CS 430 Course Notes: Applications Software Engineering

Fall 2025 - Collin Roberts

Talha Yildirim,
tyildir [at] uwaterloo [dot] ca

Contents

1 The Scope of Software Engineering

1.1 The Classical and Object Oriented Paradigms

Definition 1.1.1 (Classical (Waterfall) Life Cycle Model)

1. **Requirements Phase**
   - Elicit Client Requirements
   - Understand client needs
2. **Analysis (specification) phase**
   - Analyze client requirements
   - Draft specification Documentation
   - Draft Software Project Management Plan
3. **Design phase**
   - Design architecture: Divide software functionality into components
   - Draft detailed design for each component
4. **Implementation phase**
   - Coding (development): Code and document each component
   - Unit test each individual component
   - Integration (system) testing: Combine components, test interfaces among components
   - Acceptance testing: Use live data in client's test environment. Clients participate in testing & verification of test results, and sign off when they are happy with the results.
   - Deploy to production environment
5. **Post delivery maintenance**
   - Maintain the software while it's being used to perform the tasks for which it was developed
6. **Retirement**
   - Product is removed from service: functionality provided by S/W is no longer useful / further maintenance is no longer economically feasible

♣

Problem. Why does the Waterfall life cycle model not have any of the following phases?

- Planning
- Testing
- Documentation

Solution.

- All three activities are crucial to project success
- Therefore all three activities must happen throughout the project and cannot be limited to just one project phase.

Remark

Difference between Classical and Object Oriented paradigms

Classical paradigm → One monolithic thing

Object Oriented Paradigm → Many smaller classes that work together

Definition 1.1.2 (Corrective maintenance)

Removal of residual faults while software functionality and specification remain relatively unchanged.

a.k.a fix production problems

Definition 1.1.3 (Perfective Maintenance)

1. Implement changes the client thinks will improve effectiveness of the software product. (e.g. Additional functionality, reduce response time)
2. Specifications must be changed

Definition 1.1.4 (Adaptive Maintenance)

1. Change the software to adapt to changes in environment (e.g. new policy, tax rate, regulatory requirements, changes in systems environment) - may not necessarily add to functionality. You allow software to survive.
2. Specifications may change to address the new environment

Important

The Importance of Post delivery Maintenance

- Shelf life of good software: 10, 20, even 30 years
- Good software is a model of the real world, and the real world keeps changing, therefore software must change too
- Cost of post delivery maintenance continues to go up, while cost of Implementation is nearly flat

Proposition 1.1.5 (Problems with the Classical Paradigms)

1. Works well for small systems ($\leq$ 5000 lines of code), but does not scale effectively to larger systems
2. Fails to address growing costs of post-delivery maintenance

Proposition 1.1.6 (Team Development Aspects)

1. Communication becomes challenging when teams are far apart geographically, especially when they are in different time zones
2. Interpersonal problems can undermine team effectiveness
3. If a call to a module written by another developer mentions the arguments in the wrong order

Proposition 1.1.7 (Ethical issues)

Software engineers commit to these ethical principles:
1. Public
2. Client and Employer
3. Product
4. Judgement
5. Management
6. Profession
7. Colleagues
8. Self

♠

2 Software Life Cycle Models

2.1 Iteration and Incrementation

Proposition 2.1.1 (Idealized Software Development)

$\emptyset \longrightarrow$ Requirements $\longrightarrow$ Analysis $\longrightarrow$ Design $\longrightarrow$ Implementation

♠

The *Classical model* is most effective when the IT team can work without accepting change to the requirements after the requirements are complete.

Changing requirements negatively affects software:
- Quality
- Delivery dates
- Budget

Definition 2.1.2 (Moving Target Problem)

The **moving target problem** occurs when the requirements change while the software is being developed

♣

Definition 2.1.3 (Scope Creep)

Scope creep or feature creep is a succession of small, almost trivial requests for additions to the requirements

♣

Definition 2.1.4 (Fault)

A **fault** is the observable result of a mistake made by any project staff member while working on any artifact

♣

Definition 2.1.5 (Regression Fault)

A **regression fault** occurs when a change in one part of the software product induces a fault in an apparently unrelated part of the software product

♣

Definition 2.1.6 (Regression Test)

A **regression test** provides evidence that we have not unintentionally changed something that we did not intend to change

♣

Definition 2.1.7 (Miller's Law)

Miller's Law states that, at any one time, a human is only capable of concentrating on approximately seven chunks of Integration

♣

Remark

Miller's Law Applied:

- Any large project will have much more then 7 components
- Hence we must start by working on ≤ 7 important components first temporarily ignoring the rest
- This technique is known as **stepwise refinement**

2.2 Life-Cycle Models

Other life-cycle models

1. Code and Fix Life-Cycle Model
2. Waterfall Life-Cycle Model
3. Rapid Prototyping Life-Cycle Model
4. Open Source Life-Cycle Model
5. Agile Processes
6. Synchronize and Stabilize Life-Cycle Model
7. Spiral Life-Cycle Model

Definition 2.2.1 (Code and Fix Life-Cycle Model)

Implement the software product without requirements, specification, or design

Strengths:

1. Thus technique may work on *very small* systems ($\leq$ 200 lines of code)
2. Easy to incorporate changes to requirements
3. Generates a lot of lines of code

Weaknesses:

1. This technique is totally unsuitable for systems of any reasonable size
2. This technique is unlikely to yield the optimal solution
3. Slow
4. Costly
5. Likelihood of regression faults is high

> **Remark**
>
> Only really works for user base of size 1 (e.g. assignments)

♣

Definition 2.2.2 (Waterfall (modified) Life-Cycle Model)

Augment the "vanilla" waterfall to include feedback loops during the project, and for post delivery maintenance. No phase is complete until all its documents are complete and approved by QA

Strengths:

1. Discipline enforced by QA

Weaknesses:

1. Specification documents may not be fully understood before they are approved
2. Finished product product may not actually meet the clients needs

♣

Definition 2.2.3 (Rapid Prototyping Life-Cycle Model)

A rapid prototype is a working model that is functionally equivalent to a subset of the software product

♣

Definition 2.2.4 (Open Source Life-Cycle Model)

1. Single individual has an idea for a program, builds initial version, makes it available free of charge to anyone who wants a copy
2. If there is sufficient interest then users become co-developers for post delivery maintenance
3. All participants can offer suggestions
4. Participation is voluntary and typically unpaid
5. Success depends on the interest generated by the initial version
6. In direct competition with enterprise software

♣

Definition 2.2.5 (Agile Process)

1. Communication
2. Speed
3. MVP is always kept in mind

Scrum Method

Requirements

- User Stories
- Prioritization
- Making Backlog

Sprints

- Daily Meetings
- Reassigning Tasks

Tries to ensure frequent delivery of new versions

Strengths:

1. Speed
2. Flexibility
3. Team Cohesion
4. Some history of success with smaller projects

Weaknesses:

1. Heavy on meetings
2. Not scalable with team size

Definition 2.2.6 (Synchronize and Stabilize Life-Cycle Model)

1. Pull requirements from the clients
2. Write specification documentation
3. Divide work into four builds
    1. Critical
    2. Major
    3. Minor
    4. Trivial
4. Carry out each build using small teams working parallel
5. **Synchronize** at the end of each day
6. **Stabilize** at the end of each build

Strengths:

1. Users needs are met
2. Components are successfully integrated
3. Tolerant of changes to specifications
4. Encourages individual developers to be innovative and creative
5. Daily synchronization and build stabilization ensures developers will all work in the same direction
6. Good for large projects
7. Holay glaze what the jelly

Weaknesses:

1. Only been used successfully at Microsoft

♣

Definition 2.2.7 (Spiral Life-Cycle Model)

- Minimize risk inherent in software development by the use of constant **proof of concept prototypes** and other means
- Proof of concept aims to determine whether an architecture design is good

We can make break this down into

1. Planning / Requirements
2. Risk Analysis
3. Develop and Verify
4. Plan Next Phase

Strengths:

1. Emphasis on alternative sand constraints supports reuse and software quality
2. This technique encourages doing the correct amount of testing

Weaknesses:

1. This model is only meant for internal building of large scale software
2. Project can look fine but really be heading for disaster
3. Makes assumptions that software is developed in discrete phases

♣

Comparison of Life-Cycle Models

Life-Cycle Model	Strengths	Weaknesses
Evolution Tree (§2.2)	-Closely models real-world software production -Equivalent to iteration and incrementation	
Iteration and Incrementation (§2.5)	-Closely models real-world software production -Underlies the Unified Process	
Code-and-fix (§2.9.1)	-Fine for short programs that require no maintenance	-Totally unsuitable for non-trivial programs
Waterfall (§2.9.2)	-Disciplined approach -Document driven	-Delivered product may not meet client's needs
Rapid Prototyping (§2.9.3)	-Ensures the delivered product meets the client's needs	-Not yet proven beyond all doubt
Open Source (§2.9.4)	-Has worked extremely well in a small number of instances	-Limited applicability -Usually does not work
Agile Processes (§2.9.5)	-Works well when the client's requirements are vague	-Appear to work on only small-scale projectes
Synchrionize-and-stabilize (§2.9.6)	-Future users' needs are met -Ensures that components can be successfully integrated	-Has not been widely used other than at Microsoft
Spiral (§2.9.7)	-Risk driven	-Can be used for only large-scale, in-house products -Developers have to be competent in risk analysis and risk resolution

3 The Software Process

3.1 Unified Process

Definition 3.1.1 (Software Process)

The **software process** encompasses the activities, techniques and tools used to produce software

♣

Definition 3.1.2 (The Unified Process)

1. We want to explore the unified process, which will be our software development methodology for the rest of the course
   - Object Oriented
2. The workflows
   - Task orientation
3. Phases have
   - Economic context
   - Time orientation

♣

Definition 3.1.3 (Artifact)

An artifact is a work product from a workflow

♣

Corollary 3.1.3.1 (Iteration and Incrementation within the Object Oriented Paradigm)

All variations on iteration and incrementation, including the unified process, attempt to preserve some classical structure, while being more tolerant of change than the classical model.

Under the unified process

1. Phases are the increments
2. we iterate through increments to complete the project

♡

Definition 3.1.4 (Unified Modelling Language)

UML stands for Unified Modelling Language

♣

Definition 3.1.5 (Model)

A model is a set of UML diagrams which describes one or more aspects of the software product to be developed

♣

> **Remark**
>
> **What is the motivation behind these innovations**
>
> 1. Even the best software engineers almost never get their artifacts right on the first attempt. So stepwise refinement will be needed
> 2. UML diagrams are visual, hence more intuitive than block verbiage
> 3. The visual nature of a UML model fosters collaborative refinement

3.2 Summary of Requirements, Analysis, Design, and Implementation Workflows

Requirements Workflow

Requirement artifacts can be

1. Incorrect (only client can detect this)
2. Ambiguous (e.g. AND versus OR in the inclusion criteria IT can detect this)
3. Incomplete (e.g. mission criterion to filter down to the program - only the client can detect this)
4. Contradictory (e.g. conflicting sort criteria - IT can detect this)

Using UML diagrams effectively helps to mitigate such problems with requirements

Analysis Workflow

1. Analyze and refine the requirements to achieve the level of detail needed to begun designing the software and maintain it effectively later
2. Once the analysis is complete, the cost and duration of the development are estimated → create the Software Project Management Plan (SPMP)
3. Terminology: Deliverables, Milestones, and Budget

Design Workflow

1. Show **how** the product is to do what it must do
2. Refine the artifacts of the analysis workflow until the result is good enough for the developers to implement it
3. There are differences between the classical and the object oriented paradigms here
4. It is important to keep detailed records about design decisions

Implementation Workflow

1. Implement the target software product in the chosen implementation language
2. Usually code artifacts are implemented by different developers and integrated once implemented - thus design shortcomings may not come to light until the time of integration

Test Workflow

1. Ensure the correctness of the artifacts from all other workflows

Requirements

1. Traceability: every later artifact must trace back to a requirement artifact
2. Until implementation there will be no code to test, only documents. Hence we test by holding a review of the document, with the key stakeholders.

Analysis

1. Tactic: Hold a review of analysis artifacts with key stakeholders chaired by QA
2. Review the SPMP too

Design

1. Design artifacts must trace back to analysis artifacts
2. Tactic: Hold a review of design artifacts

Implementation

1. Some projects also incorporate **alpha** and **beta** testing
2. Although it is tempting **alpha** testing should not replace thorough testing by the QA group

Post Delivery Maintenance

1. This is **not** an afterthought - it must be planned from the start
2. Pitfall: Lack of adequate documentation
3. Testing changes must include **positive** and **regression** testing

Definition 3.2.1 (Positive Testing)

Positive testing means testing that what you intended to change was changed in the desired way

♣

Retirement

1. This is triggered when post delivery maintenance is no longer feasible or cost effective
2. Usually a software product is replaced at this point
3. True retirements are rare