. The last is the worst. 'Fix bugs' makes no sense as a story as it's the most general it could possibly be, could reference anything in any branch including 'main', and doesn't provide any context whatsoever.

Tip: If a story is easy to test, it's probably written well.

3.5 Application

Blockbuster wants software to manage their inventory in their stores. What may some of those stories be?

Samuel sat in the room he was waved to, tapping his pen on his desk. He had arrived at work that morning, only to be told he had to talk to a Blockbuster representative about some software that their company was going to write for them. He didn't mind making the trip, but he had some stories he was planning on finishing today. *Oh well, I'll just get them done-*. A woman walked through the door. She was a meter and a half tall, had brown curly hair that bounced with each step she took, and freckles that spattered her face. She wore a sweatshirt and jeans. He looked at her employee badge. *Tiffany Nguyen...* He had noticed that she hesitated for a moment when she first walked through the door. It was subtle, but he had noticed.
She extended her hand. "Hello, I'm Tiffany. I've got your specification requirements here." Samuel shook her hand, firmly, and the corner of her mouth curled into a smile. *Wow, he's got soft hands.* She didn't notice that she hadn't let go yet. Samuel coughed. Blushing, she pulled her hand away. "Sorry! I'm supposed to tell you what corporate wants." she rummaged through her bag and mumbled "I dunno why they don't just have us do this over the phone" eventually she pulled out a folder with dozens of sticky notes poking out.
"Yeah, I don't know either." Samuel was eyeing the colorful mosaic of sticky notes poking out of the folder. *She's so organized*. He couldn't help but smile. Tiffany smiled back.
"So, the software we have right now it's specialized enough. We want our employees to be able to scan a customer's card and then all the movies that they're renting. We also want to keep track of what movies we have in stock and what movies we've ordered to each location. Blockbuster locations should be able to make an order of movies to the headquarters, after which the system will mass purchase copies we don't have."
Samuel looked at the ceiling and tilted his head "Huh, I see, I see." He could see the stories already:

SCRUM-1: Establish a connection to the bar code scanner

SCRUM-2: Establish a connection to the Blockbuster database

SCRUM-3: Successfully interpret customer cards and movie bar codes

SCRUM-4: Successfully load customer information and movie rental information on successful scans

SCRUM-5: Flag movies as 'rented out' when a customer checks them out

SCRUM-6: Flag movies as 'missing' after a customer's account is terminated while they have movies rented out

SCRUM-7: Flag movies as 'missing' after a customer doesn't contact the store after thirty days after the return date

SCRUM-8: Add a feature that allows the system to apply fines to customer accounts

SCRUM-9: Automatically fine customers that have overdue movies once per day

SCRUM-10: Implement a feature that allows an employee to fine a customer for not rewinding a returned tape

SCRUM-11: Allow an employee to search a movie database for movies to compile an order

I'll need way more information. I can only think of really obvious features. Then there are all the stories for the bugs we'll encounter. Tiffany waved her hand in front of his face.
"Samuel, are you, uh, here? You kinda spaced out there."
Samuel snapped out of his daydream "Oh, yes, sorry. These are all very doable. We'll get them back to you in four weeks."
Tiffany smirked. "Four weeks? Okay, good luck."
Samuel raised an eyebrow. "And what makes you so qualified?" he didn't get easily irritated, but he was sweating a little. *What is with me right now?*
Tiffany continued listing requests from corporate. Samuel took notes, looking up occasionally. Every once in a while, their eyes would lock.

3.6 Test Driven Development

Test Driven Development is a form of development that follows a simple cycle:

1. Write code
2. Write tests that prove its functionality
3. Run those tests
4. Refactor the code until the tests pass
5. Clean up and document the code

Test driven development is extremely efficient. Writing tests may take time now, but they help catch bugs as soon as possible. As mentioned above, the **Fail Fast** philosophy is exemplified in test driven development. All unit tests may pass before a certain feature is added, but after that feature is added, half of them may begin to fail. This means that bugs are caught as soon as possible and don't hide in development builds. When writing unit tests, write as many as possible. Test edge cases, test bad input, test good input... test everything! The more tests are written, the more information they give about the state of the program. Not only does it provide assurance to the development team and the company responsible for the software, it shows customers that the software actually works.

3.6.1 Continuous Integration

Continuous integration works well with test driven development. If a company is developing into a Git repository, they may set up automatic tests that run on a pull request. Pull requests may only be completed if all tests pass. This allows changes to be continuously merged safely.

4 Requirements Engineering

A requirement is a function or a characteristic that must exist in a project. It can be high-level and abstract or extremely detailed.

4.1 Requirements Engineering Cycle

1. **Elicitation**: Ask the customer questions, observe how they work, and brainstorm features. Ultimately, this should outline the needs, constraints, and expectations of the customer
2. **Specification**: Turn those points into precise, unambiguous statements of what the system should do and under what conditions
3. **Validation**: Make sure requirements are complete, consistent, realistic, and testable, confirming with customers that you've caught their drift
4. **Change Management**: Track, negotiate, and update requirements as new needs or constraints emerge

4.2 Where To Find Requirements

1. **Interview** your customers to uncover needs and expectations
2. **Survey** your software's audience or have them fill out **questionnaires** to collect quick feedback from a large group to spot patterns
3. Hold **workshops** and/or **focus groups** to bring multiple stakeholders together to share and debate requirements
4. **Observe** your users in context to discover what they really do for **ethnographic** and behavioral insight
5. **Prototype** with rough models to spark feedback when requirements are unclear
6. Use **document analysis** to review existing records, forms, or policies to find hidden requirements

4.3 Functional and non-functional requirements

4.3.1 Overview

Functional requirements are high level statements that describe the system's services in detail. *What does the software do?*
Non-functional requirements outline timing and developmental constraints, standards of performance, security, usability. *How does the software achieve its functionality?*

4.3.2 Application

A wildlife government agency asks for software to manage their fishery on the Agave Coast. What could be some functional and non-functional requirements for the software?

1. FR: The system shall allow users to monitor the quality of the water in the coral reef
2. FR: The system shall allow users to track fish populations over time
3. FR: The system shall allow users to monitor incoming and outgoing boats on each port
4. NFR: The system shall return the pH of the water in less than a second of the query being made
5. NFR: The system shall return the salinity in g/kg in less than a second of the query being made
6. NFR: The system shall return the temperature of the water in degrees Celsius in less than a second of the query being made
7. NFR: The system shall display a graph of each fish population in thousands over the last year in less than a second of the query being made
8. NFR: The system shall automatically log boat activity (identification, action, date, and time) using SQL to the centralized server without requiring user input

Note how functional and non-functional requirements can mirror each other, referencing the same feature of the system

5 System Modeling

5.1 Context models

1. **Activity diagrams** model the functionality of a given piece of software
2. **Use case diagrams** show the interactions between a system and its environment
3. **Sequence diagrams** model the interactions between a system's actors and its objects
4. **Class diagrams** show the object classes and its associations to other classes
5. **State diagrams** show how the system reacts to internal and external events

5.2 Interaction models

1. Modeling user interaction helps to identify user requirements
2. Modeling system-to-system interaction highlights the communication problems
3. Modeling component interaction gauges if a proposed system structure is likely to deliver the required system performance and dependability
4. Use case diagrams and sequence diagrams model user interaction

5.3 Structural models

Structural models display the organization of a system in terms of the components that make it up and their relationships. They're created when discussing and designing the system architecture

5.4 Behavioral models

Behavioral models demonstrate dynamic behavior of a system in response to external input as it is executing.

5.5 Model-driven architecture

Model-driven engineering is an approach to software development in which a system is represented as a set of models that can be converted to executable code.