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 $\longrightarrow$ One monolithic thing

Object Oriented Paradigm $\longrightarrow$ 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)

$\varnothing \longrightarrow$ 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

3.3 Unified Process Part II

The Interaction Between Phases and Workflows

1. Phases have a **time orientation**
2. workflows have a **task orientation**

Why separate the 2?

Moving target problem with iteration and incrementation leads to the splitting of tasks and time, which in turn leads us to the unified process.

> **Remark**
>
> **Goal:** Determine whether it is worthwhile to develop the target software product. Is it economically viable to build it?

Requirements Workflow - Key steps

1. Understand what is

Definition 3.3.1 (Domain)

The domain of a software product is the place

e.g. TV station, hospital air traffic

2. Build

Definition 3.3.2 (Business Model)

A business model is a description of the clients business process

3. Determine the project scope

4. The developers make the initial

Definition 3.3.3 (Business Case)

A **business case** is a document which answers these questions

1. Is the proposed software cost effective? Will the benefits outweigh the costs? In what time frame? What are the costs of not developing the software?
2. Can the proposed software be delivered on time? What impacts will be realized is the software is delivered late?
3. What risks are involved in developing the software, and how can these risks be mitigated?

♣

Corollary 3.3.3.1 (Major Risk Categories)

1. Technical risks
2. Bad Requirements
3. Bad Architecture

♡

Analysis Workflow

Goal: Extract the information need to design the architecture

Design Workflow

1. Create the design
2. Answer all questions required to start implementation

Implementation Workflow

1. Usually little to no coding happens during the inception phase
2. Sometimes it will be necessary to build a proof-of-concept prototype

Test Workflow

Goal: Ensure that the requirements artifacts are correct

Deliverables from the Inception Phase

1. Initial version of the domain model
2. Initial version of the business model
3. Initial version of the requirements artifacts
4. Initial version of the analysis artifacts
5. Initial version of the architecture
6. Initial list of risks
7. Initial use cases

8. Plan for Elaboration phases
9. Initial version of the business case
   - Software is to be marketed this includes revenue projections, market estimates, initial cost estimates, etc
   - If software is to be used in-house, this includes the initial cost/benefit analysis

Elaboration Phase

Goals:

1. Refine the initial requirements
2. Refine the architecture
3. Monitor risk and refine their priorities
4. Refine the business case
5. Produce the SPMP

Deliverables:

1. The completed domain model
2. The completed business model
3. The completed requirements artifacts
4. The completed analysis artifacts
5. Updated the version of the architecture
6. Updated the list of risks
7. SPMP
8. The completed business case

Construction Phase

Goal:

Produce the first operational-quality version of the software product (the **beta** release)

Deliverables:

1. Initial user manual and other manuals
2. Completed version of the architecture
3. Updated list of risks
4. SPMP updated
5. If needed, the revised business case

Transition Phase

Goal:

Ensure the clients requirements have been met

Deliverables:

1. Final versions of all the artifacts
2. Final versions of all manuals / other documentation

3.4 One versus Two Dimensional Life-Cycle Models

Question:

Can one's "position" in the Life-Cycle be described along only one axis, or does it need two axes

Example:

1-D	2-D
Classical (Waterfall) (Figure 3.2a) Code-And-Fix Open Source? Possibly Others...	Unified Process (Figure 3.2b) Iteration and Incrementation Spiral Possibly Others...

Remark

1. Two dimensional models are more complicated, but we cant avoid them, especially the Unified Process
2. The Unified Process is the best model we have so far, but it will be surpassed in the future

Definition 3.4.1 (Capability Maturity Models)

CMMs are related group of strategies for improving the software process, irrespective of the choice of Life-Cycle model used.

1. Software-CMM (our focus)
2. People-CMM
3. Systems Engineering
4. Integrated Product Development-CMM
5. Software Acquisition-CMM

These form CMM Integration (CMMI)

Definition 3.4.2 (Software Capability Maturity Models)

Use of new software techniques alone will not result in increased productivity and profitability, because our problems stem from how we manage the software process. Improving our management of the software process should drive improvements in productivity and profitability

3.5 Levels of CMM

An organization advances incrementally through five levels of maturity

1. **Initial**
   1. No sound software engineering practices are in place
   2. Most projects are late and over budget
   3. Most activities are responses to crises, rather than preplanned tasks

2. **Repeatable**
    1. Basic Software Project Management practices are in place
    2. Some measurements are taken
    3. Managers identify problems as they arise and take immediate corrective action to prevent them from becoming crises
3. **Defined**
    1. The process for software production is fully documented
    2. There is continual process improvement
    3. Reviews are used to achieve software quality goals
    4. CASE environments increase quality / productivity further

Definition 3.5.1 (Computer Aided/Assisted Software Engineering (CASE))

CASE stands for Computer Aided / Assisted Software Engineering ♣

4. **Managed**
    1. The organization sets quality productivity goals for each software project
    2. Both are measured continually and corrective action is taken when there are unacceptable deviations
    3. Typical measure: faults / 1000 lines of code, in some time interval
5. **Optimizing**
    1. The goal is continual process improvement
    2. Statically quality / process control techniques are used to guide the organization
    3. Positive Feedback Loop: Knowledge gained from each project is used in future projects. Therefore productivity and quality steadily improve

`linebreak`

4 Teams

4.1 Team Organization

To develop a software product of any significant size a team is required

Suppose that a software product requires 12 person months to build it. Does it follow that 4 programmers could complete it in 3 months?

No!!

1. There are new issues once a team is involved
2. Not all programming tasks can be fully shared in time or sequencing

Definition 4.1.1 (Brook's Law)

Brooks Law states that adding programmers to an already late project makes it later ♣

Definition 4.1.2 (Classical Chief Programmer Team)

A classical Chief Programmer Team is a team organized according to some variation of the above picture, possibly with fewer or more programmers, and having the following roles

Chief Programmer:
- Highly skilled programmer
- Successful manager
- Does architectural design
- Writes critical / complex sections of the code
- Handles all interface issues
- Reviews the work of all team member
- Handles all interface issues
- Reviews the work of all team members

Backup Programmer
- Needed in case chief programmer wins the lottery, gets sick, falls under a bus, etc
- As competent as the chief programmer in all respects
- Does tasks independent of design process

Programmer Secretary (Librarian)
- Maintain the production library, including all project documentation

Programmer
- They just program

♣

Remark

What are the strengths and weaknesses of Classical Chief Programmer Teams?

Strengths:
- Lots of success in a couple cases

Weaknesses:
- Chief/backup programmers are hard to find
- Secretaries are also hard to find
- The programmers may be frustrated at being “second class citizens” under thus model

Definition 4.1.3 (Egoless Programming)

1. Code belongs to the team as a whole, not any individual
2. Finding faults is encouraged
3. Reviewers show appreciation at being asked for advice, rather than ridiculing programmers for making mistakes

♣

Definition 4.1.4 (Democratic Team)

A team of ≤ 10 egoless programmers constitutes a democratic team

Possible Managerial Issues:
1. For such collaboration to flourish, there must be a strong culture of open communication
2. The path for career advancement may not be clear

Strengths:
1. Rapid detection of faults $\rightarrow$ higher quality code
2. Addresses the problem of programmers being overly attached to their own code

Weaknesses:
1. Managerial issues
2. It is hard to create such a team
3. A certain organizational culture is required for such a team

4.2 Chief Programmer and Democratic Teams

Chief and Democratic are opposite ends of the spectrum

Classical Chief Programmer:
- Very hierarchical
- Little individual freedom

Democratic:
- Little to no hierarchy
- Much individual freedom

A Conflict Inherent in the Chief Programmer Model

- The Chief Programmer must attend all code reviews. They are responsible for every line of code as, as the Technical Manager of the team
- **Resolution:** Split the Chief programmer role into a Team Manager (non-technical) and Team Leader (technical)

4.3 Open Source

Why might people not want to participate in open source projects?

- Unpaid
- Philosophical disagreements about direction
- You don't own the code and can't monetize it
- You give up control over the finished product and give away your effort

Why might you contribute to an open source project?

- Fix a bug your facing
- You believe in the product / it has value to you

- You admire the lead maintainer
- You feel better "making the world a better place"
- Learning experience for your next job

4.4 Morals

1. For success, high quality programmers are required. They can succeed even in an environment as unstructured as open source one typically is
2. Provided #1 holds, the way that a successful open source project team is organized is essentially irrelevant to the success or failure of the project

Here is Figure 4.7 from the text:

Team Organization	Strengths	Weaknesses
Classical Chief Programmer Teams (§4.3)	-Major success of NYT project	-Impractical
Democratic Teams (§4.2)	-High quality code as a consequence of positive attitude towards finding faults -Particularly good with hard problems	-Experienced staff resent their code being appraised by beginners -Cannot be externally imposed
Modified Chief Programmer Teams (§4.3.1)	-Many successes	-No success comparable to the NYT project
Modern hierarchical programming teams (§4.4)	-Team manager / Team leader obviates need for chief programmer -Scales up -Supports decentralization when needed	-Problems can arise unless team manager / leader responsibilities are clearly delineated

Here is Figure 4.7 from the text:

Team Organization	Strengths	Weaknesses
Synchronize and Stabilize Teams (§4.5)	-Encourages creativity -Ensures that a huge number of developers can work towards a common goal	-No evidence so far that this method can be used outside Microsoft
Agile Process Teams (§4.6)	-Programmers do not test their own code -Knowledge is not lost if one programmer leaves -Less experienced programmers can learn from others -Group ownership of code	-Still too little evidence regarding efficacy
Open Source Teams (§4.7)	-A few projects are extremely successful	-Narrowly applicable -Must be led by a superb motivator -Required top-calibre participants

- There is no choice of team organization that is optimal in all situations

5 Tools of the Trade

1. Stepwise Refinement
2. Cost Benefit Analysis
3. Divide and Conquer
4. Separation of Concern
5. Software Metrics
6. Taxonomy of CASE
7. Scope of CASE
8. Software Versions
   1. Revisions
   2. Variations
   3. Moral
9. Configuration Control
   1. Configuration Control During post-delivery maintenance
   2. Baselines
   3. Configuration Control During Development
10. Build Tools
11. Productivity Gains with CASE Technology

5.1 Stepwise Refinement

Definition 5.1.1 (Stepwise Refinement)

Stepwise Refinement is a technique by which we defer nonessential decisions until later, while focusing on the essential decisions first ♣

- This is a response to Millers Law
- **Key Challenge:** Decide which issues must be handled in the current refinement, and which can be deferred until a later refinement.
- Brainstorming can be used in early stages
- Features of brainstorming
  - The problem to be solved initially be unclear
  - All team members are encouraged to speak, especially the shy ones
  - No editing in the first round, when ideas are being suggested

5.2 Cost Benefit Analysis

Definition 5.2.1 (Cost-Benefit Analysis)

Cost Benefit Analysis is

- A technique for determining whether a possible course of action would be profitable, in which we
  - Compare estimated future benefits against estimated future costs,
  - Often referred to as the "balance sheet view"
  - When selecting among several options the optimal choice maximizes the difference

$$(\text{estimated benefits}) - (\text{estimated costs})$$

- **Pitfalls**
  - We must quantify everything to start
  - Some things are easier to quantify than others
- **Tangible Benefits**
  - Tangible benefits are easy to measure, e.g. estimated revenue from launching a new product
- **Intangible Benefits**
  - Intangible benefits can be more challenging to quantify, e.g. the reputation of your organization

5.3 Divide and Conquer

Definition 5.3.1 (Divide and conquer)

Is it break a large problem down into sub-problems, each of which is easier to to solve.

Definition 5.3.2 (Analysis Package)

An analysis package is defined by: During the analysis workflow:

1. Partition the software product into analysis packages
2. Each package consists of a set of related classes that can be implemented as a single unit

- During the design workflow:
  - Partition the implementation workflow into corresponding manageable pieces, termed subsystems
- During the implementation workflow:
  - Implement each subsystem in the chosen programming language
- **Divide and Conquer Key Problem:** There is no algorithm for deciding how to partition a software product into smaller pieces. Experience and human intuition are required.

5.4 Separation of Concerns

Definition 5.4.1 (Separation of Concerns)

A software product has separation of concerns if it is broken in components that overlap as little as possible with respect to their functionalities

- Separation of concerns is the "new and improved" divide and conquer
  - Guiding Principal: Reduce / eliminate overlap in functionalities between the subsystems created by dividing them
- **Motivation:**
  - Minimize the number of regression faults. If separation of concerns is truly achieved then changing one module cannot affect another module
  - When done correctly this also facilitates reuse of modules in future software products
- **Manifestation of Separation of Concerns:**
  - Design technique of high cohesion
  - Design technique of loose coupling
  - Encapsulation
  - Information hiding
  - Three tier architecture

Remark

Separation of concerns is desirable for Software Engineering

5.5 Software Metric

Definition 5.5.1 (Metric)

A metric is anything that we measure quantitatively

- We need metrics to detect problems early in the software process before they become crises
- Two types:
  - Product metrics: e.g. # of lines of code
  - Process metrics: e.g. turn over rate
- Five essential fundamental metrics for software project:
  1. Size (e.g. in # lines of code)
  2. Cost to develop / maintain (in dollars)
  3. Duration to develop (in months)
  4. Effort to develop (in person-months; or as in my experience in person days)
  5. Quality (number of faults detected during the project)
- Augment this list as appropriate for your project
- There is no universal agreement among software engineers which metrics are right

5.6 Taxonomy of CASE

- CASE stands for Computer Aided/Assisted Software Engineering
- At present a computer is a tool of and not a replacement for a software professional
- CASE tools used tools during the
  - Earlier workflows are called front-end or upperCASE tools, and
  - Later workflows (implementation postdelivery maintenance) are called backend or lowerCASE
- Examples:
  - Data dictionary
  - Consistency checker
  - Report Generator
  - Screen Generator
- Combining multiple tools creates a workbench
- Combining multiple workbenches creates an environment
- So our taxonomy is: tools (task level) → workbenches (team level) → environments (organization level)

5.7 Scope of CASE

Primary motivations for implementing CASE:

1. Produce high quality code
2. Have up to date documentation at all times
3. Automation makes maintenance easier
4. Do everything more quickly, hence more cheaply

Coding tools of Case:

- Text editor
- Debuggers
- Formatters
- **Programming in the small:** Coding a single module
- **Programming in the large:** Coding at the system level
- **Programming in the many:** Software production by a team

Definition 5.7.1 (Revision)

A revision is created when a change is made, e.g. to fix a fault

- Old revisions must be retained for reference
  1. If a fault is found at a site still running the old revision
  2. For auditing

Definition 5.7.2 (Variation)

A variation is a slightly changed version that fulfills the same role in a slightly changed situation

Remark

A CASE tool is needed to effectively manage multiple revisions of multiple variations

Definition 5.7.3 (Configuration)

A configuration of a software product is a list, for every code artifact, of which version is included in the software product

Definition 5.7.4 (Configuration Control Tool)

A configuration control tool is a CASE tool for managing configurations

- **configuration Control:**
  - **Motivation:** Fix software faults effectively
  - The first step towards fixing a problem is to recreate it in a development environment
  - If many configurations are possible, then configuration control will be needed in order to recreate problem in a development environment
  - Version control also facilitates ensuring that the correct versions get included when compiling / linking
  - A common technique is to embed the version as part of the name
  - Adding details to a configuration yields a derivation of a software product

Definition 5.7.5 (Derivation)

A derivation is a detailed record of a version of the software product including

- The variation / revision of each code element (i.e. Configuration)
- The versions of the compilers/linkers used to assemble the product
- the date / time of assembly
- The name of the programmer who created the version

Remark

A version control tool is required to effectively track derivations

Configuration Control During Post-Delivery Maintenance

- There is an obvious problem when a team maintains a software product
- Suppose that two programmers receive two different fault reports. Suppose further that fixing both faults requires changes to the same code artifact
- Without any new process in place, the programmer #2 will undo programmer #1 changes at deployment time

Baselines

- When multiple programmers are working on fixing faults a baseline is needed
- A baseline is a set of versions of all the code artifacts in a project
- A programmer starts by copying the baseline files into a private workspace. The he can freely change anything without affecting anything else
- After the fault is fixed, the new code artifact is promoted to production, modifying the baseline
- The old, frozen version is kept for future reference, and can never be changed
- This technique extends to multiple programmers and multiple code artifacts

Configuration Control During Development

- During Development and Desk Checking, changes are too frequent for configuration control to be useful
- We definitely want configuration control to be in force by the time deploy to production
- The text author recommends that configuration control should apply once the code artifact is passed off to the QA team
- The same configuration control procedures as those for postdelivery maintenance should then apply
- Proper version control permits management to take corrective action if project deadlines start to slip

Definition 5.7.6 (Build Tool)

A build tool selects the correct compiled code artifact to be linked into a specific version of the software product

Build Tools

- Some organizations may not want to purchase a complete configuration control solution

- **Issue**
  - While version control tools assists programmers in deciding which version of the source code is the latest, compiled code not not automatically get a version attached to it

Productivity Gains with CASE Technology

- Research to date shows a modest gain in productivity following the introduction of CASE tools to an organization
- Other benefits of using CASE tools:
  - Faster development
  - Fewer faults
  - Better usability
  - Easier maintenance
  - Improved morale on the IT team

This list of CASE tools is summarized in Figure 5.14 in the text.

Build tool (§5.11)	Coding tool (§5.8)
Configuration-control tool (§5.10)	Consistency checker (§5.7)
Data dictionary (§5.7)	E-mail (§5.8)
Interface checker (§5.8)	Online documentation (§5.8)
Operating system front end (§5.8)	Pretty printer (§5.8)
Report generator (§5.7)	Screen generator (§5.7)
Source-level debugger (§5.8)	Spreadsheet (§5.8)
Structure editor (§5.8)	Version-control tool (§5.9)
Word-processor (§5.8)	World Wide Web browser (§5.8)

6 Testing - Non-Execution Based Testing

- Quality Issues
  - Software Quality Assurance
  - Managerial Independence
- Non-Execution Based Testing
  - Reviews
  - Walkthroughs
  - Managing Walkthroughs
  - Inspections
  - Comparison of Walkthroughs and Inspections
  - Strengths and Weaknesses of Reviews
  - Metrics for Inspections

6.1 Quality Issues

Definition 6.1.1 (Failure)

A failure is an observed incorrect behaviour of the software product cased by a fault

Definition 6.1.2 (Error)

Error is the amount by which the software product output is incorrect

Definition 6.1.3 (Defect)

A defect is a generic term for a fault failure or error

Definition 6.1.4 (Quality)

Quality describes the extent to which the software product satisfies its specification

6.2 Software Quality Assurance (QA)

- Quality alone is not enough: software must be easily maintained.
- QA must be built in throughout the project, not simply imposed by the QA group at the end of a workflow
- Primary duty of a QA group:
  - The quality of the software process
  - The quality of the software product
- Once a workflow is complete, QA verify that all artifacts are correct

6.3 Managerial Independence

- Development and QA teams should be led by independent managers, neither of whom can overrule the other
- Reason: Often faults are found towards end of deadlines. We must decide between
  - Delivering the software on time with faults
  - Fixing the faults and delivering late
- Both most report to a third manager who must then make the decision about what to do on a case by case basis

6.4 Review

Definition 6.4.1 (Review)

A review is a walkthrough or an inspection

♣

- Common Features of All Reviews
  - Non-execution based testing
  - Centred around a meeting of key stakeholders
  - Chaired by QA representative
  - The meeting is to test a document to identify, but not attempt to fix, faults in that document
    - Committees solution is usually of lower than that of trained expert
    - Committees solution takes 4-6 times as much effort as individual
    - Not all faults identified during review are truly faults
    - Takes too much time

6.5 Walkthroughs

- The two steps for a walkthrough
  1. Preparation
  2. Team analysis of the document
- 4-6 participants
  1. QA (chair)
  2. Manager responsible for requirements
  3. Manager responsible for analysis
  4. Manager responsible for design
  5. Client representative
- Two fundamental approaches to conducting a walkthrough
  1. Participant driven
  2. Document driven (more detailed, but time consuming)

6.6 Inspections

- The five steps for an inspection:

1. **Overview** document author gives the overview
2. **Preparation** participants examine the document
3. **Inspection** quick document walkthrough
4. **Rework** document author corrects all faults noted in the written report
5. **Follow Up** moderator ensures that every fault identified has been fixed and that no new faults were introduced in the process

- Roles for an Inspection
  1. Moderator (QA)
  2. Analyst (i.e. stakeholder, previous workflow)
  3. Designer (i.e. Document author; stakeholder, current workflow)
  4. Implementer (i.e. stakeholder, next workflow)
  5. Tester (QA, a different person than the moderator)

6.7 Comparison of Walkthroughs and Inspections

Although inspections are more costly there is evidence that they are most effective at finding faults

6.8 Strengths and Weaknesses of Reviews

- **Strengths**
  - Effective at detecting faults, especially early in the life cycle when they are cheaper to fix
- **Weaknesses**
  - A large software products artifacts are hard to review unless they consists of smaller independent components
  - Effectiveness of review team is hampered if not all documentation from the previous workflow is completed yet

6.9 Metrics for Inspections

1. **Inspection Rate:**
   - Requirements / design: # of pages / hour
   - Code: # of lines of code / hour
2. **Fault Density:**
   - Requirements / design: # of faults / page
   - code: # of faults / 1000 lines of code
3. **Fault detection rate:**
   - # of faults detected / hour
4. **Fault detection efficiency:**
   - # of faults detected / person hour

Remark

- Metrics attempt to measure our effectiveness at finding faults
- Spike in faults might mean decrease in software quality, not increase in QA effectiveness