CS 430
Lecture Notes
Fall 2023

Collin Roberts

December 14, 2023

Contents

1 Lecture 01 - Introduction to Software Engineering

Outline

1. Introduction to CS 430 - Course Outline
2. Introduction to the Scope of Software Engineering
3. Historical Aspects
4. Economic Aspects

1.1 Introduction to CS 430 - Course Outline

1. Look up the Course Outline on the unsecured course website.
2. This component of the evaluation for the course is still fairly new this term (used once before):
   (a) Case Studies on Kritik
      i. The Case Studies will be marked via a peer evaluation tool called Kritik. The use of Kritik is motivated by my desire to develop your critical thinking skills more than the usual approach of having you hand in assignments which are marked solely by Teaching Assistants does. Since this will be my second offering using Kritik, I am continuing with a blended approach between Kritik and traditional assignments.
      ii. Kritik is not yet supported across the University of Waterloo, so there will be a small subscription fee of $24 (I think, and will confirm soon) for you to use Kritik this term. If the subscription charge is prohibitive for you, then please reach out to me.

iii. More details about Kritik will be posted on the unsecured course website in advance of the release of Case Studies #1.

3. I will announce suggested pre-reading from the text for all the future lectures, by email.
4. **Clickers**
   (a) iClicker Remote
      i. Register your iClicker using the instructions on LEARN.
      ii. If your Participation grade still shows as 0 on LEARN after a few lectures, then please contact the instructor to correct the registration of your iClicker.
   (b) Paper Submit one piece of paper to me at the end of each lecture, indicating
      i. student number / login ID / both, and
      ii. one line per CQ, numbered my way, indicating which of options A–E you chose.

1.2 Introduction to the Scope of Software Engineering

1. **Software Engineering:** the idea of applying Engineering Principles to the building of **big** software products.

1.3 Historical Aspects

Examples:

1. On November 9, 1979, U.S. Strategic Air Command had an **alert scramble** when the worldwide military command and control system (WWMCCS) computer network reported that the Soviet Union had launched missiles aimed at the U.S.A. A simulated attack was misinterpreted as the real thing. See the text for full details.
   Moral: Software faults can have disastrous real world consequences!
2. A Standish study of 9236 software projects completed in 2006 revealed (see text Fig 1.1) that
   (a) only 35% were delivered successfully,
   (b) 19% were cancelled and
   (c) 46% were late, over budget, and/or had features missing.

Remarks:

1. Bridges occasionally collapse, and power generators occasionally fail, but not nearly as often as operating systems crash or billing systems

produce bills for incorrect amounts.
2. Even when software is delivered fault-free, it is often late, over budget, and/or fails to meet all the user's requirements.
3. This motivates the following key definition.

Definition 1.3.1. *The **software crisis** (or **software depression**) is the phenomenon whereby, all too often, software is delivered:*

1. *with faults,*
2. *late,*
3. *over budget, and/or*
4. *not meeting all the user's requirements.*

Remarks:

1. Applying the same principles that traditional engineers use can help improve the delivery of software.

Definition 1.3.2. ***Software engineering** is a discipline whose aim is the production of software that*

1. *is fault-free,*
2. *is delivered on time,*
3. *is delivered within budget, and*
4. *satisfies the client's needs.*

Furthermore the software must be easy to modify when the user's needs change.

Remarks:

1. The name software engineering indicates that software developers will have better success if they use the same principles as traditional engineers.
2. Software engineering is new field with a broad scope - math, CS, science, engineering, management, etc.
3. Software Engineering is a response to the Software Crisis.

1.4 Economic Aspects

Question: If a new coding method becomes available which is 10% faster than the current method, then should we adopt it immediately?

Pros	Cons
-short term: lower development costs -longer term: compound short term savings -possible improved security features	-higher maintenance costs with a blended system -possible need to rewrite existing code -possible compatibility problems -new environment unproven, possibly unstable -no benefit with respect to maintenance costs -might affect user experience in unexpected ways -training / learning curve -new code could be less robust -might require hardware changes -cost of purchasing the new development studio is not stated -possible issues with stability, performance -new code might be of lower quality

Moral: This is not a clear yes/no answer. More analysis is still required.
Remarks:

1. The "Pro"s touch the development phase; the "Con"s reveal impacts in other phases.
2. It turns out that historically, maintenance costs have grown faster than development costs. **Moral:** reducing maintenance costs is a bigger win.

This is Coding example. Coding = 10-15% software development effort. Similar principles apply to all aspects of software development

2 Lecture 02 - The Classical and Object-Oriented Paradigms

Outline

1. Example: Classical (Waterfall) Life-Cycle Model
2. Example: Object-Oriented Paradigm
3. Maintenance Aspects

(a) The Importance of Postdelivery Maintenance
4. Requirements, Analysis and Design Aspects
5. Team Development Aspects
6. The Object-Oriented Paradigm
7. The Object-Oriented Paradigm In Perspective
8. Ethical Issues

2.1 Example: Classical (Waterfall) Life-Cycle Model

Refer to Fig 1.2 in the text for the phases of the Classical (Waterfall) life-cycle model:

1. Requirements phase
2. Analysis (specification) phase
3. Design phase
4. Implementation phase
5. Postdelivery maintenance
6. Retirement

Refer to the Examples document on LEARN (Lecture 02)

Answers to the Questions for Discussion

1. How would each group below react to the introduction of the Waterfall life-cycle model?
   (a) IT Team
      i. initial resistance to change
      ii. We do not like the extra work required before development can begin (requirements, analysis, design)
      iii. Additional structure makes our work more efficient and effective
      iv. We like the added quality that is now possible
      v. We like the fact that we can take vacations now!
      vi. We can work effectively in teams now (this was difficult to impossible before)
      vii. Since analysis/design will now be done within IT, it should become better
      viii. Additional planning should pay off later
         A. better quality software; fewer firefights
   (b) Business Partners
      i. too slow: after freezing the requirements, no changes can be made until the next project starts - also there is a long wait

time between our two main points of influence: requirements and user acceptance testing
   ii. BUT - additional structure makes it much more likely that we get our work right the first time (unlike in the past were a lot of rework was required)
   iii. It is more likely we will get our requirements correct now
   iv. integration projects are now possible (they weren't before)
   v. We can plan better for the future, using past history
   vi. additional structure should make the software we produce be of higher quality
   vii. We are less vulnerable to knowledge loss if developers leave the organization

2. Why does the Waterfall life-cycle model not have any of the following phases?
   (a) Planning
   (b) Testing
   (c) Documentation

   Answer:
   (a) All three activities are crucial to project success.
   (b) Therefore all three activities must happen throughout the project and cannot be limited to just one project phase.

Other Observations

Pros	**Cons**
-more structure	-less freedom
-people can specialize their roles to the phases	-need to finish requirements before analysis
-better chance of getting correct documents earlier in the project	-requirements need to be good enough (not perfect)
-we have some chance to manage the present, plan for the future	-we cannot test until development/unit testing is finished
-decide which projects to do, using cost-benefit analysis	-slow

Question from the Class: Why do we study the Classical life-cycle model in CS 430?

Answer:

1. Understand why OO is better.

2. Many organizations still use Classical.
3. Much legacy code still exists, that was written using Classical techniques.

2.2 Example: Object-Oriented Paradigm

Refer to the Examples document on LEARN (Lecture 02)

Answers to the Questions for Discussion

1. How would each group react to the introduction of the Object-Oriented paradigm?
   (a) IT Team
      i. initial resistance to change
      ii. We should have fewer regression faults (Definition 3.4.4).
      iii. We should have less "spaghetti" code this way (**Instructor Remark:** Through structured programming, we should already have eliminated spaghetti code, even classically.)
      iv. Maintenance should become easier
      v. less conflict in data terms
         A. more control should we need to add / modify something
      vi. Classes can be re-used
      vii. When interfaces need to change, it can be hard to co-ordinate between multiple teams who maintain multiple classes.
      viii. learning curve with OO
   (b) Business Partners
      i. initial resistance to change
      ii. How do we give requirements in an OO way?
      iii. buy in to the benefits of more re-use, cheaper maintenance, going forward
      iv. learning curve with OO
2. What would be some advantages and disadvantages of adopting the Object-Oriented paradigm?
   Answer:

Pros	Cons
-classes can belong to libraries, not systems	-learning curve!
-hence classes are more easily reused	-it can be more difficult to enforce code standards
-fewer regression faults	-increased costs of development maintenance work
-can test classes independently of each other, unlike developing the whole system before we can test	

Question from the Class: Why does OO not come with a life-cycle picture, as Classical does?
Answer:

1. The change from Classical to Object Orientation is more a change of mindset than of methodology.
2. We change our mindset from building **one monolithic thing** (Classical) to building **many smaller classes that do work for us together** (OO). Many life-cycle models (including Waterfall) can be used to build these classes effectively.

2.3 Maintenance Aspects

We will look at maintenance in the context of the Classical (aka Waterfall) Life-Cycle Model, invented in 1970. Phases:

1. Requirements
   (a) Elicit client requirements.
   (b) Understand client needs.
2. Analysis
   (a) Analyze client requirements.
   (b) Draft specification document - formal.
   (c) Draft Software Project Management Plan (SPMP).
3. Design
   (a) Design architecture - divide software functionality into components.
   (b) Draft detailed design for each component.
4. Implementation
   (a) Coding (development) - code & document each component

(b) Unit test each individual component
(c) Integration (system) testing - combine components, test interfaces among components
(d) 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.
(e) 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.
   (a)

   Definition 2.3.1. ***Corrective Maintenance****: Removal of residual faults while software functionality & specs remain relatively unchanged. (aka fix* ***production problems****)*
   (b)

   Definition 2.3.2. ***Perfective Maintenance****:*
   i. *Implement changes the client thinks will improve effectiveness of product (e.g. additional functionality, reduce response time) (aka* ***enhancements*** *or* ***upgrades****)*
   ii. *Specs must be changed*
   (c)

   Definition 2.3.3. ***Adaptive Maintenance:***
   i. *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*
   ii. *Specs may change to address the new environment*
6. Retirement
   (a) Product is removed from service: functionality provided by software is no longer useful / further maintenance is no longer economically feasible.

2.3.1 The Importance of Postdelivery Maintenance

- Shelf life of good software: 10, 20, even 30 years
- Good software is a model of real world & real world keeps changing, therefore software must change too.

- Cost of Post delivery Maintenance continues to go up, while (possibly surprisingly) cost of implementation is nearly flat.

Example: My first project at OpenText was to develop a **Consolidated Customer Database**. After the initial scrubbing of the data, management opted not to re-scrub the following year. The database withered and died because management was unwilling to pay for post delivery maintenance.

2.4 Requirements, Analysis and Design Aspects

Key Facts:

1. The earlier in the life cycle a fault is found, the cheaper it is to fix. (See Figures 1.5 and 1.6 on pp13-14 of the text.)
2. Correcting a fault in the early phases usually just requires changing a document.
3. Hence the requirements, analysis and design phases need to be improved.

2.5 Team Development Aspects

Remarks:

1. Hardware keeps getting cheaper and cheaper, and able to run more and more complex programs.
2. Hence modern software must be developed by **teams**.
3. But this can lead to problems, e.g.
   (a) Communication becomes challenging when teams are far apart geographically, especially when they are in different time zones.
   (b) Interpersonal problems can undermine team effectiveness.
   (c) if a call to a module written by another developer mentions the arguments in the wrong order. (If the variable types are the same, then even the compiler may not catch this fault).
4. Software Engineering must include techniques for ensuring teams are properly managed.

2.6 The Object-Oriented Paradigm

Problems With The Classical Paradigm

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.

Reason: Classical techniques focus on **data** or **operation**, but **not both**.

Contrast With The Object-Oriented Paradigm:

1. The object-oriented paradigm treats **data** (attributes) and **operations** (methods) together, as equally important.

2.7 The Object-Oriented Paradigm In Perspective

Remarks:

1. Like any software production technique, the OO paradigm must be applied correctly to be effective.
2. The OO paradigm is the best technique invented so far; yet is is sure to be superseded by a superior technique in the future.

2.8 Ethical Issues

Remarks:

1. Since software is developed by people, there are ethical issues connected with software development.
2. Software engineers commit to these ethical principles (each is explained more fully in the text):
   (a) Public
   (b) Client and Employer
   (c) Product
   (d) Judgment
   (e) Management
   (f) Profession
   (g) Colleagues
   (h) Self

3 Lecture 03 - Iteration and Incrementation

Outline

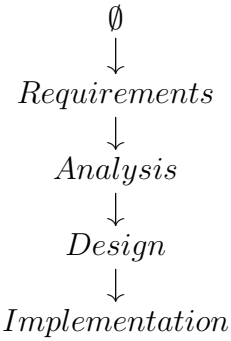
1. Introduction to Software Development Life-Cycle Models
2. Software Development in Theory
3. Winburg Example
4. Iteration and Incrementation

3.1 Introduction to Software Development Life-Cycle Models

Where Chapter 1 attempted to describe software development in the ideal world, Chapter 2 attempts to describe software development in the real world.

3.2 Software Development in Theory

Idealized Software Development



In theory, we do not have to deal with any changes once the Requirements phase is complete.

3.3 Winburg Example

See the description in the text and in the examples for the course.

Key Observations from the Example:

1. Anecdote from a business line at a bank: IT was perceived as very slow to respond to requests for changes to their systems.
   In Lecture 02 we stated that the slowness of getting projects done using the Classical model was a drawback of that model.
   Corollary: IT resisted accepting changes to the requirements once the requirements were complete.
2. The Example Provides an Example of the Software Crisis:
   (a) Requirements were incomplete: there was no requirement describing performance (not meeting client's needs).
   (b) Assuming the project completes at the end of Episode 4, the project was
      i. very late, and

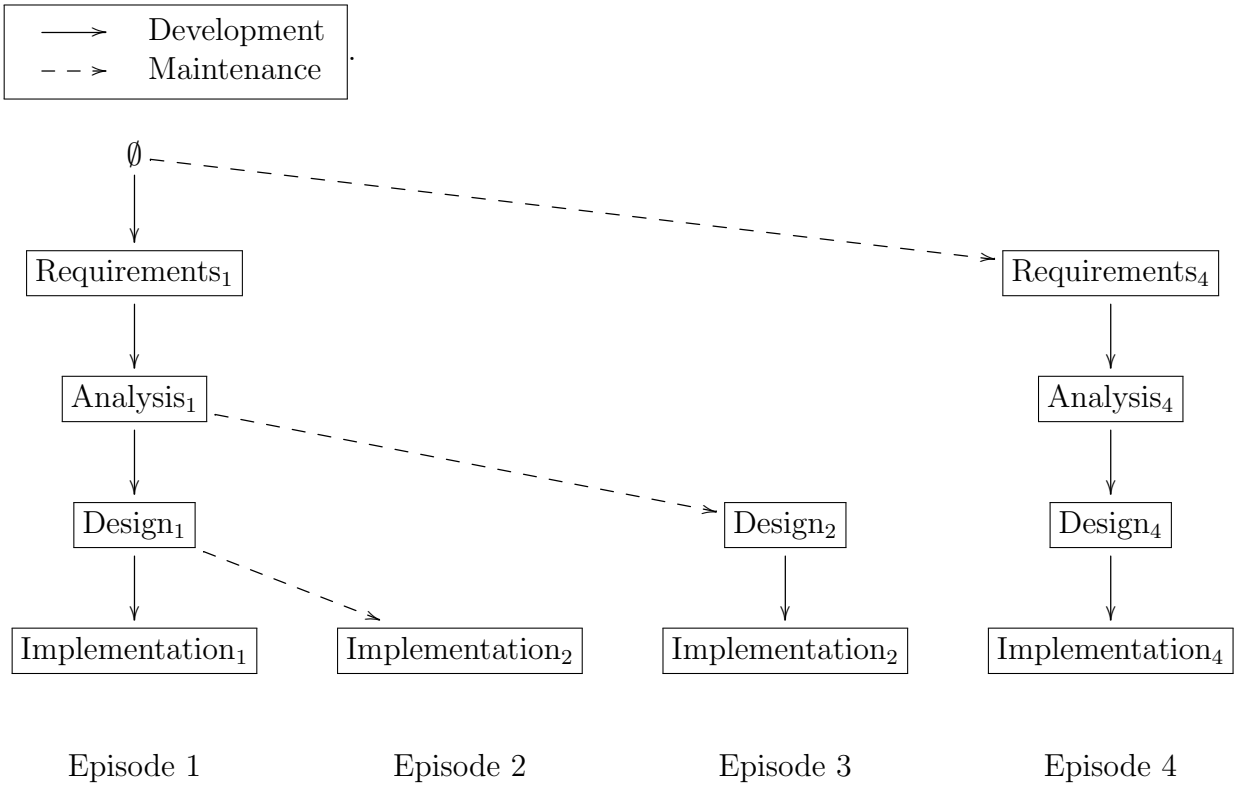
ii. over budget.

(c) Nothing in the Example explicitly says that there were faults in the completed software product.

3. There was lots of rework, which was needed because each episode spawned a classical life-cycle effort (iteration), in which work done in one iteration had no easy way to feed into the next, if they overlapped in time.
4. This was caused, in part, by the overall slowness of the Classical model.
5. Starting to develop the single-precision fix before confirming it would provide the desired performance improvement was a waste of time.
6. Some re-use was achieved when the scanning software was packaged and re-sold.
7. There was testing throughout the case.
8. More testing throughout Episode 1 might have revealed the performance problems sooner.
9. **Instructor Remark:** Perhaps a small pilot project, prototyping the scanning hardware and software together would have revealed the performance problems earlier. This is a **proof of concept prototype**. We will discuss such prototypes again in Chapter 5.
10. Packaging and re-selling was a win.
11. The project ultimately did satisfy the specification.
12. Based on our own work experience to date, this is not the worst case we have seen so far. (The text agrees with us on this point.)

Morals of the Example:

1. The Classical model is most effective when the IT team can work without accepting changes to the requirements after the requirements are complete. Changes to requirements (e.g. adding the performance requirement, the Mayor's later change) negatively affects software quality, delivery dates, and budgets.
2. BUT in the real world, change is inevitable. We cannot prevent change; we must learn to manage it.
3. Here is a sketch of Figure 2.2 in the text for an example of the **evolution-tree life cycle model** for this example, using the key:



This is an ad-hoc response to the **moving target problem** (Definition 3.4.1).

Key Idea: Each Episode spawns a new (sometimes partial) instance of the Classical development life-cycle model.

4. The rest of Chapter 2 is concerned with adapting the Classical life-cycle model to manage change.

3.4 Iteration and Incrementation

Key Idea: Think of Iteration and Incrementation as a generalization of the ad-hoc, **evolution tree** life-cycle from the Example. Break the project into (say 4) **increments**, then each increment runs as a small waterfall project. See Figures 2.4 through 2.6 in the text.

Goals:

1. Get the benefits of Classical structure, while
2. Being more tolerant of change than the Classical model is.

Useful Definitions:

1.

Definition 3.4.1. *The* ***moving target problem*** *occurs when the requirements change while the software is being developed.*
Unfortunately this problem has no solution!

2.

Definition 3.4.2. ***Scope creep*** *aka* ***feature creep*** *is a succession of small, almost trivial requests for additions to the requirements.*
Remarks:

(a) If the IT team can refuse such changes, then scope creep need not contribute to the moving target problem.
(b) All too often the IT team does not have this power.

3.

Definition 3.4.3. *A* ***fault*** *is the (observable) result of a coding mistake made by a programmer.*

4.

Definition 3.4.4. *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.*

5.

Definition 3.4.5. *A* ***regression test*** *provides evidence that we have not unintentionally changed something that we did not intend to change (i.e. that there are no regression faults).*
Typical Strategy:

(a) Choose test cases that all fall under all the business rules **not** touched by the project specification.
(b) Execute the production and the modified code against the chosen test cases.
(c) Compare the outputs. **Success = no differences**.

6.

Definition 3.4.6. ***Miller's Law*** *states that, at any one time, a human is only capable of concentrating on approximately seven chunks of information.*
Why this Matters for Software Engineering:

(a) One person can effectively work on at most seven items at once.

(b) Any software project of significant size will have many more than seven components.
(c) Hence we must start by working on ≤ 7 highly important things first, temporarily ignoring all the rest.
(d) This is the technique of **stepwise refinement** (Definition 9.1.1). This technique will come up again in Chapter 5.

When you have time, you may enjoy listening to this YouTube video (the visual is just a static image) about Iteration & Incrementation:
`https://youtu.be/FTygpfEFFKw`
Next Time: (Almost) all the remaining life-cycle models are variations on Iteration and Incrementation.

4 Lecture 04 - Life-Cycle Models

Outline
1. Other Life Cycle Models
   (a) Code and Fix Life-Cycle Model
   (b) Waterfall (Modified) Life-Cycle Model
   (c) Rapid Prototyping Life-Cycle Model
   (d) Open Source Life-Cycle Model
   (e) Agile Processes
   (f) Synchronize and Stabilize Life-Cycle Model
   (g) Spiral Life-Cycle Model
2. Comparison of Life-Cycle Models

4.1 Other Life Cycle Models

4.1.1 Code and Fix Life-Cycle Model

Key Idea: Implement the product without requirements, specification or design.
Remarks:
1. See Figure 2.8 in the text or on slide 17 for Chapter 2; but know that it is the only possible picture without requirements, specification or design.
2. **Strengths:**
   (a) This technique may work on very small systems ($\leq$ 200 lines of code).

(b) Easy to incorporate changes to requirements.
(c) Generates a lot of lines of code (whether this is actually a strength depends on organizational norms).

3. **Weaknesses:**
   (a) This technique is totally unsuitable for systems of any reasonable size.
   (b) This technique is unlikely to yield the optimal solution.
   (c) Slow.
   (d) Costly.
   (e) Likelihood of regression faults is high.

Remarks:

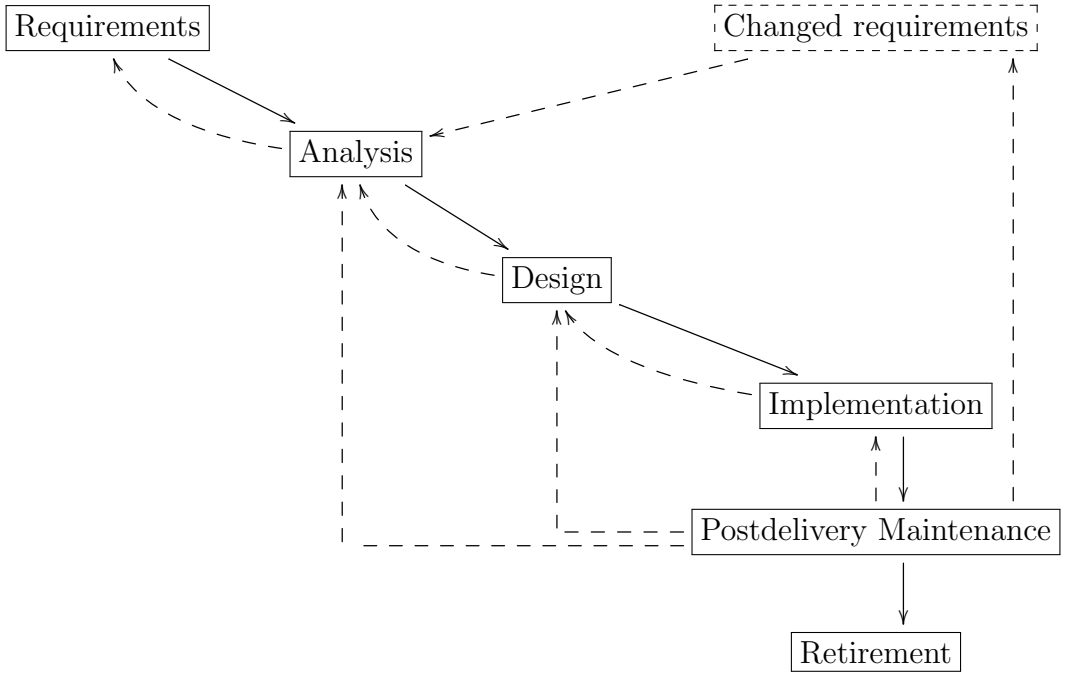
1. It is appropriate (and really the only choice) for a user base of size 1, e.g. for any programming assignment you would do for a CS assignment at uWaterloo.
2. We met this model once before: it was the only model in existence before the Waterfall model was introduced in 1970.

4.1.2 Waterfall (Modified) Life-Cycle Model

Key Idea: Augment the "vanilla" waterfall diagram, to add the "feedback loops" during the project, and for post-delivery maintenance. Here is a sketch

of Figure 2.9 in the text, using the key: 

.



Remarks:

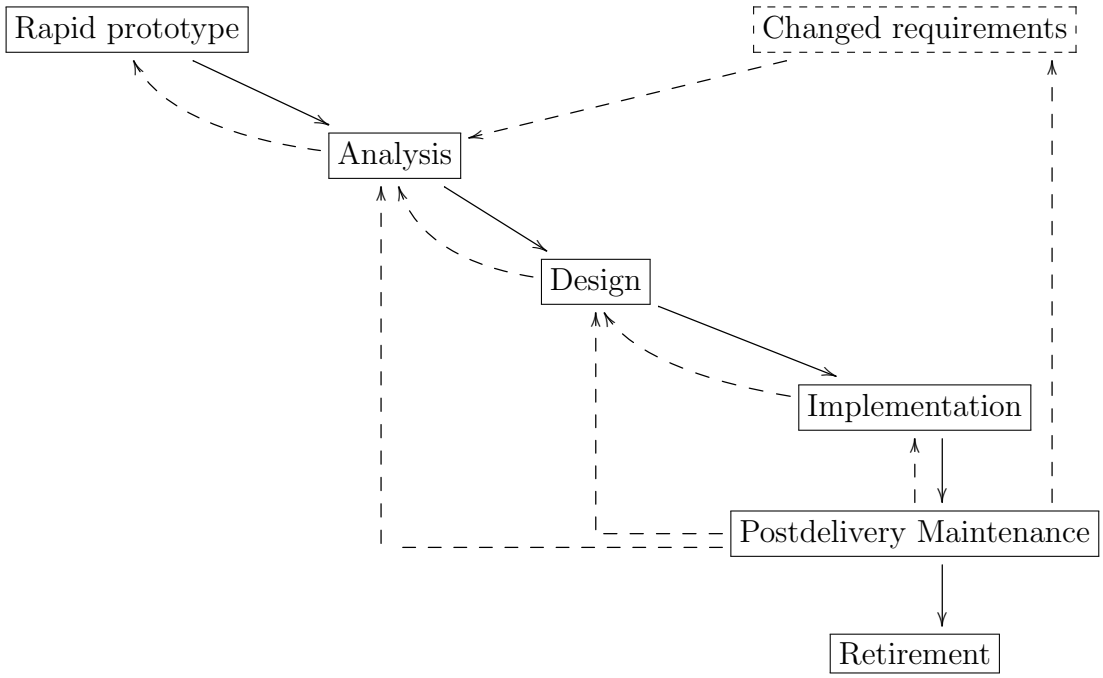
1. No phase is complete until all its documents are complete, and the output(s) of the phase are approved by the **SQA (Software Quality Assurance)** team.
2. Testing is carried out throughout the project.
3. **Strengths:**
   (a) Discipline enforced by SQA.
4. **Weaknesses:**
   (a) Specification documents are often written in a way that does not enable the client to understand what the finished product will look like.
      i. Hence specification documents may not be fully understood before they are approved.
      ii. Hence the finished product may not actually meet the client's needs.

   The next model, **rapid prototyping**, is an adaptation if the waterfall model to address this key weakness.

4.1.3 Rapid Prototyping Life-Cycle Model

Here is a sketch of Figure 2.10 in the text, using the key: 

.



Remarks:

1. This diagram looks almost identical to that for Waterfall (Modified).
2. **Key Difference:** Requirements has been replaced with Rapid Prototype. Huh?

Definition 4.1.1. *A **rapid prototype** is a working model that is functionally equivalent to a subset of the software product.*

Motivation: Develop a rapid prototype (during Requirements phase) to let the client interact and experiment with it early. This way the requirements document can be written with higher confidence that the software product it describes will meet the client's needs. Users can give better feedback from working with a rapid prototype than from reading a long requirements document.

Examples:

1. If the product is a payroll system, then a rapid prototype might have a subset of the screens and might produce mocked-up pay stubs, but might not have any database updating or batch processing behind the scenes.

Remarks:

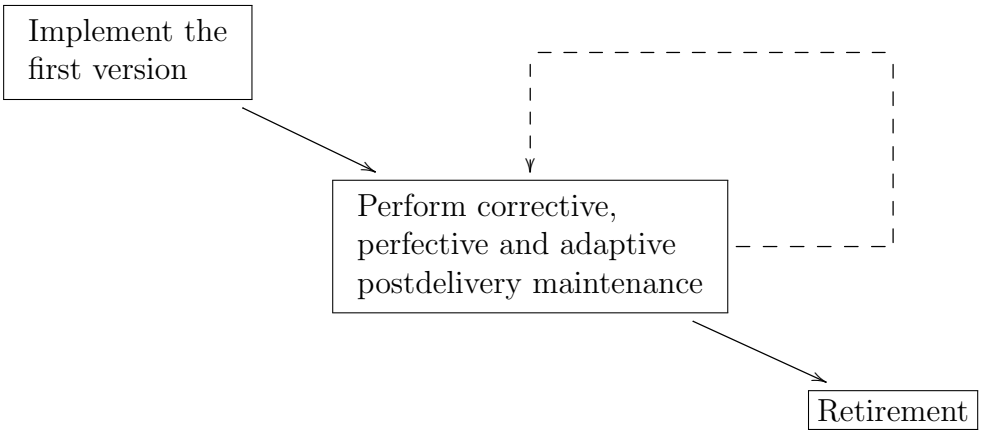
1. The feedback loops from the waterfall model are less heavily used here.
2. The word "rapid" is crucial. Speed is of the essence!

Summary: The purpose of a rapid prototype is to **improve requirements**.

4.1.4 Open Source Life-Cycle Model

Here is a sketch of Figure 2.11 in the text, using the key: 

.



Key Idea: Open Source software projects proceed in two phases:

1. A single individual has an idea for a program (e.g. MySQL, LibreOffice, Notepad++, R, Linux, Firefox, Apache, etc.), builds the initial version, and makes it available free of charge to anyone who wants a copy.
2. (Informal) If there is sufficient interest, then users become co-developers (co-maintainers) for Post-Delivery Maintenance:
   (a) Report / correct faults (Corrective Maintenance)
   (b) Add additional functionality (Perfective Maintenance)
   (c) Port the program to new platforms (Adaptive Maintenance)
3. All participants can offer suggestions:
   (a) new features
   (b) new platforms

4. Participation is voluntary and unpaid.
5. **Roles:**
   (a) Core group: dedicated maintainers
   (b) Peripheral group: suggest bug fixes from time to time
6. Success depends on the interest generated by the initial version.

Many open source projects do not amount to anything. But there have been some spectacularly successful examples (mentioned at the beginning of the section).

Reasons Why Open Source Projects Are Successful:

1. Perception that the initial release is a "winner" (most important)
2. Large potential user base

Instructor Remarks:

1. Participation in an Open Source project is voluntary and unpaid.
2. The idea of Open Source is in direct conflict with a corporation's need to achieve competitive advantage, by writing good software.

4.1.5 Agile Processes

Guiding Principles

1. Communication
2. Speed: Satisfying the Client's needs as quickly as possible (ideally new versions every 2-3 weeks)

According to the **Scrum Method**, we iterate through the following two phases until the backlog of tasks is empty.

Requirements	Sprints
User Stories Prioritization Build Backlog of Tasks	Daily Meetings Eventually Reassign Tasks

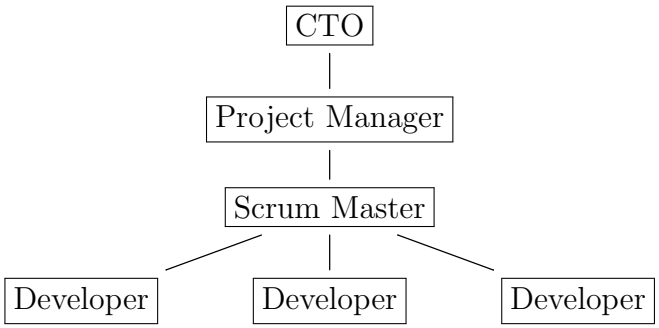
Techniques to ensure frequent delivery of new versions:

1. **timeboxing:** Fix an amount of time to work on a task; do as much as possible on the task during that time window. Agile processes demand fixed time, not fixed features.
2. **daily 15 minute stand-up meeting (to raise and resolve issues):** Each team member answers five questions:
   (a) What have I done since yesterday's meeting?
   (b) What am I working on today?
   (c) What problems are preventing me from achieving my goal for today?

(d) What have we forgotten?
(e) What did I learn that I would like to share with the team?

Differences Between Agile and Classical:

1. Diagram of Team Organization:



   (a) 1-week "sprints"
   (b) Each sprint gets us closer to the ultimate goal.
2. Iterative process
3. One phase need not finish before the next can start
4. A client representative sits with the IT team
5. No specializations
6. Members from all different areas work together at different times
7. Working software is prioritized over detailed documentation
8. test-driven development

Remarks:

1. **Strengths:**
   (a) Speed
   (b) Flexibility
   (c) Team Cohesion
   (d) Some history of success with smaller projects.
2. **Weaknesses:**
   (a) Heavy on meetings
   (b) Not scalable with team size
   (c) This technique is untested on large projects (many software professionals have expressed doubts that this will be successful)

When you have time, you may enjoy watching this YouTube video about Iteration & Incrementation Leading to Agile Processes:
`https://youtu.be/Vlc2r_U30yo`

Remarks on Agile Processes:

1. The text makes a big deal of **Extreme Programming (XP)**, and states that a key feature of XP is **pair programming**. I had always suspected that this was a bit too rigid - now we have this suspicion confirmed by presentations from students who have worked under this model. It made a lot more sense to me that the groups formed to do the work need not always be pairs - they are whatever is appropriate to the task at hand.

4.1.6 Synchronize and Stabilize Life-Cycle Model

This is Microsoft's adaptation of Iteration and Incrementation.

1. Pull requirements from the clients.
2. Write Specification document.
3. Divide the work into four **builds** (most important features in earlier builds):
   (a) critical
   (b) major
   (c) minor
   (d) trivial

   N.B. Developers can add requirements during a build.
4. Carry out each build using small teams working in parallel.
5. **Synchronize** at the end of each day, then
6. **Stabilize** at the end of each build (then freeze).

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-ly stabilization ensure developers will all work in the same direction
6. Good for large projects

Weaknesses:

1. So far, this has only been used successfully at Microsoft

4.1.7 Spiral Life-Cycle Model

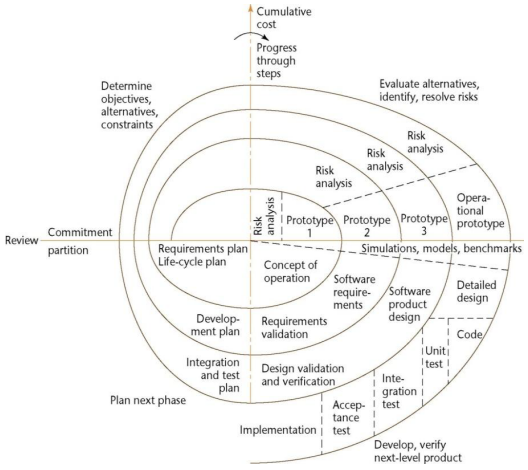
This incorporates elements of several of the earlier models.

Key Problem: There are many risks associated with software development projects, which if realized will mean that the project is a failure.

Key Ideas:

1. Minimize risks inherent in software development by the (repeated) use of **proof-of-concept prototypes** and other means.
2. N.B. Unlike **rapid prototypes**, which aim to improve requirements by letting users interact with a subset of the target functionality, a **proof-of-concept prototype** aims to determine whether an architecture design is good (e.g. will it perform quickly enough?)

Figure 2.13: Spiral, Full

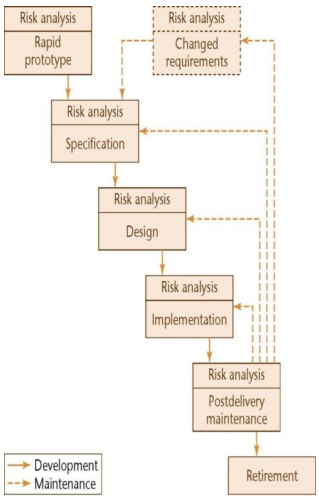


Remarks:

1. The quadrants in the above diagram could be labelled:

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

Figure 2.12: Spiral, Simplified



Remarks:

1. **Strengths:**
   (a) Emphasis on alternatives and constraints supports re-use, and software quality.
   (b) This technique encourages doing the correct amount of testing.
2. **Weaknesses:**
   (a) This model is only meant for internal building of large-scale software.
   (b) If risks are not analyzed correctly, then all may appear fine even when the project is headed for disaster.
   (c) Makes the (often wrong) assumption that software is developed in discrete phases, when in reality, software is developed iteratively and incrementally (like in the Winburg example).

4.2 Comparison of Life-Cycle Models

Here is Figure 2.14 from the text:

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

5 Lecture 05 - The Unified Process I

Outline

1. Introduction to the Software Process
2. The Unified Process
3. Iteration and Incrementation Within the Object-Oriented Paradigm

4. Requirements Workflow
5. Analysis Workflow
6. Design Workflow
7. Implementation Workflow
8. Test Workflow
   (a) Requirements
   (b) Analysis
   (c) Design
   (d) Implementation
9. Post-Delivery Maintenance
10. Retirement

5.1 Introduction to the Software Process

Definition 5.1.1. *The* ***software process*** *encompasses the activities, techniques and tools used to produce software.*

Remarks:

1. With Definition 5.1.1, we could have defined the **software crisis** (Definition 1.3.1) as our inability to manage the software process effectively.
2. The goal of Software Engineering is to improve the software process.

5.2 The Unified Process

1. **Idea:** We want to explore the **Unified Process**, which will be our software development methodology for the rest of the course. This methodology will be
   (a) object-oriented, and
   (b) extendable.
2. This is Figure 3.1 on p88 of the text:

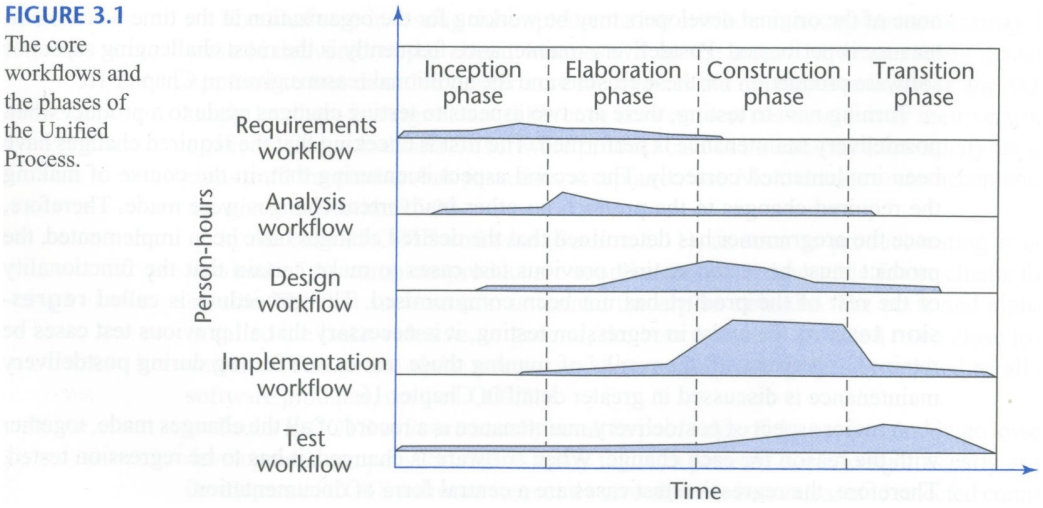


FIGURE 3.1
The core workflows and the phases of the Unified Process.

3. The workflows have
   (a) a **technical context**, e.g. the business case in the requirements workflow is technical, and
   (b) a **task orientation**.
4. The phases have
   (a) an **economic** context, e.g. the business case in the Inception phase is economic, and
   (b) a **time orientation**.

Definition 5.2.1. *An **artifact** is a work product from a workflow.*

Questions From the Class

1. **Q:** Why are the artifacts tied to workflows, instead of to phases?
   A: Since it is more natural to think of the artifacts from a task point of view, it is more natural to tie the artifacts to the workflows than to the phases.

5.3 Iteration and Incrementation Within the Object-Oriented Paradigm

Key Idea: 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 is. Under the **Unified Process**,

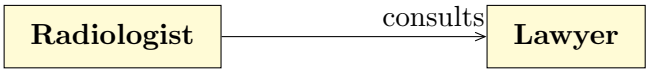
1. the phases are the increments, and

2. we iterate through the increments (each having a mini-Classical shape) to complete the project.

Definition 5.3.1. ***UML*** *stands for the* ***Unified Modelling Language****.*

Definition 5.3.2. *A* ***model*** *is a set of UML diagrams which describes one or more aspects of the software product to be developed.*

Example:



"The Radiologist class consults the Lawyer class."

Motivation:

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 a block of verbiage. "A picture is worth a thousand words."
3. The visual nature of a UML model fosters collaborative **refinement**.

Remarks:

1. Presenting the entire Unified Process would take more time and space than we have during the remainder of this term, hence we will stick to the highlights.
2. The names of the workflows (mostly) match the names of the phases of the classical model. The descriptions of the workflow artifacts that follow are similar to the outputs of the corresponding classical phases.
3. The classical model tied tasks and time together in sequence. The unified model separates tasks and time.

Summary of Requirements, Analysis, Design and Implementation Workflows

1. Each workflow corresponds (task-based) with the Classical phase having the same name.
2. See the notes below for full details.

5.4 Requirements Workflow

1. **Goal:** Determine the client's needs, and determine what constraints there are (often referred to as **concept exploration**).

2. **Pitfalls:** Do the Lecture 05 Example Here.
   Problems Found With Requirements Given in Example
   (a) Standings Changes: do we actually want all increases, all decreases, or both?
   (b) Standing Display: If we don't include previous, then changes will not be clear from the report
   (c) Some students with no changes in their standings should be included (e.g. if they are still on probation)
   (d) MAV used for criteria, but only CAV is displayed on the report - unclear
   (e) Conflicting sort criteria in different parts of the specification
   (f) Missing criterion for filtering down to just the program for which I am responsible
   (g) Should the two inclusion criteria be combined with `AND` or with `OR`?
   (h) And possibly more...

   Moral: To summarize the Example, requirements **artifacts** can be
   (a) incorrect (only the client can detect this)
   (b) ambiguous (e.g. `AND` versus `OR` in the inclusion criteria - IT can detect this)
   (c) incomplete (e.g. missing criterion to filter down to the program - only the client can detect this)
   (d) contradictory (e.g. conflicting sort criteria - IT can detect this)
3. Using UML diagrams correctly helps to mitigate the above problems with requirements.

5.5 Analysis Workflow

1. **Goal:** Analyze and refine the requirements to achieve the level of detail needed to begin designing the software, and to 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**.

5.6 Design Workflow

1. **Goal:** Show **how** the product is to do what it must do.
2. More precisely, 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.

5.7 Implementation Workflow

1. **Goal:** 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.

5.8 Test Workflow

Goal: Ensure the correctness of the artifacts from the other workflows.

5.8.1 Requirements

1. Key Idea: **traceability:** every later artifact must trace back to a requirement artifact.
2. Key Observation: 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. We will delve deeper into this in Chapter 6.

5.8.2 Analysis

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

5.8.3 Design

1. Again, design artifacts must trace back to analysis artifacts.

2. Tactic: Again, hold a **review** of design artifacts (likely without the client this time)

5.8.4 Implementation

Remarks:

1. This will be explained in detail in Chapter 6.

The testing must include

1. desk checking (programmer)
2. unit testing (SQA)
3. integration testing (SQA)
4. product testing (SQA)
5. (user) acceptance testing (SQA and client)

Remarks:

1. Some projects also incorporate **alpha** and **beta** testing (usually the beta version is the first version that the public would see).
2. Although it is tempting, **alpha** testing should **not** replace thorough testing by the SQA group.

5.9 Post-Delivery Maintenance

1. This is **not** an afterthought - it must be planned from the start.
2. **Pitfall:** lack of adequate documentation (deadline pressures during initial delivery contribute to this)
3. Testing of changes must include **positive** and **regression** testing.

   Definition 5.9.1. ***Positive testing*** *means testing that what you intended to change was changed in the desired way.*

   Strategy:

   (a) Select test cases exercising the changed business rules.
   (b) Compare pass 0 (no changes) against pass 1 (with changes).
   (c) Confirm that the pass 1 output has the desired changes applied.

5.10 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. The software product must be replaced if the business need persists.
3. True **retirements** are rare.

6 Lecture 06 - The Unified Process II

Outline

1. The Phases of the Unified Process
   (a) The Interaction Between Phases and Workflows
   (b) Inception
   (c) Elaboration
   (d) Construction
   (e) Transition
2. One- Versus Two-Dimensional Life-Cycle Models
3. Improving the Software Process
4. Capability Maturity Models

6.1 The Phases of the Unified Process

6.1.1 The Interaction Between Phases and Workflows

As pointed out last lecture,

1. phases have a **time orientation** (questions like "when do we need to do deliver artifact x?" can be answered by naming a phase), and
2. workflows have a **task orientation** ("with what related artifacts should artifact x be grouped?" can be answered by naming a workflow).

Motivation to Separate Workflows and Phases: Why 1-D models like Waterfall break down in practice: they assume that the time and task orientations agree with one another. In reality they do not, because of the **moving target problem**. Iteration and Incrementation leads to the splitting of tasks and time, which in turn leads us to the Unified Process.

Global Remarks:

1. What follows is a summary of what artifacts are typically produced for each workflow and phase.
2. Different projects will require different timelines, and hence different phase breakdowns.

6.1.2 Inception Phase

Goal: Determine whether it is worthwhile to develop the target software product. Is it economically viable to build it?
Here we explain the interaction between the Inception phase and each workflow (i.e. which workflow artifacts are typically produced during the Inception phase).

1. Requirements Workflow Key Steps:
   (a) Understand what is

   Definition 6.1.1. *The **domain** of a software product is the place (e.g. TV station, hospital, air traffic control tower, etc.) in which it must operate.*

   (b) Build

   Definition 6.1.2. *A **business model** is a description of the client's business process, i.e. "how the client operates within the domain".*

   (c) Determine the project **scope**.
   (d) The developers make the initial

   Definition 6.1.3. *A **business case** is a document which answers these questions.*
   i. *Is the proposed software cost effective? Will the benefits outweigh the costs? In what timeframe? What are the costs of **not** developing the software?*
   ii. *Can the proposed software be delivered on time? What impacts will be realized if the software is delivered late?*
   iii. *What risks are involved in developing the software, and how can these risks be mitigated? Similarly to above, what risks are there if we do not build it? There are three major risk categories.*
      A. *Technical Risks*
      B. *Bad Requirements*
      C. *Bad Architecture*

2. Analysis Workflow
   (a) Extract the information needed to design the architecture.
3. Design Workflow
   (a) Create the design.

   (b) Answer all questions required to start Implementation.
4. Implementation Workflow
   (a) Usually little to no coding is done during the inception phase.
   (b) Sometimes it will be necessary to build a **proof-of-concept prototype**.
5. 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 (from analysis workflow, usually documented in UML)
8. plan for Elaboration phase (we must always plan for the next phase)
9. initial version of the business case (**overall aim of Inception phase**). This describes the scope of the software product plus financial details.
   (a) If software is to be marketed, then this includes revenue projections, market estimates and initial cost estimates, etc.
   (b) If software is to be used in-house, then this includes the initial cost/benefit analysis.

6.1.3 Elaboration Phase

Goals:

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

Deliverables from the Elaboration Phase:

1. the completed domain model
2. the completed business model
3. the completed requirements artifacts
4. the completed analysis artifacts
5. updated version of the architecture
6. updated list of risks

7. SPMP
8. the completed business case

6.1.4 Construction Phase

Goal: Produce the first operational-quality version of the software product (the **beta** release).
Deliverables from the Construction Phase:

1. initial user manual and other manuals, as appropriate
2. completed version of the architecture
3. updated list of risks
4. SPMP updated
5. if needed, the revised business case

6.1.5 Transition Phase

Goal: Ensure the client's requirements have been met (using, in part, feedback from the users of the beta version).
Deliverables from the Transition Phase:

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

Example: Do the Lecture 06 Example Here. In each answer that follows, we state

1. the workflow first, then
2. the phase(s) during which that artifact will likely be delivered.

1. Business Model: First Draft
   (a) Requirements
   (b) Inception
2. Business Model: Completed
   (a) Requirements
   (b) Inception or Elaboration
3. User Requirements: First Draft
   (a) Requirements
   (b) Inception
4. User Requirements: Completed
   (a) Requirements
   (b) Elaboration (or Construction)
5. Inspection Report: User Requirements

   (a) Testing
   (b) Elaboration (or Construction)
6. SPMP: First Draft
   (a) Analysis
   (b) Inception (or Elaboration)
7. SPMP: Completed
   (a) Analysis
   (b) Transition (or Construction)
8. System Architecture Design: First Draft
   (a) Design
   (b) Elaboration (or Inception)
9. Code: First Draft
   (a) Implementation
   (b) Construction (or Elaboration for "low hanging fruit")
10. Code: Ready for Deployment
   (a) Implementation
   (b) Transition (or Construction for "low hanging fruit")

6.2 One- Versus Two-Dimensional Life-Cycle Models

Fundamental Question: Can one's "position" in the Life-Cycle be described along only one axis (in which case the model is 1-D), or does one need two axes (in which case the model is 2-D)?
Examples (not exhaustive):

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

Remarks:
1. Two-dimensional models are more complicated, but for all the reasons from Chapter 2, we cannot avoid working with them, especially the Unified Process.
2. The Unified Process is the best model we have so far, but it is sure to be superseded by a superior methodology in the future.

6.3 Improving the Software Process

1. Our **fundamental problem** in Software Engineering is our inability to manage the software process effectively (the text cites a US government report from 1987 to justify this statement).
2. The US DoD responded by creating the Software Engineering Institute (SEI) at Carnegie Mellon University.
3. SEI in turn created the **Capability Maturity Model (CMM)** initiative.

6.4 Capability Maturity Models

The CMMs are a related group of strategies for improving the software process, irrespective of the choice of Life Cycle model used. The word "maturity" indicates that an organization matures as it improves its processes.

1. SW-CMM (software) We will focus on this one.
2. P-CMM (HR, "P" for "people")
3. SE-CMM (systems engineering)
4. IPD-CMM (integrated product development)
5. SA-CMM (software acquisition)

These are gathered up as **CMM Integration (CMMI)**.

Idea of SW-CMM:

1. 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.
2. An organization advances incrementally through five levels of maturity.
   (a) Initial
      i. No sound software engineering practices are in place: everything is done ad-hoc.
      ii. Most projects are late and over budget.
      iii. Most activities are responses to crises, rather than preplanned tasks.
      iv. Many organizations are at the initial level!
   (b) Repeatable
      i. Basic software Project Management practices are in place ("repeatable" because planning & management techniques are based on past experience with similar projects).

ii. Some measurements are taken (e.g tracking costs, schedules).
iii. Managers identify problems as they arise and take immediate corrective action to prevent them from becoming crises.

(c) Defined
i. The process for software production is fully documented (management / technical).
ii. There is continual process improvement.
iii. **Reviews** are used to achieve software quality goals.
iv. **CASE** environments increase quality / productivity further.

Definition 6.4.1. ***CASE*** *stands for* ***Computer Aided/Assisted Software Engineering****.*

We will discuss CASE in more detail in Chapter 5.

(d) Managed
i. The organization sets quality/productivity goals for each software project.
ii. Both are measured continually and corrective action is taken when there are **unacceptable** deviations. (Statistical methods are used to distinguish a random deviation from a meaningful violation of standards.)
iii. Typical measure: # faults / 1000 lines of code, in some time interval.

(e) Optimizing
i. The goal is **continual process improvement**.
ii. Statistical quality / process control techniques are used to guide the organization.
iii. **Positive Feedback Loop:** Knowledge gained from each project is used in future projects. Therefore productivity and quality steadily improve.

7 Lecture 07 - Teams I

Outline

1. Team Organization
2. Classical Chief Programmer Teams
3. Democratic Teams
4. Beyond Chief Programmer and Democratic Teams

7.1 Team Organization

1. To develop a software product of any significant size, a **team** is required.
2. **Question:** Suppose that a software product requires 12 person-months to build it. Does it follow that 4 programmers could complete the work in 3 months?
   Answer: No:
   (a) There are new issues (communication / integration / etc.) once a team is involved, as contrasted with an individual.
   (b) Not all programming tasks can be fully shared in time or in sequencing. Maybe the software product naturally has three chunks, or maybe it has many chunks with complicated dependencies.
   (c) A project manager's **Gantt Chart** is a tool for managing the dependencies in a team project.
3. Another key point:

   Definition 7.1.1. ***Brooks' Law*** *states that* ***adding programmers to an already late project makes it later.***

   Fred Brooks observed this phenomenon while managing the development of OS/360, for IBM 360 mainframes.
   One reason (not the only one):
   The more programmers there are on a team, the more communication paths there are, and hence the slower overall communication becomes.

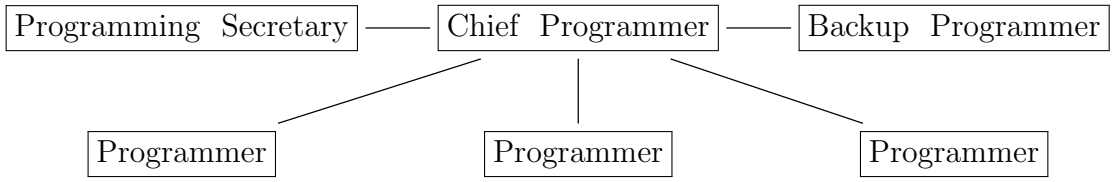
Remarks:

1. The rest of Chapter 4 focuses on team organization applied to the **implementation workflow**. The problems and solutions are equally applicable to other workflows.

7.2 Classical Chief Programmer Teams

- A six-person team without a chief programmer has $\binom{6}{2} = 15$ communication paths. Every pair of people can communicate directly with each other.
- A six-person team with a chief programmer (this is Figure 4.3 in the

text) looks like:



There are only 5 communication paths.

•

Definition 7.2.1. *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 members (responsible for every line of code)
- Backup Programmer
  - needed in case chief programmer wins the lottery, gets sick, falls under a bus, changes jobs, etc.
  - as competent as the chief programmer in all respects.
  - does tasks independent of the design process (e.g. selecting test cases for black box testing)
- Programming Secretary (aka Librarian)
  - maintain the production library, including all project documentation
    - source code (responsible for compiling)
    - JCL (job control language, for running mainframe batch jobs)
    - test data (executes all tests)
- Programmer
  - They just program.

Strengths:

1. This has been enormously successful in a few cases. It was first used in 1971, by IBM, to automate the clippings data bank ("morgue") of

the New York Times and other publications. If you have the text, see §4.3.1.

Weaknesses:

1. Chief/Backup Programmers are hard to find.
2. Secretaries are also hard to find.
   (a) We seek someone with strong technical skills, then demand only clerical work from them.
3. The Programmers may be frustrated at being "second class citizens" under this model.

Remarks:

1. In reality, most team organizations lie somewhere between the two extremes of classical chief programmer (very hierarchical) and democratic (non-hierarchical).

7.3 Democratic Teams

- Basic concept:

  Definition 7.3.1. ***egoless programming****:*
  1. *Code belongs to the team as a whole, not to any individual.*
  2. *Finding faults is encouraged.*
  3. *Reviewers show appreciation at being asked for advice, rather than ridiculing programmers for making mistakes.*

-

  Definition 7.3.2. *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 (by definition, a democratic team has no leader).
- **Strengths:**
  1. Rapid detection of faults $\rightarrow$ high quality code.
  2. Addresses the problem of programmers being overly attached to their own code.
- **Weaknesses:**
  1. managerial issues as mentioned above

2. It is hard to create such a team. Such teams tend to spring up spontaneously from the "grass roots", often in the context of research as contrasted with the context of business.
3. A certain organizational culture is required before such a team can emerge.

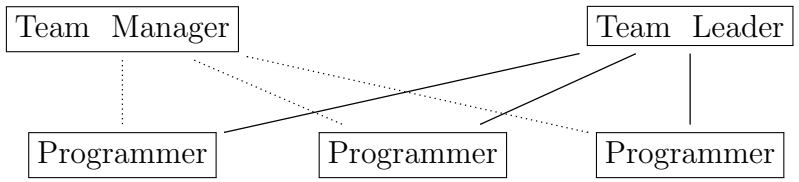
7.4 Beyond Chief Programmer and Democratic Teams

These two team organizations sit at opposite ends of the continuum:

Classical Chief Programmer	**Democratic**
very hierarchical	little hierarchy
little individual freedom	much individual freedom

A Conflict Inherent in the Chief Programmer Model:

1. The Chief Programmer must attend all code reviews. They are responsible for every line of code, as the Technical Manager of the Team.
2. The Chief Programmer must not attend any code reviews. They are the HR Manager, and reviews should never be used for HR performance appraisals (see Chapter 6).
3. Resolution
   (a) **Suggestion From Class:** Divorce the performance appraisals from review results (sounds good in theory; problematic in practice).
   (b) **From Text:** Split the Chief Programmer role into a Team Manager (non-technical) and Team Leader (technical).
4. Requirement Clearly demarcate the duties of each role, wherever there could be some overlap. The picture below can be scaled up, as in Figure 4.5 of the text.



Student Questions:

1. Do the Programming Secretary and Backup Programmer roles still exist here?
   Instructor Answer: In my experience, no:

(a) The first version of this model dates from 1971, when a program was a stack of punch cards. In today's environment, with "soft" code, and robust version control, the Programming Secretary is obsolete.
(b) While a backup programmer is no longer explicitly identified, every organization must grapple with succession planning, somehow.

2. Does the Backup Programmer role also have to be split up, like the Chief Programmer does?
   Instructor Answer: No, the split would occur when the Backup Programmer is promoted to Chief Programmer.
3. What are the strengths of the Classical Chief Programmer Team Organization?
   Instructor Answer:
   (a) Key Observation: This team organization is extremely rigid with respect to which communication paths are permitted. Recall Brooks' Law (Definition 7.1.1). A Classical Chief Programmer is least vulnerable to the effects of Brooks' Law, because the number of communication paths only grows linearly as the number of programmers grows. Under a team organization in which any programmer can talk to any other programmer, the number of communication paths grows as the square of the number of programmers.
   (b) As will be suggested later for **Synchronize and Stabilize** teams, this team organization fosters a team culture in which all team members work together towards a common goal.

8 Lecture 08 - Teams II

Outline

1. Synchronize and Stabilize Teams
2. Teams for Agile Processes
3. Open Source Programming Teams
4. People Capability Maturity Models
5. Choosing an Appropriate Team Organization

8.1 Synchronize and Stabilize Teams

1. Recall that so far, the synchronize-and-stabilize model has only been

used within Microsoft.

2. **Rule #1:** The developers must adhere strictly to the agreed upon time to check their code in for that day's synchronization.
3. **Rule #2:** If a developer's code prevents the product from being compiled for that day's synchronization, then the problem must be fixed immediately, so that the rest of the team can test and debug.
4. **Remark:** The culture of the organization must fully support Rules #1 and #2 before this life-cycle model and team organization can have any success.
5. **Strengths:**
   (a) Encourages individual programmers to be creative and innovative, a characteristic of a **democratic team**.
   (b) The synchronization step ensures that all programmers work together for a common goal, a characteristic of a **chief programmer team**.
6. **Weaknesses:**
   (a) There is no evidence yet that this model can work outside of Microsoft.
   A Possible Explanation: There is something unique about Microsoft's culture, which has yet to be replicated elsewhere.

8.2 Teams for Agile Processes

1. Advantages of **pair programming**:
   (a) "Two heads are better than one."
   (b) It should produce high quality code.
   (c) Fewer typos/small bugs.
   (d) Programmers do not test their own code.
   (e) All knowledge is not lost if one programmer leaves. The remaining programmer from the pair can train a new pair programmer.
   (f) Less experienced programmers can learn from more experienced programmers.
   (g) The technique promotes group ownership of the code, a key feature of **egoless programming**.
2. Disadvantages of **pair programming**:
   (a) Twice the person-hours: more expensive.
   (b) It can be slow; programmers can become distracted.
   (c) Subjective disagreements can waste time.

(d) Each programmer must regard the other as an equal.
(e) Feedback given by teammates may not always be constructive.
(f) Extremely shy people might dislike this technique - they must speak up while (pair) programming and during (daily) meetings. Overbearing people might dominate.

3. More research is needed to determine whether the benefits outweigh the costs.

8.3 Open Source Programming Teams

1. Reasons why people would **not** want to participate in an open source project:
   (a) unpaid
   (b) philosophical disagreements about direction.
   (c) Since the programmer does not own the code, he/she cannot monetize the work at all, even after the development is done.
   (d) You must give up control over the finished product (or even your own piece of it).
   (e) Intellectual property problems: You give away what you produce during such an effort.
2. Reasons why people choose to participate in an open source project:
   (a) You are empowered to fix problems.
   (b) It benefits everyone to have some successful open-source products available.
   (c) the sheer enjoyment of accomplishing a worthwhile task. "Making the world a better place."
      i. Volunteers must continue to perceive that the project is worthwhile; they will drift away if the project begins to seem futile.
   (d) the learning experience
      i. Employers frequently view experience gained working on a large, successful open source project as more desirable than additional academic qualifications.
      ii. Hence it is crucial that the project be perceived as possibly successful to retain its programmers.
   (e) An organization depends on an open source application, and hence is motivated to devote resources to supporting the open source team.

   In summary, an open source project must be viewed at all times as

a "winner" to attract and retain volunteers to work on it. **Corollary:** The key individual behind the project must be a superb motivator.

Morals:

1. For success, top-calibre programmers are required. Such programmers can succeed, even in an environment as unstructured as an open-source one.
2. The way that a successful open-source project team is organized is essentially irrelevant to the success/failure of the project.

8.4 People Capability Maturity Models

1. Recall that P-CMM was the capability maturity model for People. It describes best practices for managing and developing the workforce of an organization.
2. Similarly to SW-CMM, an organization progresses through five levels of maturity with the aim of continuously improving individual skills and engendering effective teams.
3. Also similarly to SW-CMM, P-CMM is a framework for improving an organization's processes for managing and developing its workforce, and no specific choice of team organization is put forward.

8.5 Choosing an Appropriate Team Organization

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
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

1. There is no one choice of team organization that is optimal in all situations. Different strengths / weaknesses will matter more at different times.

2. In practice most teams are organized according to some variant of the (modified) chief programmer model.

9 Lecture 09 - Tools of the Trade I

Outline

1. Stepwise Refinement
2. Cost-Benefit Analysis
3. Divide and Conquer
4. Separation of Concerns
5. Software Metrics

9.1 Stepwise Refinement

Definition 9.1.1. ***Stepwise refinement*** *is a technique by which we defer nonessential decisions until later, while focusing on the essential decisions first.*

1. This is a response to **Miller's Law** (Definition 3.4.6).
2. The text presents a mini case study in §5.1.1, about designing an updater for a master file.
3. The details of each step in the text example of stepwise refinement are not important. The important thing is to notice how decisions get deferred until they must be settled in later iterations.
4. I like the fact that the example in the text is a refinement of a design. I have found this technique extremely fruitful during my own design work. It would be less effective during the implementation workflow, for example.
5. **Key Challenge:** Decide which issues must be handled in the current refinement, and which can be deferred until a later refinement. There is **no algorithm** to decide! Experience and human intuition are required.
6. In my experience the technique can be effective when working on a problem either individually or in a group. In a group setting
   (a) **brainstorming** can be used in the early stages, and
   (b) more structured **reviews** can be used in the later stages.
7. Features of Brainstorming

(a) The problem to be solved may initially be unclear e.g. the team might start with a symptom, and understand the underlying cause through brainstorming.
(b) All team members are encouraged to speak, especially the shy ones.
(c) No editing in the first round(s), when ideas are being suggested. Editing happens after all ideas have been suggested.
(d) **Student Question:** Is brainstorming always top-down then?
Instructor Answer: Brainstorming can be
   i. top-down for Intuitives, and
   ii. bottom-up for Sensors.

Either way can be productive.

9.2 Cost-Benefit Analysis

Definition 9.2.1. ***Cost-Benefit Analysis*** *is*

1. *a technique for determining whether a possible course of action would be profitable, in which we*
2. *compare estimated future benefits against estimated future costs,*
3. *often referred to as the "balance sheet view".*
4. *When selecting from among several options, the optimal choice maximizes the difference*

$$(\textit{estimated benefits}) - (\textit{estimated costs}).$$

Pitfalls:

1. We must quantify **everything** to start. Some things are easier to quantify than others.
   (a) Tangible benefits are easy to measure, e.g. estimated revenue from a new product.
   (b) Intangible benefits can be more challenging e.g. the reputation of your organization (think Facebook, recently).
      i. To quantify intangible benefits, we must make **assumptions**, e.g. Facebook hacks will cause 5000 users to close their accounts - then we can estimate lost advertising revenue, using historical data.
         A. Advantage: With better assumptions (say from improved historical data or from a new team member who brings

new experiences) we can obtain more accurate quantifications of our intangible benefits.

B. As software engineering practitioners, we must gather all of our information by **ethical** means!

9.3 Divide and Conquer

Definition 9.3.1. *To **divide and conquer** is to break a large problem down into sub-problems, each of which is easier to solve.*

Remarks:

1. Like Stepwise Refinement, Divide and Conquer is also common sense.
2. This is the "oldest trick in the book".
3. This is a component of the Unified Process.

Definition 9.3.2. *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** (Definition 16.3.1) that can be implemented as a single unit.*
4. During the design workflow:
   (a) Partition the implementation workflow into corresponding manageable pieces, termed **subsystems**.
5. During the implementation workflow:
   (a) Implement each subsystem in the chosen programming language.
6. **Key Problem:** There is **no algorithm** for deciding how to partition a software product into smaller pieces. Experience and human intuition are required.
7. **Example:** My last large project at SunLife (2003) was developing a new intranet site. The homepage consisted of four independent quadrants. Hence the home page naturally broke down into four analysis packages, and later, into four subsystems and four streams of implementation.

9.4 Separation of Concerns

Definition 9.4.1. *A software product has **separation of concerns** if it is broken into components that overlap as little as possible with respect to their functionalities.*

Remarks:

1. Separation of concerns is a "new and improved" version of divide and conquer. The new guiding principle for how to divide up the components is to reduce or eliminate the overlaps in their functionalities.

Motivation:

1. Minimize the number of regression faults! If separation of concerns is truly achieved, then changing one module cannot affect another module.
2. When done correctly, this also facilitates **re-use** of modules in future software products.
3. Manifestations of separation of concerns:
   (a) design technique of **high cohesion:** maximum interaction within each module (§7.2).
   (b) design technique of **loose coupling:** minimum interaction between modules (§7.3).
   (c) **encapsulation** (§7.4).
   (d) **information hiding** (§7.6).
   (e) **three tier architecture** (§8.5.4).
4. Tracking which modules were written by weaker programmers may facilitate more proactive maintenance work.

Moral: Separation of concerns is desirable for Software Engineering.

9.5 Software Metrics

Definition 9.5.1. *A **metric** is anything that we measure quantitatively.*

1. We need **metrics** to detect problems early in the software process before they become crises.
2. Examples:
   (a) # LOC, lines of code (measures **size**)
   (b) # faults / 1000 lines of code (measures **quality**)
   (c) (after deployment) mean time between failures (measures **reliability**)
   (d) number of person-months to build (measures **size**)
   (e) staff turnover (high turnover affects budgets and timelines)
   (f) cost
3. Two types (Exercise: categorize the list above into one of these types):
   (a) **product** metrics, e.g. # lines of code for a software product.

   (b) **process** metrics, e.g.
      i. # lines of code for the organization.
      ii. $\frac{\text{\# of faults detected during product development}}{\text{\# of faults detected during product's lifetime}}$, taken over all software products in the organization. (measures effectiveness of fault detection during development)
4. Some metrics are clearly tied to a certain workflow (e.g. we cannot count lines of code until implementation)
5. Five essential, fundamental metrics for a software project:
   (a) Size (e.g. in # Lines of Code)
   (b) Cost to develop / maintain (in dollars)
   (c) Duration to develop (in months)
   (d) Effort to develop (in person-months; or as in my experience in person-days)
   (e) Quality (in number of faults detected during the project)
6. There is no universal agreement among software engineers about which metrics are right, or even preferred.

10 Lecture 10 - Tools of the Trade II

Outline

1. Taxonomy of CASE
2. Scope of CASE
3. Software Versions
   (a) Revisions
   (b) Variations
   (c) Moral
4. Configuration Control
   (a) Configuration Control During Postdelivery Maintenance
   (b) Baselines
   (c) Configuration Control During Development
5. Build Tools
6. Productivity Gains with CASE Technology

10.1 Taxonomy of CASE

1. Recall, per Definition 6.4.1, CASE stands for **Computer Aided/Assisted Software Engineering**, not **Computer Automated Software En-**

gineering.

2. At present, a computer is a tool of, and not a replacement for, a software professional.
3. CASE tools used during the
   (a) earlier workflows (requirements, analysis, design) are called **front-end** or **upperCASE** tools, and
   (b) later workflows (implementation, postdelivery maintenance) are called **back-end** or **lowerCASE** tools.
4. **Examples**
   (a) **data dictionary** - list of every data item defined in the software product. Some things to include:
      i. an English description of every item in the dictionary
      ii. **Module names** ✓
      iii. **Procedure names:** ✓
         A. parameters, and
         B. their types,
         C. locations where they are defined (i.e. which module), and
         D. description of purpose
      iv. **Variable names:** ✓
         A. types, and
         B. locations (i.e. which module & procedure) where they are defined
   (b) **consistency checker** - to confirm that every data item in the specification document is reflected in the design, and vice versa.
   (c) **report generator**
   (d) **screen generator** - for creating **data capture** screens.
5. Taxonomy
   (a) Combining multiple tools creates a **workbench**.
   (b) Combining multiple workbenches creates an **environment**.
   (c) So our taxonomy is
   tools (task level) → workbenches (team level) → environments (organization level).

10.2 Scope of CASE

1. Primary motivations for implementing CASE:
   (a) Produce high-quality code.
   (b) Have up-to-date documentation at all times.

   (c) Automation makes maintenance easier.
   (d) Do everything more quickly, hence more cheaply.
2. For example, if a specification is created by hand, there may not be any way to tell whether the document is current by reading it. On the other hand, if the specification is maintained within CASE software, then the latest version is the one the CASE software displays.
3. Similarly, other documentation about the software is easier to maintain inside of CASE software.
4. **Online documentation**, **word processors**, **spreadsheets**, **web browsers**, and **email** are CASE tools.
5. **Coding tools** of CASE include
   (a) **text editors** (including **structure editors** which are sensitive to syntax, including **online interface checking**), **debuggers**, **pretty printers / formatters**, etc.
6. An **operating system front end** allows the programmer to issue operating system commands (e.g. compile, link, load) from within the editor.
7. A **source-level debugger** automatically causes trace output to be produced. An **interactive source-level debugger** is what its name says.
8. **Programming-in-the-small:** coding a single module.
9. **Programming-in-the-large:** coding at the system level.
10. **Programming-in-the-many:** software production by a team.

10.3 Software Versions

10.3.1 Revisions

Definition 10.3.1. *A **revision** is created when a change is made, e.g. to fix a fault.*

1. Old revisions must be retained for reference, e.g.
   (a) if a fault is found at a site still running the old revision,
   (b) for auditing and
   (c) for other reasons.

10.3.2 Variations

Definition 10.3.2. *A **variation** is a slightly changed version that fulfills the same role in a slightly changed situation.*

Examples:

1. two printer drivers, one for a laser printer and one for an inkjet printer, or
2. optimizing an application to run on different platforms, e.g. desktop vs. smart phone.

Remarks:

1. Often the variation is also embedded into the file name.

10.3.3 Moral

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

10.4 Configuration Control

Definition 10.4.1. *A **configuration** of a software product is a list, for every code artifact, of which version is included in the S/W product.*

Definition 10.4.2. *A **configuration control tool** is a CASE tool for managing configurations (Definition 10.4.1).*

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

   Definition 10.4.3. *A **derivation** is a detailed record of a version of the S/W product, including*
   *(a) the variation/revision of each code element (i.e. the **configuration**),*

(b) the versions of the compilers/linkers used to assemble the product,
(c) the date/time of assembly, plus
(d) the name of the programmer who created the version.

7. A **version-control** tool is required to effectively track derivations.

10.4.1 Configuration Control During Postdelivery Maintenance

1. There is an obvious problem when a team maintains a software product.
2. Suppose that two different programmers receive two different fault reports. Suppose further that fixing both faults require changes to the same code artifact.
3. Without any new process in place, the programmer #2 will undo programmer #1's changes at deployment time.
4. See the next subsection for a possible solution to this problem, using **baselines**.

10.4.2 Baselines

1. When multiple programmers are working on fixing faults, a **baseline** is needed.
2. A **baseline** is a set of versions of all the code artifacts in a project (i.e. what versions are in production right now).
3. A programmer starts by copying the baseline files into a **private workspace**. Then he/she can freely change anything without affecting anything else.
4. The programmer **freezes** the version of the artifact to be changed to fix the fault. No other programmer can modify a frozen version.
5. After the fault is fixed, the new code artifact is promoted to production, modifying the baseline.
6. The old, frozen version is kept for future reference, and can never be changed.
7. This technique extends in the natural way to multiple programmers and multiple code artifacts.
8. **Instructor Remark:** In my experience, the strict technique described here is too slow. Instead developer #2 starts work right away, and incorporates developer #1's changes as soon as they are promoted to production. SQA needs to be kept informed in this situation!
   One could argue that this technique is vulnerable to exponential growth of effort as the number of faults in a code artifact increases. The

instructor counter-argues that if we achieve **separation of concerns** in our software products, then the probability of >> 2 simultaneous faults in one code artifact is low.

Student Question: What if #1 and #2 actually touch the same code?
Instructor Answer: I recommend using the same technique, being mindful that extra care will be needed when

1. incorporating #1's changes into #2's version, and
2. doing SQA (e.g. what should be the test cases and expected results for pass 0 and for pass 1?).

10.4.3 Configuration Control During Development

1. During Development and Desk Checking, changes are too frequent for configuration control to be useful.
2. We definitely want configuration control to be in force by the time we deploy to production.
3. The text author recommends that configuration control should apply once the code artifact is passed off to the SQA group.
   (a) In practice, we can decide when between the end of development and the time of deployment to begin enforcing configuration control.
4. The same configuration control procedures as those for postdelivery maintenance should then apply.
5. Proper version control permits management to take corrective action if project deadlines start to slip (as they are then aware of the status of every code artifact).

10.5 Build Tools

Definition 10.5.1. *A **build tool** selects the correct compiled-code artifact to be linked into a specific version of the S/W product.*

1. Some organizations may not want to purchase a complete configuration-control solution. Then at least a version control tool must be used in conjunction with a **build tool** (Definition 10.5.1).
2. **Issue:** While a version control tool assists programmers in deciding which version of the source code is the latest, compiled code does not automatically get a version attached to it. Possible solutions (present in class only if time permits):

(a) Automatically re-compile and re-link every night. Obviously this is expensive.
(b) Use a tool like `make` to decide more intelligently, based on date and time stamps of compiled code. This idea has been incorporated into many different programming environments.

3. **Student Question:** What is the difference between a **build tool** (Definition 10.5.1) and a **configuration control tool** (Definition 10.4.2)?
   Answer: The purpose of a **build tool** is to make certain we have the correct compiled code artifacts linked in to a specific version of the S/W product. This can be effective for a small organization, managing one version of a S/W product at one location. This explains why auto-recompiling each night is a viable technique.
   A **configuration control tool** is needed to manage multiple revisions of multiple variations. E.g. for a large organization which must manage multiple configurations running simultaneously across multiple locations.

10.6 Productivity Gains with CASE Technology

1. Research to date shows a modest gain in productivity following the introduction of CASE tools to an organization.
2. Other benefits of using CASE tools:
   (a) faster development
   (b) fewer faults
   (c) better usability (e.g. from a screen generator)
   (d) easier maintenance
   (e) improved morale on the IT team
3. 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)

11 Lecture 11 - Testing I - Non-Execution-Based Testing

Outline

1. Quality Issues
   (a) Software Quality Assurance (SQA)
   (b) Managerial Independence
2. Non-Execution Based Testing
   (a) Reviews
   (b) Walkthroughs
   (c) Managing Walkthroughs
   (d) Inspections
   (e) Comparison of Walkthroughs and Inspections
   (f) Strengths and Weaknesses of Reviews
   (g) Metrics for Inspections

11.1 Quality Issues

Terminology:
Recall Definition 3.4.3 of a **fault**.

Definition 11.1.1. *A **failure** is an observed incorrect behaviour of the S/W product caused by a fault.*

Definition 11.1.2. ***Error*** *is the amount by which the software product's output is incorrect (i.e. the statistical sense of error).*

Definition 11.1.3. *A **defect** is a generic term for a fault, failure or error.*

Definition 11.1.4. ***Quality*** *describes the extent to which the S/W product satisfies its specification.*

11.1.1 Software Quality Assurance (SQA)

1. Quality alone is not enough: the software also must be easily maintained.
2. SQA must be built in throughout the project, not simply imposed by the SQA group at the end of a workflow, say.
3. **Primary Duty of SQA Group:** Ensure

   (a) the quality (usual English meaning) of the S/W process, and thus ensure
   (b) the quality (S/W product meaning) of the S/W product.
4. Once the developers complete a workflow and check their work, the SQA team must verify that all artifacts are correct.

11.1.2 Managerial Independence

1. Development and SQA teams should be led by independent managers, neither of whom can overrule the other.
2. **Reason:** Often major faults are found as the delivery deadline approaches. Then the S/W organization must decide between
   (a) delivering the S/W on time with faults (likely development's choice, since they are more often driven by deadlines), or
   (b) fixing the faults and delivering late (likely SQA's choice, since they are more often driven by quality).
3. Both must report to a third manager, who must then make the decision about what to do on a case-by-case basis.

11.2 Non-Execution Based Testing

11.2.1 Reviews

Definition 11.2.1. *A **review** is a **walkthrough** or an **inspection**.*

Common Features of All Reviews

1. **non-execution based testing**, i.e. no code is executed for this type of test
2. centred around a **meeting** of key stakeholders
3. chaired by SQA representative (because SQA has the biggest stake in getting all artifacts correct, and not letting faults slip through).
4. the meeting is to test a document to **identify**, but **not attempt to fix**, faults in that document.
   Reasons:
   (a) committee's solution is usually of lower quality than that of a trained expert
   (b) committee's solution takes 4-6 times as much effort as an individual's.
   (c) not all "faults" identified during a review are truly faults.

(d) takes too much time: a review should last at most two hours.

11.2.2 Walkthroughs

The two steps for a walkthrough:

1. preparation
2. team analysis of the document

4-6 participants (e.g. for an analysis artifact):

1. SQA (chair - as above)
2. manager responsible for requirements (previous workflow)
3. manager responsible for analysis (current workflow)
4. manager responsible for design (next workflow)
5. client representative (maybe less crucial for later workflows)

11.2.3 Managing Walkthroughs

1. Two fundamental approaches to conducting a walkthrough:
   (a) **participant** driven
   (b) **document** driven (usually more detailed and hence more time-consuming and effective at finding faults)

   Here is where the text explains (again) why reviews should **not** be used for performance appraisals:
   (a) in a review, success = finding faults
   (b) in a performance appraisal, success = finding no faults

11.2.4 Inspections

The five steps for an inspection (each with a formal process):

1. **overview** document author gives the overview; document is distributed to the participants.
2. **preparation** participants examine the document, individually.
3. **inspection** quick document walkthrough; immediately commence fault-finding.
4. **rework** document author corrects all faults noted in the written report from step 3.
5. **follow-up** moderator ensures that every fault identified has been fixed, and that no new faults were introduced in the process of fixing.

Roles for an Inspection (e.g. for a Design Artifact):

1. **moderator** (from SQA)

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** (SQA, a different person than the moderator)

11.2.5 Comparison of Walkthroughs and Inspections

Remarks:

1. Although inspections are more costly, there is evidence (see text §6.2.3) that they are more effective at finding faults.

11.2.6 Strengths and Weaknesses of Reviews

Strengths:

1. effective at detecting faults, especially
2. early in the life-cycle, when they are cheaper to fix.

Weaknesses:

1. A large S/W product's artifacts are hard to review, unless they consist of smaller, independent components. Using OO helps to mitigate this.
2. Effectiveness of review team is hampered if not all documentation from the previous workflow is completed yet.

11.2.7 Metrics for Inspections

Examples:

1. **inspection rate:**
   (a) **requirements/designs:** # of pages / hour
   (b) **code:** # of lines of code / hour
2. **fault density**
   (a) **requirements/designs:** # of faults (major/minor) / page
   (b) **code:** # of faults (major/minor) / 1000 lines of code
3. **fault detection rate:** # of faults (major/minor) detected / hour
4. **fault detection efficiency:** # of faults (major/minor) detected / person-hour

Remarks:

1. The metrics attempt to measure our effectiveness at finding faults.
2. A spike in any of these metrics might indicate that the quality of the S/W development work has suddenly decreased, and not that fault detection has suddenly improved.

12 Lecture 12 - Testing II - Execution Based Testing

Outline

1. Execution-Based Testing
2. What Should Be Tested?
   (a) Utility
   (b) Reliability
   (c) Robustness
   (d) Performance
   (e) Correctness

12.1 Execution-Based Testing

Testing is a crucial part of any software development life-cycle. However we must keep in mind that (as Dijkstra points out), testing can demonstrate the presence of faults in a software product, **not** their absence.
One reason: Test cases are only as good as the tester selecting them. Things can get missed.

12.2 What Should Be Tested?

Definition 12.2.1. ***Execution-Based Testing*** *is a process of inferring certain behavioural properties of a software product based, in part, on the results of running the software product in a known environment with selected inputs.*

Three Troubling Details About Definition 12.2.1:

1. Testing is an inferential process. There is no algorithm for determining whether faults are present! A test run with correct results may simply fail to expose a fault.
2. What do we mean by **known environment**? We can never fully know our environment. The text gives the example that an intermittent hardware fault in the computer's memory system could cause failures, even if the code is perfect.
3. What do we mean by **selected inputs**? With a real-time system, no control over the inputs is possible, e.g.

(a) an avionics system in an aircraft, for which the inputs describing the current state of the aircraft's flight cannot be controlled (a partial solution to this problem is provided by a **simulator**), and
(b) a system for controlling trains.

Remarks:

1. Despite these problems, Definition 12.2.1 is the best one available.

12.2.1 Utility

Definition 12.2.2. *The **utility** of a software product is the extent to which the software product meets the user's needs when operated under conditions permitted by its specification.*

Elements:

1. Is the software product easy to use?
2. Does the software product perform useful functions?
3. Is the software product cost effective?

Remarks:

1. If a software product fails a test of its utility, then testing should proceed no further!

12.2.2 Reliability

Definition 12.2.3. *The **reliability** of a software product measures the **frequency** and **severity** of its failures.*

Elements:

1. **mean time between failures** (Recall Definition 11.1.1.) Long times → more reliable.
2. **mean time to repair failures** Long times → less reliable.
   (a) **Also important (often overlooked):** time required to fix the **effects** of the failure (e.g. correcting corrupted data). Long times → less reliable.

12.2.3 Robustness

Elements:

1. range of operating conditions (permissible by the specifications, or not)
   (a) A robust product has a wide range of operating conditions, including some outside its specification.

2. possibility of unacceptable output given acceptable input
   (a) A robust product produces acceptable output, given acceptable input.
3. acceptability of output given unacceptable input
   (a) A robust product produces acceptable output (e.g. a helpful error message instead of a crash), even given unacceptable input.

12.2.4 Performance

1. It is crucial to verify that a software product meets its constraints with respect to:
   (a) Space constraints which can be critical in miniature applications, e.g.
      i. missile guidance systems as in the text, or
      ii. smart phone apps.
   (b) Time constraints which can be critical in **real time** applications, e.g.
      i. measuring core temperature in a nuclear reactor as in the text, or
      ii. controlling signals on a railroad network.

12.2.5 Correctness

Definition 12.2.4. *A software product is* ***correct*** *if it satisfies its output specification, without regard for the computing resources consumed, when operated under permissible (pre-)conditions.*

Remarks:

1. This definition is **partial correctness**. It tacitly assumes that the program **terminates**.

Problems with Definition 12.2.4:

1. Specifications can be wrong.
   (a) Then a software product can be correct, but not be acceptable.
      i. Cute text example: a sort program whose specification omits the requirement that the sorted list be a permutation of the original list - clearly not acceptable!
2. (a) A software product can be acceptable, but not be correct.
      i. Cute text example: a compiler, faster than its predecessor, but which prints a spurious error message (which is easily

ignored) in one rare situation. This compiler is acceptable. However it is not correct since producing the spurious error message is not part of its specification.

13 Lecture 13 - Testing III - Proving Program Correctness

Outline

1. Testing Versus Correctness Proofs
   (a) Example of a Correctness Proof
   (b) Correctness Proof Mini Example
   (c) Correctness Proofs and Software Engineering
2. Who Should Perform Execution-Based Testing?
3. When Testing Stops

13.1 Testing Versus Correctness Proofs

Definition 13.1.1. *A **correctness proof** is a mathematical technique for demonstrating that a program is **correct** (see Definition 12.2.4).*

Remarks:

1. The text shows a technique which uses **flowcharts** to argue the correctness of a program. This technique is cute, but is not used in industry. So we will **not** spend time learning this technique.
2. My lecture notes/slides show an example (directly stolen from CS 245) which uses the technique of **Hoare triples** (assertions inserted into the code, which assemble into a proof of program correctness). This technique is used in industry, but requires mathematical machinery (**Predicate logic**, a.k.a. **first-order logic**) which we do **not** have as a pre-requisite for CS 430. So we will not spend time learning this technique in detail either.
3. It is enough for us to know that the Hoare triple technique can be carried out, with enough mathematical background, and patience.

13.1.1 Example of a Correctness Proof

Prove the total correctness of the program below, which computes a **factorial**.

$(\!|x \geq 0|\!)$
```
y = 1 ;
z = 0 ;
while (z != x) {
        z = z + 1 ;
        y = y * z ;
}
```
$(\!|y = x!|\!)$

At the `while` statement:

x	y	z	$z \neq x$
5	1	0	true
5	1	1	true
5	2	2	true
5	6	3	true
5	24	4	true
5	120	5	false

From the trace and the post-condition, a candidate **loop invariant** is $y = z!$ Here is the annotated program.

$(\!\|x \geq 0\|\!)$	
$(\!\|1 = 0!\|\!)$	assignment
`y = 1 ;`	
$(\!\|y = 0!\|\!)$	assignment
`z = 0 ;`	
$(\!\|y = z!\|\!)$	assignment
`while (z != x) {`	
$\quad(\!\|(y = z! \wedge z \neq x)\|\!)$	partial-`while`
$\quad(\!\|y(z+1) = (z+1)!\|\!)$	implied (b)
`  z = z + 1 ;`	
$\quad(\!\|yz = z!\|\!)$	assignment
`  y = y * z ;`	
$\quad(\!\|y = z!\|\!)$	assignment
`}`	
$(\!\|(y = z! \wedge z = x)\|\!)$	partial-`while`

$(\!|y = x!|\!)$ implied (b)

Proof of implied (a): $\{x \geq 0\} \vdash 1 = 0!$.
This result is obvious, by definition of factorial.
Proof of implied (b): $\{(y = z! \wedge z \neq x)\} \vdash y(z+1) = (z+1)!$.
This result is obvious.
Proof of implied (c): $\{(y = z! \wedge z = x)\} \vdash y = x!$.
This result is also obvious.
This completes the proof of partial correctness.
Proof of Termination: The factorial code from earlier has a **loop guard** of $z \neq x$, which is equivalent to $x - z \neq 0$.
What happens to the value of $x - z$ during execution?

$(\!|x \geq 0|\!)$
```
y = 1 ;
z = 0 ;                          At start of loop: x − z = x ≥ 0✓
while (z != x) {
        z = z + 1 ;              x − z decreases by 1 ✓
        y = y * z ;              x − z unchanged
}
```
$(\!|y = x!|\!)$

The value of $x - z$ will eventually reach 0. The loop then exits and the program terminates. ✓
This completes the proof of total correctness.

13.1.2 Correctness Proof Mini Example

See the Example document.
Moral: Even if a proof of a program's correctness has been found, the program must still be tested thoroughly.

13.1.3 Correctness Proofs and Software Engineering

1. Proposed reasons why correctness proving should not be a standard software engineering technique:

(a) S/W Engineers lack the mathematical training to write correctness proofs. **Partial Refutation:**
   i. This may have been true in the past.
   ii. However many CS graduates today (including all from uWaterloo) do have the required mathematical background.

(b) Correctness proving is too time consuming and hence too expensive. **Partial Refutation:**
   i. Costs can be assessed using a cost-benefit analysis, on a project-by-project basis.
   ii. The benefit is weighted higher the more that correctness matters, e.g. where human lives depend on program correctness.

(c) Correctness proving is too difficult. **Partial Refutation:**
   i. Some non-trivial S/W products have successfully been proven correct.
   ii. There exists theorem-proving software to save manual work in some situations.
   iii. However proving program correctness in general is an **undecidable** problem, so no theorem-prover can handle every possible situation.

Morals:

1. Correctness proving is a useful tool, when human lives are at stake, or when the cost-benefit analysis justifies doing it for other reasons.
2. However correctness proving alone is not enough. Testing is still a crucial need for a S/W product.
3. Languages like Java and C++ support variations of an `assert` statement, which permits a programmer to embed assertions directly into the code. A switch then controls whether assertion checking is enabled (slower) or not (faster) at run time.
4. **Model checking** is a new technology that may eventually replace correctness proving. It is describe in Chapter 18 of the text, which unfortunately will be beyond the scope of CS 430.

13.2 Who Should Perform Execution-Based Testing?

1. Programmers should **not** have the ultimate responsibility to test their own code. **Reasons:**
   (a) Fundamental conflict of motivations
      i. Coding is **constructive**.

ii. Testing's goal (exposing faults) is **destructive**.
iii. Programmers feel protective of their own code, hence they have an incentive not to expose faults in the code.

(b) The programmer may have misunderstood the specification.
i. An SQA professional has a better chance to understand the specification correctly, and to test accordingly.

2. After the programmer completes and hands off the code artifact, SQA should perform **systematic testing**:

Definition 13.2.1. ***Systematic testing*** *is described by the following procedure:*
(a) Select test cases to exercise all parts of the specification.
(b) For each test case, determine its expected output ***before execution starts****.*
(c) Execute the program on each test case, and ***record the actual results****.*
(d) Compare the actual results to the expected results. ***Document all differences****.*
(e) Correct faults (either in the specification or in the code or possibly both) which explain each difference, and repeat the execution.
(f) Archive all test results electronically, for purposes of regression testing during future projects and post-delivery maintenance.

(a) Ambiguity about the term **desk checking** in the text:
i. first mention (description of testing workflow): Here desk checking meant the testing that a programmer does during development. This is the meaning with which I was already familiar from my time in industry.
ii. second mention (description of who should perform execution-based testing): Here desk checking means the checking of the design artifact that the programmer does before starting to code.

3. As outlined earlier, the SQA group must have managerial independence from the development team.

13.3 When Testing Stops

1. Only when the S/W product is decommissioned and removed from service, should testing stop.

Questions from the Class:

1. Will we have to write correctness proofs like the one in the notes for this lecture?
   Answer: No.
   (a) I will include a small example of the Hoare Triple technique for the next assignment, which can be done "with bare hands" (i.e. you will not need the machinery that the example uses).
   (b) There will be no correctness proving on the Final Exam.

14 Lecture 14 - The OO Paradigm - Cohesion and Coupling

Outline

1. What is a Module?
2. Cohesion (§7.2)
3. Coupling (§7.3)
4. Cohesion / Coupling Example

14.1 What is a Module?

Remarks:

1. Chapter 7 does a poor job of explaining the object-oriented paradigm. So we will use more industry-standard definitions than the text does. Use our definitions instead of the text's, where they disagree.
2. All of Chapter 7 is "white-box", not "black-box". We cannot assess cohesion / coupling unless we can see all of the code.
3. However, soon (i.e. after discussing **encapsulation** (Definition 15.1.1) and **information hiding** (Definition 16.2.1), we will see that it is a best practice **not** to expose all of the code to the outside world.

Key Quotation: When a large S/W product consists of a single monolithic block of code, maintenance is a nightmare.
A working definition for us:

Definition 14.1.1. *A **module** is a lexically contiguous sequence of program statements, bounded by boundary elements and having an aggregate identifier.*

Remarks About Definition 14.1.1:

1. Every function/procedure of the classical paradigm is a module.

2. In the OO paradigm, every class and every method within a class is a module.
   (a) The main idea of OO is to keep data, and operations on that data, together.
   (b) We need to be clear about the difference between the program statements that define the properties (a.k.a. attributes) of a class, and some **instantiation** of that class. Only an instantiation of a class can actually contain data.

Definition 14.1.2. ***C/SD*** *is an acronym for* ***composite/structured design****.*

Remarks:
1. The aim of C/SD is to apply common sense to make S/W product designs "make sense". (E.g. see Figures 7.1 to 7.3 in the text for designs that do, and do not, make sense.)
2. C/SD done well achieves **separation of concerns** (Definition 9.4.1).

14.2 Cohesion (§7.2)

Definition 14.2.1. ***Cohesion*** *of a module is the degree of interaction within that module.*

Remarks:
1. The text defines many levels of cohesion. Do **not** memorize these!
2. For us, it will be enough to distinguish between **high** and **low** cohesion. high = good; low = bad.

14.3 Coupling (§7.3)

Definition 14.3.1. ***Coupling*** *of a pair of modules is the degree of interaction between the two modules.*

Remarks:
1. The text defines many levels of coupling. Do **not** memorize these!
2. For us, it will be enough to distinguish between **loose** and **tight** coupling. loose = good; tight = bad.

14.4 Cohesion & Coupling Example

Remarks on Assessing Cohesion and Coupling:

1. Suppose that we are given two pairs of modules and it is our job to assess which pair's modules have
   (a) high versus low cohesion, and
   (b) loose versus tight coupling.
2. Because we have dispensed with the detailed levels of cohesion and coupling from the textbook, therefore making both judgments is **relative**, **not absolute**.
3. We can decide
   (a) which pair's modules have higher cohesion than the modules of the other pair, and
   (b) which pair's modules have looser coupling than the modules of the other pair.
4. In past offerings of CS 430, cohesion and coupling has caused some confusion. Keep our definitions, plus the above remarks in mind, and work out your comparisons carefully.

Cohesion / Coupling Example Refer to the Examples document.

Results:

1. low cohesion, tight coupling (bad)
2. high cohesion, loose coupling (good)

Why Coupling is Important

1. Tight coupling means a higher probability of regression faults.
2. Suppose modules `p` and `q` are tightly coupled.
3. Then it is likely that making a change to `p` requires a change to `q`.
4. Making the change to `q` adds time, and hence cost, to the project (which would not be required with looser coupling).
5. Not making the change to `q` likely causes a fault later on.
6. The stronger the coupling with some other module, the more fault-prone a module is.
7. This in turn makes the module the more difficult and costly to maintain.
8. As mentioned above, our goal is high cohesion and loose coupling. The rest of Ch7 is about refining the techniques to achieve this goal. Ch14 of the text goes into more detail; unfortunately this will be beyond the scope of CS 430.
9. Also note that separation of concerns (in general terms) means high cohesion and loose coupling (in OO terms).

15 Lecture 15 - The OO Paradigm - Encapsulation and Abstraction

Outline

1. Encapsulation (§7.4)
   (a) Encapsulation and Development (§7.4.1)
   (b) Encapsulation and Maintenance (§7.4.2)

15.1 Encapsulation (§7.4)

1. We briefly studied modules having high cohesion and loose coupling from §7.2 and §7.3.
2. These are key ingredients in understanding the OO paradigm.
3. We introduce another key ingredient now.

Definition 15.1.1. *In OO programming,* ***encapsulation*** *refers to one of two related but distinct notions, and sometimes to the combination thereof:*

1. *A language construct for restricting direct access to some parts of a module.*
2. *A language construct for* ***bundling*** *data with the methods (or other functions) operating on that data.*

We will adopt Definition # 2.

Remarks:

1. Why we adopt Definition # 2: In many OO languages, hiding of components is not automatic or can be overridden; thus, **information hiding** (Definition 16.2.1) is defined as a separate notion.
2. Encapsulation plus **information hiding** (Definition 16.2.1) is used to hide the values of a structured data module, preventing unauthorized parties' direct access to them.
3. Publicly accessible methods are provided (so-called **getters** and **setters**) to access the values; other client modules call these methods to retrieve/modify the values within the module.
4. Hiding the internals of the module protects its integrity by preventing users from setting the internal data of the module into an invalid / inconsistent state.
5. A benefit of encapsulation is that it can reduce system complexity, and thus increase **reliability**, by allowing the developer to limit the inter-

dependencies between S/W components (i.e. this provides a technique for achieving **separation of concerns**).

6. The features of encapsulation are supported by using **classes** (Definition 16.3.1) in OO programming languages.
7. Encapsulation is not unique to OO programming. Implementations of **abstract data types** (Definition 16.1.1) offer a similar form of encapsulation.
8. See the Example (text pp 199-201) of refining a S/W product from an initial design having low cohesion into a better design having encapsulation.
9. In the first solution to the Cohesion/Coupling example (last lecture), we could have achieved high cohesion and loose coupling by simply copying all the needed code into both modules. But this would indicate a failure to **abstract** effectively (Definition 15.1.2). We would have duplicated code in the two modules.
10. **Moral:** Doing OO effectively requires doing a good job on **all** of its ingredients.

15.1.1 Encapsulation and Development (§7.4.1)

Remarks:

1. **Abstraction** is a way of simplifying things so that they become easier to understand. E.g. representing the motion of the objects in the solar system by abstracting planets to points.
2. Effective abstraction helps us to see how things which appear different at first glance are actually the same in all relevant ways.
3. In S/W development, abstraction lets us focus on **what** a module does and not on **how** the module does it.

Definition 15.1.2. ***Abstraction*** *is suppressing irrelevant details and accentuating relevant details.*

Definition 15.1.3. *A* ***data abstraction*** *is an abstraction done on data.*

Definition 15.1.4. *A* ***procedural abstraction*** *is an abstraction done on code.*

Remarks:

1. **Abstraction** is a means of achieving **stepwise refinement** (Definition 9.1.1).

2. As a recommendation to the programmer, the **abstraction principle** reads

   > Each significant piece of functionality in a program should be implemented **in just one place** in the source code. Where similar functions are carried out by distinct pieces of code, combine them into one, abstracting out the varying parts.

   In short, "Don't repeat yourself."
3. Effective abstraction guides us to good choices of what to encapsulate when we design and develop our S/W.
4. Abstraction and encapsulation are different, but go hand-in-hand in OO design and development.
5. Abstraction permits a designer to temporarily ignore the details of the levels **above** and **below** the level currently being worked on, both in terms of data and procedures. An example of a **data abstraction** (Definition 15.1.3) is:
   (a) A database designer focuses on designing a table, temporarily ignoring the details of
      i. the whole database (the level above), and
      ii. the other tables having foreign key relationships to the current table (the level below).

15.1.2 Encapsulation and Maintenance (§7.4.2)

Idea: Design a S/W product to encapsulate the parts that are most likely to change in the future. Doing this effectively will minimize the impact of inevitable changes, on the other components. N.B. There is no algorithm for deciding how to do this. Human intuition and experience are required.

1. Data structures tend not to change very frequently (but **data abstraction** helps if they do).
2. Business rules tend to change more frequently (and **procedural abstraction** helps when they do).

16 Lecture 16 - The OO Paradigm - Abstract Data Types, Information Hiding and Objects

Outline

1. Abstract Data Types (§7.5)
2. Information Hiding (§7.6)
3. Objects (§7.7)

16.1 Abstract Data Types (§7.5)

Definition 16.1.1. *An **abstract data type** is a mathematical model of*

1. *the data objects comprising a data type, and*
2. *the functions that operate on these data objects.*

Examples:

1. The **integers** are an ADT, defined as the values $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, and by the operations of $+$, $-$, $*$, and sometimes $/$, etc., which behave according to the familiar rules of arithmetic (e.g. associativity, commutativity, distributive laws, no dividing by 0, etc).
   Typically integers are represented in a data structure as binary numbers, but there are many representations.
   The user is abstracted from the concrete choice of representation, and can simply use the data objects and operations according to the familiar rules.
2. a **stack** (i.e. a last-in, first-out data structure).

Remarks:

1. An **abstract data type (ADT)** need not be an arithmetic object itself; however each of its operation must be defined by some **algorithm**.
2. In CS, an **abstract data type (ADT)** is a mathematical model, where a **data type** is defined by its behaviour ("what it does", not "how it does it") from the point of view of a user (not an implementer), specifically:
   (a) possible values,
   (b) possible operations on data of this type, and
   (c) the behaviour of these operations.

3. This contrasts with **data structures**, which are concrete representations of data, from the point of view of an implementer, not a user.
4. Using abstract data types supports abstraction of both kinds, data and procedural.
5. Hence abstract data types are desirable from the viewpoints of both development and maintenance.

16.2 Information Hiding (§7.6)

This is the last key ingredient in understanding the OO paradigm.

Definition 16.2.1. ***Information hiding*** *means hiding the implementation details of a module (data + code) from the outside world.*

How Information Hiding is Useful at Design Time:

1. Make a list of implementation decisions which are likely to change in the future.
2. Design the resulting modules such that these implementation details are **hidden** from other modules.
3. This practice protects other parts of the software product from the impact of extensive changes if the implementation decisions are changed.

Remarks:

1. A module affords this protection by
   (a) encapsulating the data/operations to be hidden together,
   (b) hiding the details using a language construct like `private`, and
   (c) providing a stable **interface**.
2. A **class** (Definition 16.3.1) may be implemented
   (a) without information hiding (bad), or
   (b) with information hiding (good).

16.3 Objects (§7.7)

Remarks:

1. The text attempts to exhibit a "straight line" path from modules to objects. In my humble opinion this does not tell the OO story correctly.
2. All ingredients need to be (independently) done well for effective OO development, which will realize the benefits of:
   (a) fewer regression faults,
   (b) cheaper maintenance and

   (c) re-use.
3. **Reminder:** Use our definitions from the Lectures Notes instead of the definitions from the text, where there are any conflicts.

Definition 16.3.1. *A **class** is an **abstract data type** (Definition 16.1.1) that supports **inheritance** (Definition 16.3.2).*

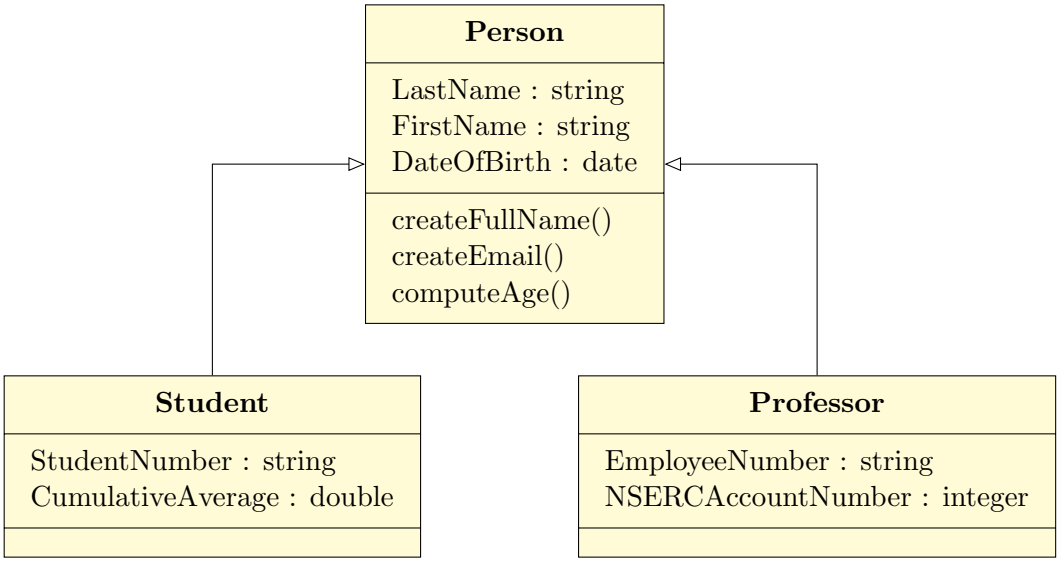
Definition 16.3.2. ***Inheritance** allows a new data type to be defined as an extension of a previously defined type, rather than having to be defined from scratch.*

Examples (Remark: the text example, in which `Person` is the parent class of the `Parent` class, is needlessly confusing!)

1. Start with a `Person` class, having
   (a) **Properties (/ Attributes)**
      i. `LastName`,
      ii. `FirstName`,
      iii. `DateOfBirth`

   and
   (b) **Methods**
      i. `createFullName`,
      ii. `createEmail` and
      iii. `computeAge`.
2. Then define a `Student` class, having all the **Properties/Methods** of `Person`, plus
   (a) **Properties**
      i. `StudentNumber`
      ii. `CumulativeAverage` (in reality we would compute this from individual grades rather than storing it; we make it a property here for simplicity).
3. Then define a `Professor` class, having all the **Properties/Methods** of `Person`, plus
   (a) **Properties**
      i. `EmployeeNumber`
      ii. `NSERCAccountNumber`.
4. Then each of `Student`, `Professor`
   (a) **inherits** from `Person`,
   (b) **isA** `Person`, and
   (c) is a **specialization** of `Person`.

5. Here is a diagram of the relationships between these classes.



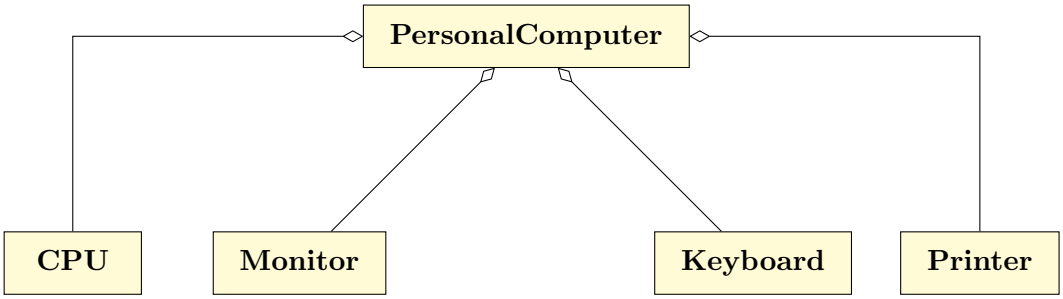
Definition 16.3.3. *An **object** is an instantiation of a class (Definition 16.3.1).*

Examples:

1. `CollinRoberts` could be an instantiation of the `Professor` class.

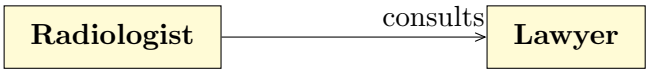
Definition 16.3.4. ***Aggregation/Composition*** *refers to the component classes of a larger class (i.e. grouping related classes creates a larger class).*

Example (Aggregation):



Definition 16.3.5. ***Association*** *refers to a relationship (of some kind) between two apparently unrelated classes.*

Example (Association):



The diagram indicates that `Radiologist` consults `Lawyer`.

17 Lecture 17 - The OO Paradigm - Inheritance, Polymorphism, and Dynamic Binding

Outline

1. Inheritance, Polymorphism, and Dynamic Binding (§7.8)
2. The Object-Oriented Paradigm (§7.9)

17.1 Inheritance, Polymorphism, and Dynamic Binding (§7.8)

Example:

1. Consider a `File` class, with an `Open` method.
2. An instantiation of a `File` might be stored on
   (a) hard disk,
   (b) flash drive or
   (c) tape,
   so the code inside the `Open` method must be different in each situation.
3. The `File` base class has derived classes
   (a) `HardDiskFile`,
   (b) `FlashDriveFile` and
   (c) `TapeFile`,
   each having an `Open` method specific to its medium.
4. The `File` class has a dummy `Open` method.
5. **Definition 17.1.1.** *At run time, the system decides which* `Open` *method to invoke. This is called* ***dynamic binding****.*
6.

Definition 17.1.2. *The* `Open` *method is called* ***polymorphic****, because it applies to different sub-classes, differently.*

7. Problems with Dynamic Binding/Polymorphism
   (a) We cannot determine at compile time which version of a polymorphic method will be called at run time. This can make failures hard to diagnose.
   (b) Similarly a S/W product that makes heavy use of polymorphism can be hard to understand and hence hard to maintain/enhance.

17.2 The Object-Oriented Paradigm (§7.9)

17.2.1 Summary of Reasons Why OO is Better than Classical

1. OO treats data and operations on that data together, with equal importance.
2. So a well-designed class does a good job of modelling some real-world entity.
3. A well-designed class also fosters **re-use**.
4. High cohesion + loose coupling → fewer regression faults.
5. Postdelivery maintenance is also improved.

17.2.2 The History that Led Us to the Current State of S/W Engineering.

1. In the 1960s and early 1970s, S/W Engineering was non-existent.
2. The **Code-And-Fix** model was the norm.
3. Hence the Classical model was most developers' first experience with S/W Engineering practices.
4. Adopting the Classical life-cycle model yielded major improvements in productivity and S/W quality at the time.
5. However as S/W products grew larger and more complex, the weaknesses of the Classical paradigm (which we have already discussed) became more pronounced, and the OO paradigm was proposed as a better alternative.

17.2.3 Problems With OO

Problem: There is a **learning curve** associated with adopting the OO paradigm for the first time. The first project done with OO takes longer than doing the

same project with the Classical paradigm. This is particularly pronounced if the project has a large GUI component. But after the initial project,

1. the re-use of classes in subsequent projects usually pays back the initial investment (again, this is more pronounced with a large GUI component) and
2. post-delivery maintenance costs are reduced.

17.2.4 Problems With inheritance

Definition 17.2.1. *Any change to the base class affects* ***all of its descendants****. This phenomenon is known as the* ***fragile base class problem****.*

1. In the best case, all descendants need to be recompiled after the base class is changed.
2. In the worst case, all descendants have to be re-coded then re-compiled. This is bad!

To mitigate this, meticulously design all classes, especially parent classes in an inheritance tree.

17.2.5 Cavalier use of inheritance

1. Unless explicitly prevented, every subclass inherits **all** the Properties/Methods of its parent. The reason to create a subclass is to add Properties/Methods. Hence objects lower in the inheritance tree can quickly become large, leading to storage problems.
2. Recommendation: change our philosophy from "use inheritance whenever possible" to "use inheritance whenever appropriate".
3. Also explicitly exclude Properties/Methods from being inherited, where this makes sense.

17.2.6 One Can Code Badly in Any Language

1. This is especially true of programming in an OO language. OO languages have constructs that add unnecessary complexity to the S/W product when they are misused.
2. We must endeavour to produce high-quality code when working with the OO paradigm.

17.2.7 OO Will Be Replaced In The Future

1. As mentioned earlier, the OO paradigm is certain to be superseded by some superior methodology in the future.
2. **Aspect Oriented Programming (AOP)** (covered in §18.1 in the text) is one possible candidate to replace the OO paradigm.

18 Lecture 18 - Reusability

Outline

1. Re-Use Concepts
2. Impediments to Re-Use
3. Types of Re-Use
   (a) Accidental (Opportunistic)
   (b) Deliberate (Systematic)
4. Objects and Re-Use
5. Re-Use During Design and Implementation
   (a) Library (toolkit)
   (b) Application Framework
   (c) Software Architecture
   (d) Component-Based Software Engineering

18.1 Re-Use Concepts

Importance of Re-usability

1. Advantages of Re-Use
   (a) Save time/resources during development/testing. "Don't re-invent the wheel".
   (b) Maintenance becomes cheaper.
   (c) Library subroutines are tested, (supposedly) well–documented
2. Pitfalls of Re-Use
   (a) Depending too heavily on re-use can make us averse to writing new code, even where this is needed.
   (b) Suppose that we need to extend/enhance an existing module before we can re-use it. This risks introducing regression faults for existing consumers of the module.
   (c) Old modules might not be as "good" (efficient, secure, having good style, etc.) as new modules.

(d) If we view the re-used module as black-box, then we may struggle to confirm that our S/W product will actually match the spec; if a failure occurs in the re-used module after deployment, then we may be slow to diagnose the cause.
(e) Compatibility Issues:
    i. S/W versions, or
    ii. the provided interface (the `Adapter` design pattern can sometimes solve this problem).
(f) Writing a module to handle multiple situations can make the module less efficient than if a separate module was written for each individual situation - but this would not be effective **abstraction**.
(g) If performance of the re-used module is not optimized, then all re-users will suffer a performance hit.
(h) Undetected faults get propagated.
(i) Documentation is often poor in practice.

3. Other Aspects
   (a) On average, 15% of any S/W product is written to serve a unique purpose.
   (b) In theory, remaining 85% could be standardized and reused.
   (c) In practice, only 40% reuse is achieved.
4. Re-use refers not only to code, but also to
   (a) documents (e.g. design, manuals, SPMP, etc.)
   (b) duration/cost estimates
   (c) test data
   (d) architecture
   (e) etc.

18.2 Impediments to Re-Use

1. Sometimes, what is a candidate for being re-used is not obvious.
   (a) Poor documentation (external, or internal, e.g. lack of comments in code) can contribute to this problem.
   (b) If we abstract effectively during analysis/design workflows, then what to re-use becomes clearer.
2. SQA test cases: too outdated to use (if business rules change)
3. **Ego:** unwillingness to use someone else's code ("Not Written Here" syndrome)
4. **Quality Concerns:** sometimes justified, as above.

5. Re-use can be expensive. It is costly to:
   (a) develop reusable modules, and
   (b) search the libraries and re-use the right module.
6. Legal issues with contract developers (possible intellectual property problems)
7. Commercial Of The Shelf (COTS): Developers do not provide the source code, so there is limited to no ability to modify and to re-use.
8. Etc.

18.3 Types of Re-Use

18.3.1 Accidental (Opportunistic)

Idea: Developer of a new S/W product realizes that a previously developed module can be re-used as a subroutine in the new S/W product (e.g. re-use previously developed `Mean` function).

18.3.2 Deliberate (Systematic)

Idea: S/W modules are specially designed and constructed to be used in multiple S/W products.

18.4 Objects and Re-Use

Key Fact: OO classes are the best type of module that we know about so far for fostering re-use.

18.5 Re-Use During Design and Implementation

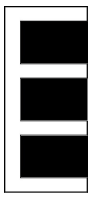
Remarks on Notation:

1. The diagrams for each type of re-use have
   (a) shaded areas for the parts that are re-used, and
   (b) whitespace for the parts that the re-user must supply.

We consider the following types of re-use.

18.5.1 Library (toolkit)

Assumes either the Classical or the OO paradigm.
Details:



1. **What is Re-Used:** There is a **library**, a set of related re-usable operations e.g.
   (a) A `Matrix` library contains many operations - +, *, `determinant`, `invert`, etc.
   (b) `GUI` library contains different GUI classes - `window`, `menu`, `radio button`, etc.

   The re-user calls modules from the library.
2. **What is New:** The re-user must
   (a) supply **control logic** of S/W product as a whole, and
   (b) call library routines at the right moment using the control logic
   (c) See Figure 8.2a in the text.

18.5.2 Application Framework

Assumes either the Classical or the OO paradigm.
Details:

1. **What is Re-Used:** Opposite to library approach: Control logic is re-used
2. **What is New:** The re-user must
   (a) design application-specific sub-routines fitting inside the control logic.
   (b) See Fig 8.2b in the text.
3. If the goal is to improve S/W development speed, then reusing a framework will be more effective than using libraries/toolkits WHY? It takes
   (a) more effort to design control logic, and
   (b) less effort to develop application-specific sub-routines, but

i. in my experience, Library re-use is much more common than Application Framework re-use.
   Reason: It is rare to find two different S/W products with identical control logic.

4. **Examples of Application Framework Re-Use:**
   (a) games
   (b) Automated Teller Machines (ATMs)
      i. Suppose you are managing a team to develop S/W for ATMs, deployed by several banks.
      ii. The control logic for an ATM deposit will be the same, regardless of the bank (note, we are over-simplifying a tiny bit here).
      iii. However the details of how to carry out a deposit will depend completely on the choice of bank.
      iv. A side comment here is that this would be an example of deliberate (systematic) re-use. We would design and build the control logic with the intent to re-use it at all of the banks.

18.5.3 Software Architecture

Remarks:

1. Software architecture encompasses a wide range of design issues, including
   (a) organization of components (logical and physical)
   (b) control structures
   (c) communication / synchronization issues
   (d) DB organization and access
   (e) performance
   (f) choice of design alternatives
2. Architecture can also be re-used.
3. A more detailed treatment of architecture will be beyond the scope of CS 430.

18.5.4 Component-Based Software Engineering

Goal: construct a standard library of re-usable components (i.e. for **Library** Re-Use). See §18.3 in the text if you want to read further.

19 Lecture 19 - Design Patterns

Outline

1. Design Patterns
   (a) Introduction
   (b) Adapter Design Pattern (§8.6.2)
   (c) Bridge Design Pattern (§8.6.3)
   (d) Iterator Design Pattern (§8.6.4)
   (e) Abstract Factory Design Pattern (§8.6.5)
   (f) Categories of Design Patterns (§8.7)
   (g) Strengths/Weaknesses of Design Patterns (§8.8)
2. Re-Use During Post-Delivery Maintenance

19.1 Design Patterns

19.1.1 Introduction

Unlike Library (Toolkit) and Application Framework from last lecture, Design patterns **assume the OO paradigm**.

Definition 19.1.1. *A **design pattern** is a solution to a general design problem, in the form of a set of interacting classes that have to be customized to create a specific design.*

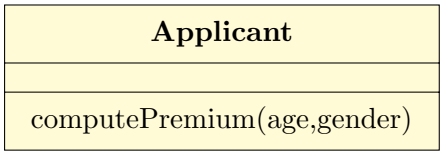
1. **What is Re-Used:** relationships among classes (usually expressed as a class diagram)
2. **What is New:** details within each class (usually a new class diagram, with the generic classes from the previous diagram replaced by classes tailored to the specific problem to be solved)

19.1.2 Adapter Design Pattern (§8.6.2)

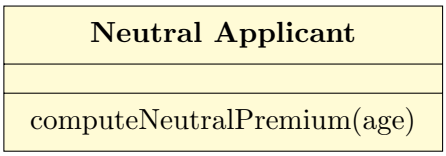
Motivation: FLIC Example (§8.6.1)

1. Until recently, premiums at Flintstock Life Insurance Company (FLIC) depended on both the age and the gender of the applicant for coverage.
2. FLIC has recently decided that some policies will now be gender-neutral. That is, the premiums for those policies will depend solely on the age of the applicant.

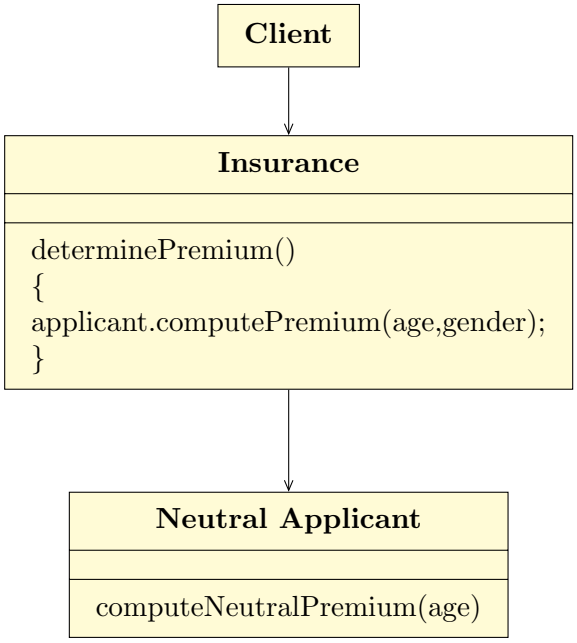
3. The old computation of premiums used this class:



4. The new computation of premiums will use this class:

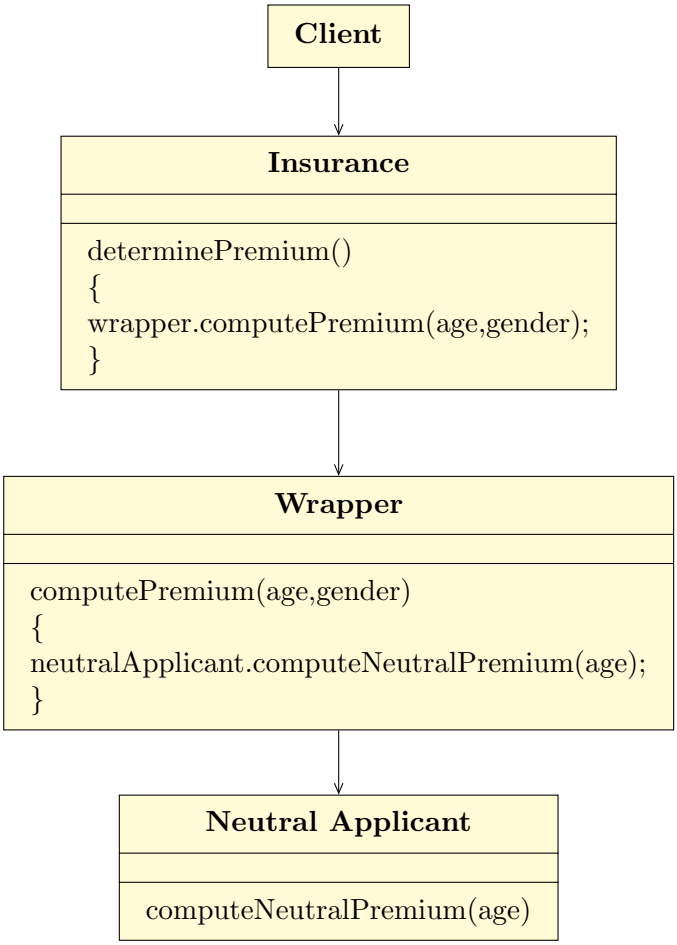


5. However there has not been enough time to change the entire system. The situation is displayed in the following figure (Fig 8.4 in the text).



Notation: ⟶ for "References".

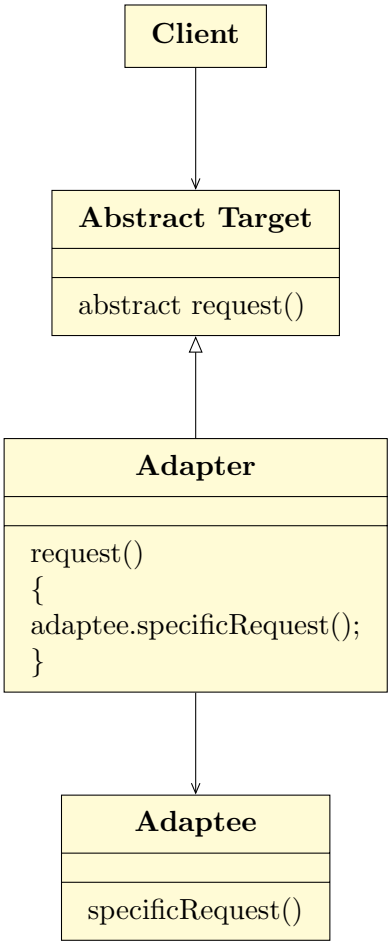
6. Note the three interface problems with the bottom reference in the above diagram:
   (a) `Insurance` calls the `Applicant` class instead of the `NeutralApplicant` class.
   (b) `Insurance` calls the `computePremium` method instead of the `computeNeutralPremium` method.
   (c) The parameters passed are `age` and `gender`, instead of `age` alone.
7. To solve these problems, we interpose the `Wrapper` class, as shown in this diagram (Figure 8.5 in the text):



Notation: $\longrightarrow$ for "References".

The Adapter Design Pattern

1. Generalizing the **Wrapper** construction above leads to the Adapter Design Pattern (Figure 8.6 in the text):



Notation: $\longrightarrow$ for "References".

Definition 19.1.2. *An **abstract class** is a class which cannot be instantiated, but which can be used as a base class for inheritance.*

Example: `Abstract Target` in the Adapter Design Pattern is an abstract class.

Definition 19.1.3. *An **abstract method** is a method which has an interface, but which does not have an implementation.*

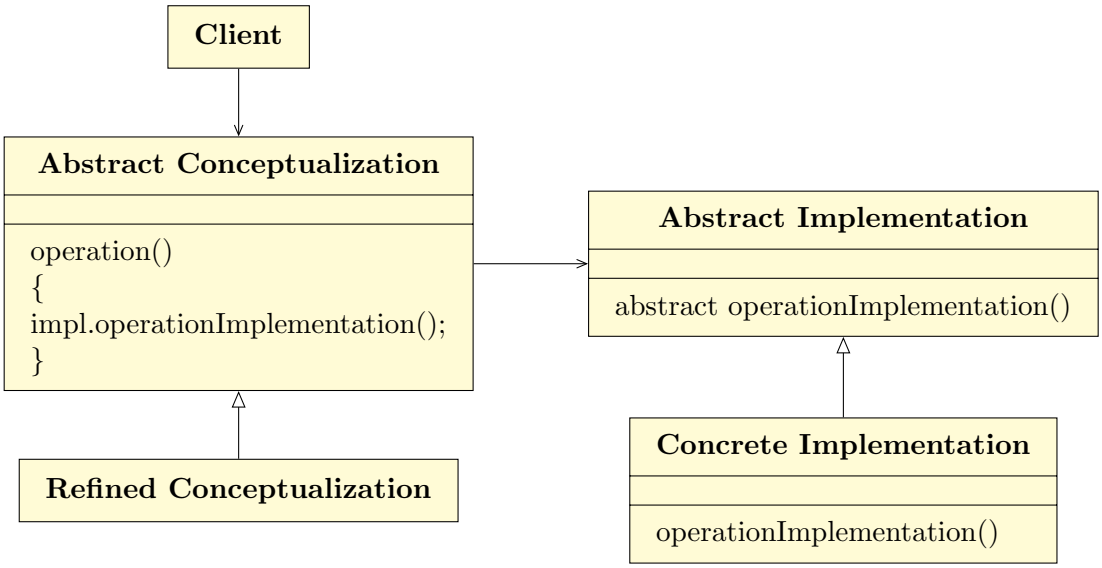
Example: In the Adapter Design Pattern, `Abstract Target` class, `request()` is an abstract method. Usually abstract methods live inside of abstract classes.

2. Abstract methods are implemented in subclasses of the abstract class.

3. The abstract `request` method from `Abstract Target` is implemented in the (concrete) subclass `Adapter`, to invoke the `specificRequest` method in `Adaptee`.
4. This solves the interfacing problems from earlier. This is the raison d'être for the Adpater design pattern.
5. But the pattern is more powerful than that. It provides a way for an object to permit access to its internal implementation in such a way that clients are not coupled to the structure of that internal implementation. In other words, it provides the benefits of **information hiding** (Definition 16.2.1) without having to actually hide the implementation details.

19.1.3 Bridge Design Pattern (§8.6.3)

1. **Prototype:** a **device driver**, e.g. a printer driver.
2. **Key Idea:** De-couple an abstraction from its implementation, so that the two can be changed independently of one another.
3. **Technique:** Construct a **bridge** which separates the part of the application which is not hardware-dependent from the part of the application which is.
4. Below is a class diagram for the Bridge Design Pattern. This is Figure 8.7 in the text.



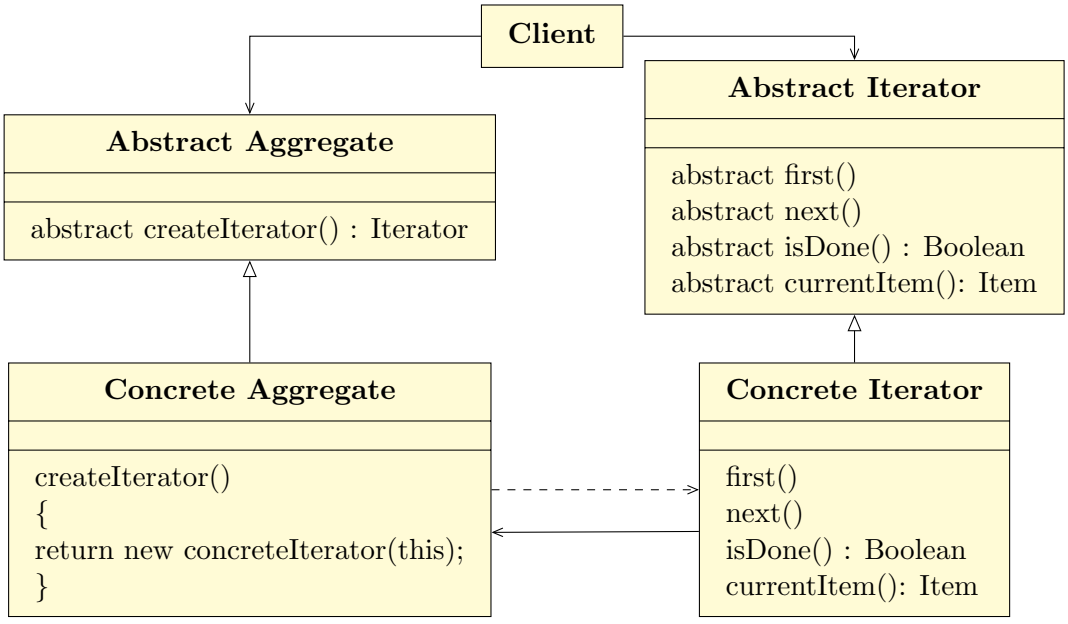
Notation: $\longrightarrow$ for "References".

19.1.4 Iterator Design Pattern (§8.6.4)

Goal: Iterate through each object inside an aggregate object, and "do something" with each one.
Prototype Example: Iterate through each element of a **linked list**, and process the element.
Remarks:

1. In the most general type of linked list, the elements need **not** all be of the same type. Hence one advantage of the Iterator Design Pattern lies in insulating ourselves from needing to know up front, what types of objects might be in the list.
2. Examples of aggregate objects:
   (a) linked list
   (b) hash table
   (c) database table (see below)
3. **Definition 19.1.4.** *An **iterator** is a programming construct that allows a programmer to traverse the elements of an aggregate object without exposing the implementation of the aggregate. Another name for an iterator is a **cursor**, especially in a database context.*
4. Two key ingredients:
   (a) **element traversal**: `first`, `next`, `isDone` in the example.
   (b) **element access**: `currentItem` in the example.

Below is a class diagram for the Iterator Design Pattern. This is Figure 8.9 in the text.



Notation: - - -> for "Creates" and ——> for "References".

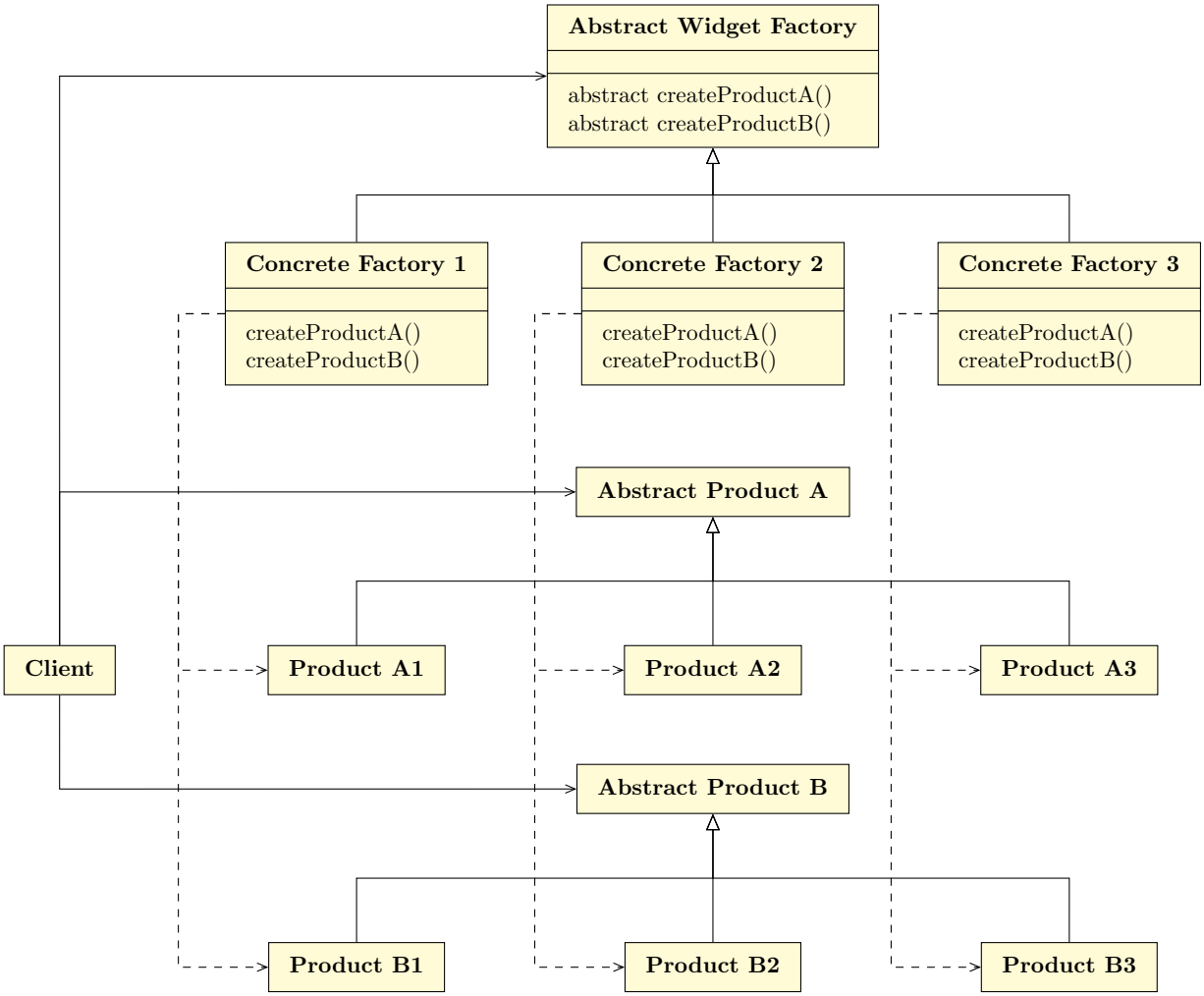
Remarks on the Diagram:

1. A `Client` deals with only `Abstract Aggregate` and `Abstract Iterator` (essentially an interface).
2. The `Client` object asks the `Abstract Aggregate` object to create an iterator for the `Concrete Aggregate` object, and then uses the returned `Concrete Iterator` object to traverse the elements of the aggregate.
   (a) `Abstract Aggregate` needs the abstract method `createIterator`, as a way of returning an `Iterator` to the `Client`.
   (b) The `Abstract Iterator` (interface) needs to define only the basic four abstract traversal methods. These four methods are implemented at the next level of abstraction, in `Concrete Iterator`.
3. Since implementation details of the elements are hidden from the `Iterator` itself, we can use an `Iterator` to process each element in an aggregate, independently of the implementation of the aggregate.
4. Hence using the pattern promotes **loose coupling**.
5. The pattern permits different traversal methods, since they are implemented only in `Concrete Iterator`.
6. There will be one pair of `Concrete Aggregate` and `Concrete Iterator` per type of concrete aggregate object.

7. Multiple different traversal methods may be present, if there are multiple types of list elements. Because of the common interface provided by `Abstract Iterator`, we do **not** need to know up front which types are possible.
8. Last lecture we pointed out that some up-front investment is required to position for future re-use.
9. We see this phenomenon in the example too: it takes more time to create the abstract classes (interfaces), then create the concrete classes, than it would take to write the concrete classes alone. But omitting the abstract classes forces the client to refer to all the concrete classes directly, frustrating our efforts to achieve re-use.

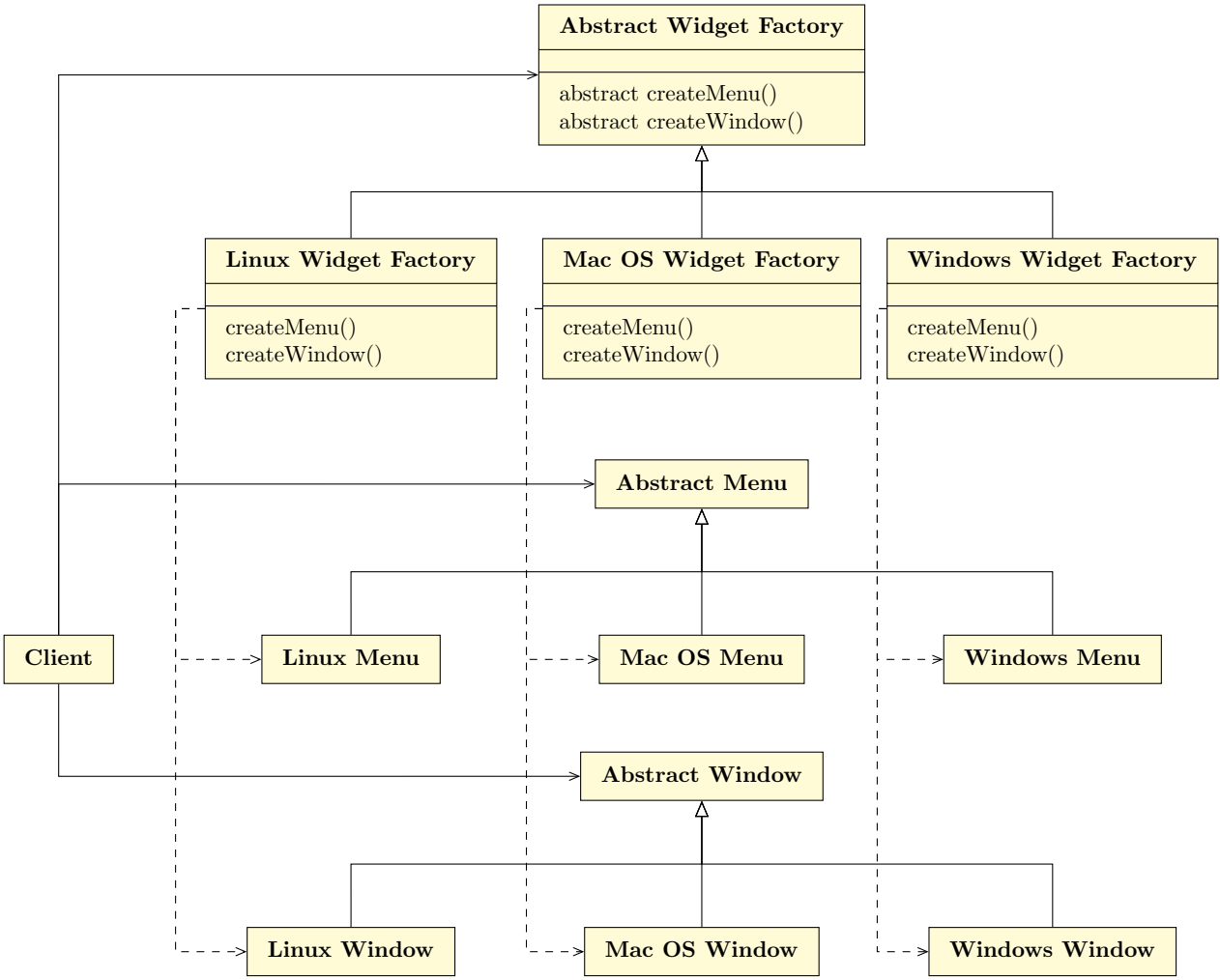
19.1.5 Abstract Factory Design Pattern (§8.6.5)

1. **Example:** an application with a GUI, which in particular has `Menus` and `Windows`.
2. **Key Idea:** The factory can be called by different developers working in different operating systems. Developers can re-use the set of classes developed by the widget factory, rather than developing those classes from scratch each time.
3. This is Fig 8.11 in the text, for the **Abstract Factory Design Pattern**.



Notation: - - -> for "Creates" and ——> for "References".

4. This is Fig 8.10 in the text, our instance of the **Abstract Factory Design Pattern**, in the case of a toolkit for a graphical user interface.



Notation: - - -> for "Creates" and ——> for "References".

5. In a real example, `Abstract Widget Factory` would contain many abstract methods; we include only two here, for simplicity.
6. Each of the three (concrete) subclasses of `Abstract Widget Factory` contains the methods to create widgets that run under a given operating system.
7. For example, invoking `createMenu` inside `Linux Widget Factory` creates a menu that will run under Linux.
8. To create a window, a `Client` need only invoke the `createWindow` method of `Abstract Widget Factory`, and polymorphism ensures that a window for the correct operating system is created.

9. The critical aspect of this pattern is that the three interfaces between `Client` and the widget factory (namely `Abstract Widget Factory`, `Abstract Menu` and `Abstract Window`) are all abstract classes. None of these classes is specific to any one operating system. Consequently, the design of Fig 8.10 has uncoupled the application program from the operating sytem.

19.1.6 Categories of Design Patterns (§8.7)

1. Creational, e.g. Abstract Factory
2. Structural, e.g. Adapter, Bridge
3. Behavioural, e.g. Iterator, Mediator

See Figure 8.12 in the text for the complete list of 23 documented by Gamma, Helm, Johnson and Vlissides.

19.1.7 Strengths/Weaknesses of Design Patterns (§8.8)

Strengths

1. promote re-use by solving a general design problem,
2. provide high-level documentation of the design, because patterns specify design abstractions,
3. may already have implementations written, and
4. make maintenance easier for programmers who are familiar with the patterns.

Weaknesses

1. lack a systematic way to determine when patterns should be applied,
2. often require multiple patterns together, which is complicated, and
3. are incompatible with the Classical paradigm.

19.2 Re-Use During Post-Delivery Maintenance

1. As we have seen throughout the course, an improvement in S/W methodology has a bigger payoff in maintenance than it does in development. This is true for the technique of re-use also:
   (a) Reusable components are well designed, thoroughly tested, well documented and independent. These are the features of low maintenance S/W.
   (b) Reusable components do not cause problems during maintenance.

20 Lecture 20 - Portability

Outline

1. Portability Concepts
2. Hardware Incompatibilities
3. Operating System Incompatibilities
4. Numerical System Incompatibilities
5. Compiler Incompatibilities
6. Is Portability Really Necessary?
7. Techniques for Achieving Portability
   (a) Portable Operating System Software
   (b) Portable Application Software
   (c) Portable Data
   (d) Object-Oriented Technologies (OOT)

20.1 Portability Concepts

Definition 20.1.1. *A program, P1, is* ***portable*** *if it is* ***significantly*** *cheaper to convert it to P2 (and run it on H/W H2, with OS O2 & compiler C2) than to re-code P2 from scratch.*

Remarks:

1. Portability does not mean porting the code only:
   (a) We must port documentation & manuals too.
   (b) If S/W is changed, then all docs must also change.

20.2 Hardware Incompatibilities

1. Character codes:
   (a) American Standard Code for Information Interchange (ASCII): 00000001
   (b) Extended Binary Coded Decimal Interchange Code (EBCDIC): 10000001
   (c) S/W developed on a platform with one encoding must be modified to work on a platform with the other encoding.

20.3 Operating System Incompatibilities

1. MAC OS versus Windows.

2. Similar problems on mainframe-scale systems.
3. JCL (**J**ob **C**ontrol **L**anguage, for specifying all the parameters needed to run mainfrmame batch jobs)
   (a) Each OS's JCL is slightly different.
4. Virtual Memory (i.e. augmenting physical memory by allocating some disk space as virtual memory)
   (a) If S/W is developed on an O/S that supports virtual memory, then there is no practical limit on the amount of memory available.
   (b) But if that same S/W is then ported to an O/S that does not support virtual memory, then there is a hard limit on the amount of memory available.

20.4 Numerical System Incompatibilities

1. Word size:
   (a) S/W developed on a 64-bit platform will not run on a 32-bit platform.

20.5 Compiler Incompatibilities

1. Different compiler versions can enforce different syntax rules.
   (a) Often newer compilers are more strict.

20.6 Is Portability Really Necessary?

Q: Does it make sense to spend time/resources to develop portable S/W?
A: Yes:

1. If your firm's business is selling software, then portability = higher revenue.
2. Even if not, i.e. if your organization builds software to support another primary business (e.g. selling insurance at SLF), keep in mind that good software lives 10-20 years or more, while hardware changes every 4-5 years. So portability saves money here too.

20.7 Techniques for Achieving Portability

20.7.1 Portable Operating System Software

1. UNIX O/S was constructed for maximum portability:

   (a) platform-independent (portable):
      i. 9000 LOC written in C
   (b) platform-dependent (must be re-written for each platform):
      i. 1000 LOC written in Assembly
      ii. 1000 LOC of C - device drivers
2. Lessons of UNIX
   (a) We should emulate the techniques used to design/build UNIX as much as possible.
   (b) When we have a choice of O/S, we should choose UNIX.

20.7.2 Portable Application Software

1. Although we may not always have control over which programming language we must use, whenever possible we should choose a high-level language (higher-level = more insulated from the hardware level).

20.7.3 Portable Data

Porting large amounts of data can be very problematic.

1. Flat files are the most portable data format. Problems:
   (a) Misunderstandings about file formats.
   (b) Self-documenting file formats (e.g. XML) solve problem 1a, but make files get big.

20.7.4 Object-Oriented Technologies (OOT)

1. Major promise of OOT:
   (a) final S/W product is portable & reusable

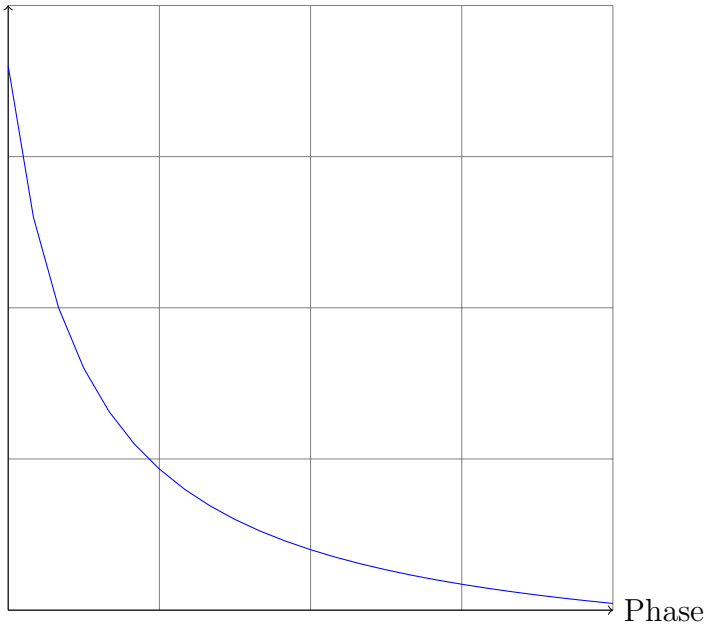
21 Lecture 21 - Planning and Estimation I - Function Points

Outline

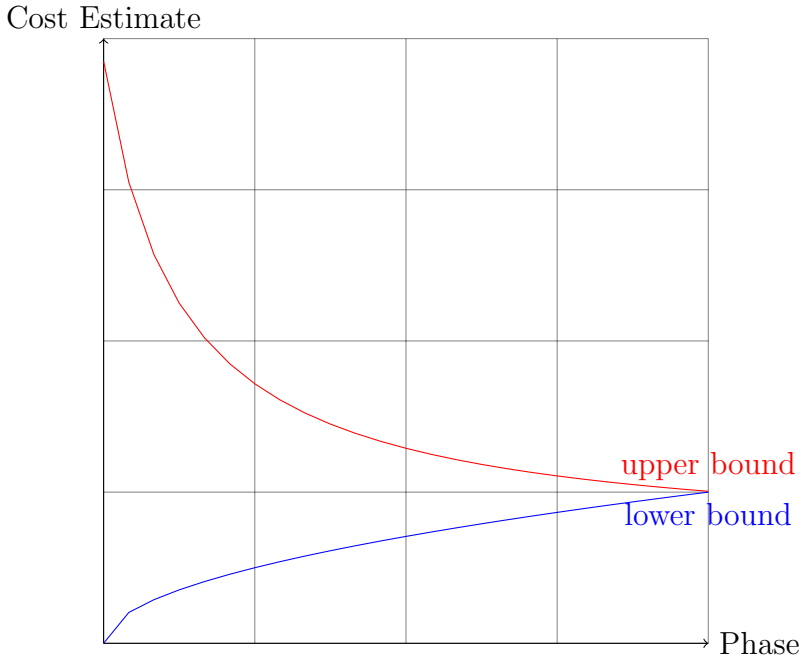
1. Planning and the Software Process
2. Estimating Duration and Cost
   (a) Metrics for the Size of a S/W Product

21.1 Planning and the Software Process

1. **When to estimate**
   (a) after requirements workflow - only an informal understanding of what is needed
      i. At this point, our ranges of estimates must be broad.
      ii. Figures 9.1 & 9.2 explain somewhat why this is true.
      iii. This is a summarized Figure 9.1 from the text. It displays a model for estimating the relative range of a cost estimate for each workflow.



      iv. This is a summarized Figure 9.2 from the text. It displays the range of cost estimates, in millions of dollars, for a software product that costs $1 million to build.



v. We provide a preliminary estimate here, so that the client can decide whether to proceed to analysis or not.

(b) after analysis workflow - a more detailed understanding of what is needed

i. For the rest of Ch 9, we assume that we are estimating at this point.

Remarks:

1. In practice, you may find yourself getting pressured by the client to reduce your preliminary estimates, to ensure that the project goes ahead. Common sense says that a client cannot dictate both the requirements and the costs to satisfy them. If the client thinks that the preliminary estimates are too high, then they can:
   (a) reduce the scope of the requirements, to reduce the estimated cost, or
   (b) increase the total budget.

   Giving in to pressure to reduce estimates at this point ALWAYS leads to problems later on.

21.2 Estimating Duration and Cost

1. **Estimating Cost:** All Costs of Development:
   (a) **internal**, i.e. the cost of our developers, e.g.
      i. salaries of project team members
      ii. costs of H/W and S/W
      iii. overhead costs
   (b) **external**, i.e. the price to the client, e.g.
      i. usually internal costs plus some mark-up
2. **Estimating Duration:** The client will need to know when to expect the S/W product to be delivered.
3. **Obstacles to Estimating Accurately:**
   (a) **human**
      i. variations in quality
      ii. turnover
      iii. varying levels of experience

21.2.1 Metrics for the Size of a S/W Product

1. **Function Points** provide a consistent basis for comparing the sizes of different S/W products.
2. Some larger projects were counted in terms of **function points (FP)** during my time at SLF.
3. Example like on pp273–275 in the text:
   (a) Compute the **unadjusted function points (UFP)** for a software product having the following function point counts in conjunction with Figure 9.3 in the text (reproduced here).

Figure 9.3 - Table of Function Point Values

	Level	of	Complexity
Component	Simple	Average	Complex
Input item	3	4	6
Output item	4	5	7
Inquiry	3	4	6
Master file	7	10	15
Interface	5	7	10

Function Point Counts to Use

	Level	of	Complexity
Component	Simple	Average	Complex
Input item	12	8	0
Output item	10	7	1
Inquiry	8	4	1
Master file	1	1	1
Interface	6	2	0

Solution:

$$\begin{aligned}
\text{UFP(Input item)} &= (12)(3) + (8)(4) + (0)(6) \\
&= 68 \\
\text{UFP(Output item)} &= (10)(4) + (7)(5) + (1)(7) \\
&= 82 \\
\text{UFP(Inquiry)} &= (8)(3) + (4)(4) + (1)(6) \\
&= 46 \\
\text{UFP(Master file)} &= (1)(7) + (1)(10) + (1)(15) \\
&= 32 \\
\text{UFP(Interface)} &= (6)(5) + (2)(7) + (0)(10) \\
&= 44, \text{ so that the total UFP is} \\
UFP &= 68 + 82 + 46 + 32 + 44 \\
&= 272.
\end{aligned}$$

(b) Compute the **technical complexity factor (TCF)** using the given counts for each factor in Figure 9.4 from the text (reproduced here).

Figure 9.4 (augmented) - Technical factors for function point computation

Factor	Name	Count to Use
1	Data communication	2
2	Distributed data processing	0
3	Performance criteria	3
4	Heavily utilized hardware	1
5	High transaction rates	3
6	Online data entry	5
7	End-user efficiency	5
8	Online updating	1
9	Complex computations	3
10	Reusability	3
11	Ease of installation	0
12	Ease of operation	5
13	Portability	3
14	Maintainability	5

Solution: Summing the counts in the above table gives us the **total degree of influence**:

$$\begin{aligned} DI &= 2+0+3+1+3+5+5+1+3+3+0+5+3+5 \\ &= 39, \end{aligned}$$

so that the corresponding **technical complexity factor (TCF)** is

$$\begin{aligned} TCF &= 0.65 + (0.01)DI \\ &= 0.65 + (0.01)(39) \\ &= 1.04. \end{aligned}$$

(c) Use the results of parts a) and b) to compute the **function points (FP)** for the given software product.
Solution:

$$\begin{aligned} FP &= (UFP)(TCF) \\ &= (272)(1.04) \\ &= 282.88, \end{aligned}$$

so that we measure this software product at $\boxed{283FP}$. (Only whole numbers make sense here; we **always round up** to be conservative.)

Remarks About Function Points

1. Observe that nowhere in the computation of UFP or FP did we ask
   (a) in what language is this software product written? or
   (b) how many lines of code does this software product have?

   FP are designed to be independent of these factors. FP compare sizes of different software products, regardless of their implementations.

22 Lecture 22 - Planning and Estimation II - Intermediate COCOMO

Outline

1. Estimating Duration and Cost
   (a) Techniques for Cost Estimation
   (b) Intermediate COCOMO
   (c) COCOMO II
   (d) Tracking Duration and Cost Estimates

22.1 Estimating Duration and Cost

22.1.1 Techniques for Cost Estimation

Definition 22.1.1. ***KDSI*** *stands for* ***Thousand Delivered Source Instructions*** *(i.e. 1000s of Lines of Code).*

Remarks:

1. There is **no perfect technique** for estimating the cost/duration of a S/W project.
2. Some factors to consider:
   (a) skill levels of project personnel (including familiarity with the S/W product)
   (b) complexity of project
   (c) project deadlines
   (d) target hardware
   (e) availability of CASE tools
3. **Techniques of Estimation**
   (a) Expert Judging by Analogy
      i. experts using history of similar past projects.

(b) Bottom-Up Approach
   i. analogous to **divide and conquer** (Definition 9.3.1), and
   ii. most common in my SLF experience.

(c) Algorithmic Cost Estimation Models (e.g. COCOMO)
   i. Compute the **size** of the S/W product, using **function points**, or some other method.
   ii. Use the size of the S/W product from 3(c)i to estimate cost & duration of the project to build it.

22.1.2 Intermediate COCOMO (COnstructive COst MOdel)

1. COCOMO comprises three models (highest level → lowest level):
   (a) macroestimation
   (b) intermediate (what we use here)
   (c) microestimation
2. Two stages in Intermediate COCOMO: estimate each of
   (a) **nominal effort**
   (b) **estimated effort**
3. Example like on pp278-280 in the text:
   (a) Compute the **nominal effort** for a software product having
      i. **organic** development mode (with multiplier 3.2 as in the text),
      ii. **exponent** 1.07, (different from the 1.05 used in the text) and
      iii. 12,000 LOC (i.e. 12 KDSI).

   Solution:

$$\begin{aligned} \text{nominal effort} &= 3.2(KDSI)^{1.07}\text{person-months} \\ &= 3.2(12)^{1.07} \\ &\approx 45.69555028, \end{aligned}$$

   and so we state the nominal effort as 46 person-months. (Only whole numbers make sense here; we **always round up** to be conservative.)

   (b) Use part a) to compute the **estimated effort**, using the given multipliers for each cost driver in Figure 9.6 from the text (reproduced here).

Figure 9.6 - Intermediate COCOMO software development effort multipliers

Cost Drivers	Rating: Very Low	Low	Nominal	High	Very High	Extra High
Product Attributes						
-Required software reliability	0.75	0.88	1.00	1.15	1.40	
-Database size		0.94	1.00	1.08	1.16	
-Product complexity	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
-Execution time constraint			1.00	1.11	1.30	1.66
-Main storage constraint			1.00	1.06	1.21	1.56
-Virtual machine volatility		0.87	1.00	1.15	1.30	
-Computer turnaround time		0.87	1.00	1.07	1.15	
Personnel Attributes						
-Analyst capabilities	1.46	1.19	1.00	0.86	0.71	
-Applications experience	1.29	1.13	1.00	0.91	0.82	
-Programmer capability	1.42	1.17	1.00	0.86	0.70	
-Virtual machine experience	1.21	1.10	1.00	0.90		
-Programming language experience	1.14	1.07	1.00	0.95		
Project Attributes						
-Use of modern programming practices	1.24	1.10	1.00	0.91	0.82	
-Use of software tools	1.24	1.10	1.00	0.91	0.83	
-Required development schedule	1.23	1.08	1.00	1.04	1.10	

multipliers to use

Cost Drivers	Rating to Use
Product Attributes	
-Required software reliability	Nominal
-Database size	Low
-Product complexity	Low
Computer Attributes	
-Execution time constraint	High
-Main storage constraint	Nominal
-Virtual machine volatility	Low
-Computer turnaround time	Nominal
Personnel Attributes	
-Analyst capabilities	Very High
-Applications experience	Very High
-Programmer capability	High
-Virtual machine experience	Low
-Programming language experience	High
Project Attributes	
-Use of modern programming practices	Nominal
-Use of software tools	Nominal
-Required development schedule	High

Solution: The effort multipliers for the given drivers are:

multipliers to use

Cost Drivers	Rating to Use	Multiplier to Use
Product Attributes		
-Required software reliability	Nominal	1.00
-Database size	Low	0.94
-Product complexity	Low	0.85
Computer Attributes		
-Execution time constraint	High	1.11
-Main storage constraint	Nominal	1.00
-Virtual machine volatility	Low	0.87
-Computer turnaround time	Nominal	1.00
Personnel Attributes		
-Analyst capabilities	Very High	0.71
-Applications experience	Very High	0.82
-Programmer capability	High	0.86
-Virtual machine experience	Low	1.10
-Programming language experience	High	0.95
Project Attributes		
-Use of modern programming practices	Nominal	1.00
-Use of software tools	Nominal	1.00
-Required development schedule	High	1.04

Using the given effort multipliers gives

$$\begin{aligned} & (1.00)(0.94)(0.85) \\ & (1.11)(1.00)(0.87)(1.00) \\ & (0.71)(0.82)(0.86)(1.10)(0.95) \\ & (1.00)(1.00)(1.04)46 \\ \approx\ & 19.31377308, \end{aligned}$$

and so we state the estimated effort as 20 person-months. (Only whole numbers make sense here; we **always round up** to be conservative.)

22.1.3 COCOMO II

1. COCOMO was introduced in 1981 (before OO was widely accepted; most systems were mainframe-based; classical paradigm was prevalent), and it became less reliable as time went on.

2. COCOMO II was a major revision to address these weaknesses.
   (a) COCOMO is all based on LOC (equivalently KDSI)
   (b) 3 applications of COCOMO II:
      i. application composition model
      ii. early design model
      iii. post architecture model
   (c) Where COCOMO outputs a single estimate, COCOMO II outputs a range of estimates for each model.
   (d) When I have taught CS 430 in the past, I have made a note to myself to present COCOMO II instead of Intermediate COCOMO, because we make the case throughout the course that we should adopt the OO paradigm.
   (e) However I found that doing this was not practical. I have posted a .pdf detailing COCOMO II on LEARN. Please peruse it at your leisure.
   (f) You may also find the following web pages about COCOMO II interesting:
      i. Overview: `http://sunset.usc.edu/csse/research/cocomoii/cocomo_main.html`
      ii. Calculator: `http://csse.usc.edu/tools/COCOMOII.php`
3. We don't have time to go into the details of COCOMO II in CS 430. See the text for references for additional reading if interested.

22.1.4 Tracking Duration and Cost Estimates

Key Ideas:

1. It is extremely rare for a S/W project to be completed ahead of schedule and under budget. Deviations from estimates usually make the project late and over budget.
2. Hence it is critical to detect deviations from our estimates ASAP, so that we can take **immediate** corrective action.

23 Lecture 23 - Planning and Estimation III - Project Management

Outline

1. Components of a SPMP

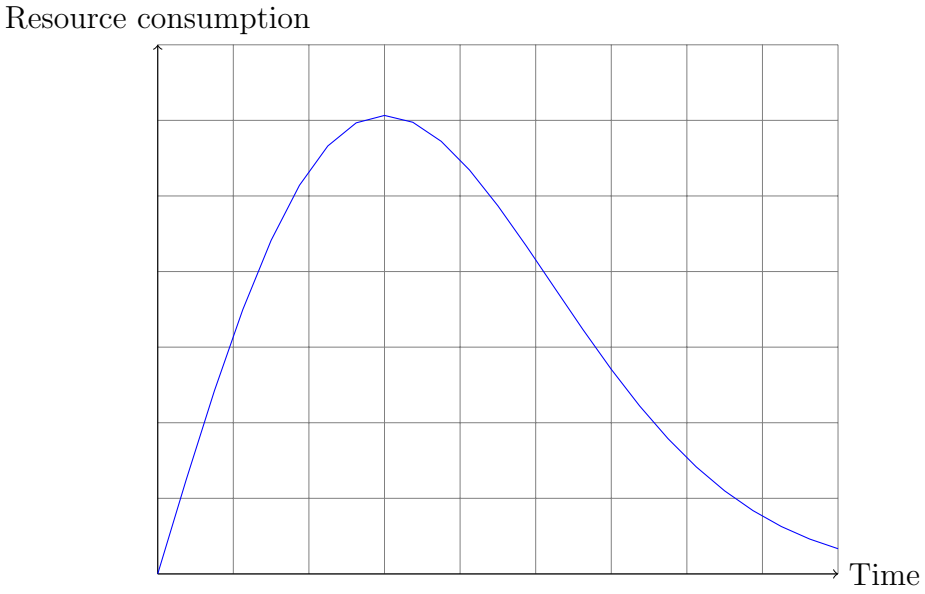
2. SPMP Framework
3. IEEE SPMP
4. Planning Testing
5. Planning OO Projects
6. Training Requirements
7. Documentation Standards
8. CASE Tools for Planning and Estimating
9. Testing the SPMP

23.1 Components of a SPMP

1. Three main components:
   (a) the work to be done
      i. **project functions** continue throughout the project, not related to any workflow (e.g. project management).
      ii. **activities/tasks** are related to a particular workflow.
         A. **Activities:** major units of work.
         B. **Tasks:** minor units of work.
   (b) the resources with which to do the work, e.g.
      i. people
      ii. hardware
      iii. software

   Include the timing of when those resources should be consumed.

   This is a summarized Figure 9.8 from the text. It is a Raleigh curve, showing typical resource consumption with respect to time.



While this is cute, it will **not** appear on the final exam. In my experience, I have never seen the Raleigh distribution used in reality.

(c) money to pay for it all

i. Detail the money to be spent, and when it will be spent.

23.2 SPMP Framework

1. SPMPs come in many forms. Each organization has a template that it prefers to use (level of detail depends on the organization's size and culture).
2. §9.5 of the text gives full details of the IEEE version, which could be used in the rare case where an organization needed to create its own template.
3. This plan covers projects of all sizes, so some of its pieces do not apply to smaller projects.

23.3 IEEE SPMP

The following template would be appropriate for a large project. For a small or medium-sized project, some parts could be omitted to make it suitable to the project.

1. **Overview.**
   (a) **Project summary.**
      i. **Purpose, scope and objectives.** A brief description is given of the purpose and scope of the software product to be delivered, as well as project objectives. Business needs are included in this subsection.
      ii. **Assumptions and constraints.** Any assumptions underlying the project are stated here, together with constraints, such as the delivery date, budget, resources, and artifacts to be reused.
      iii. **Project deliverables.** All the items to be delivered to the client are listed here, together with the delivery dates.
      iv. **Schedule and budget summary.** The overall schedule is presented here, together with the overall budget.
   (b) **Evolution of the project management plan.** No plan can be cast in concrete. The project management plan, like any other plan, requires continual updating in the light of experience and change within both the client organization and the software development organization. In this section, the formal procedures and mechanisms for changing the plan are described, including the mechanism for placing the project management plan itself under configuration control.
2. **Reference materials.** All documents referenced in the project management plan are listed here.
3. **Definitions and acronyms.** This information ensures that the project management plan will be understood the same way by everyone.
4. **Project organization.**
   (a) **External interfaces.** No project is constructed in a vacuum. The project members have to interact with the client organization and other members of their own organization. In addition, subcontractors may be involved in a large project. Administrative and managerial boundaries between the project and these other entities must be laid down.
   (b) **Internal structure.** In this section, the structure of the development organization itself is described. For example, many software development organizations are divided into two types of groups: development groups that work on a single project and support groups that provide support functions, such as config-

uration management and quality assurance, on an organization-wide basis. Administrative and managerial boundaries between the project group and the support group also must be defined clearly.

(c) **Roles and responsibilities.** For each project function, such as quality assurance, and for each activity, such as product testing, the individual responsible must be identified.

5. **Managerial process plans.**
   (a) **Start-up plan.**
      i. **Estimation plan.** The techniques used to estimate project duration and cost are listed here, as well as the way these estimates are tracked and, if necessary, modified while the project is in progress.
      ii. **Staffing plan.** The numbers and types of personnel required are listed, together with the durations for which they are needed.
      iii. **Resource acquisition plan.** The way of acquiring the necessary resources, including hardware, software, service contracts, and administrative services, is given here.
      iv. **Project staff training plan.** All training needed for successful completion of the project is listed in this subsection.
   (b) **Work plan.**
      i. **Work activities.** In this subsection, the work activities are specified, down to the task level if appropriate.
      ii. **Schedule allocation.** In general, the work packages are interdependent and further dependent on external events. For example, the implementation workflow follows the design workflow and precedes product testing. In this subsection, the relevant dependencies are specified.
      iii. **Resource allocation.** The various resources previously listed are allocated to the appropriate project functions, activities, and tasks.
      iv. **Budget allocation.** In this subsection, the overall budget is broken down at the project function, activity, and task levels.
   (c) **Control plan.**
      i. **Requirements control plan.** As described in Part B of the text, while a software product is being developed, the requirements frequently change. The mechanisms used to monitor

and control the changes to the requirements are given in this section.

ii. **Schedule control plan.** In this subsection, mechanisms for measuring progress are listed, together with a description of the actions to be taken if actual progress lags behind planned progress.

iii. **Budget control plan.** It is important that spending should not exceed the budgeted amount. Control mechanisms for monitoring when actual cost exceeds budgeted cost, as well as the actions to be taken should this happen, are described in this subsection.

iv. **Quality control plan.** The ways in which quality is measured and controlled are described in this subsection.

v. **Reporting plan.** To monitor the requirements, schedule, budget, and quality, reporting mechanisms need to be in place. These mechanisms are described in this subsection.

vi. **Metrics collection plan.** As explained in text §5.5, it is not possible to manage the development process without measuring relevant metrics. The metrics to be collected are listed in this subsection.

(d) **Risk management plan.** Risks have to be identified, prioritized, mitigated, and tracked. All aspects of risk management are described in this section.

(e) **Project close-out plan.** The actions to be taken once the project is completed, including reassignment of staff and archiving of artifacts, are presented here.

6. **Technical process plans.**

(a) **Process model.** In this section, a detailed description is given of the life-cycle model to be used.

(b) **Methods, tools and techniques.** The development methodologies and programming languages to be used are described here.

(c) **Infrastructure plan.** Technical aspects of hardware and software are described in detail in this section. Items that should be covered include the computing systems (hardware, operating systems, network, and software) to be used for developing the software product, as well as the target computing systems on which the software product will be run and CASE tools to be employed.

(d) **Product acceptance plan.** To ensure that the completed soft-

ware product passes its acceptance test, acceptance criteria must be drawn up, the client must agree to the criteria in writing, and the developers must then ensure that these criteria are indeed met. The way that these three stages of the acceptance process will be carried out is described in this section.

7. **Supporting process plans.**
   (a) **Configuration management plan.** In this section, a detailed description is given of the means by which all artifacts are put under configuration management.
   (b) **Testing plan.** Testing, like all other aspects of software development, needs careful planning.
   (c) **Documentation plan.** A description of documentation of all kinds, whether or not to be delivered to the client at the end of the project, is included in this section.
   (d) **Quality assurance plan.** All aspects of quality assurance, including testing, standards, and reviews, are encompassed by this section.
   (e) **Reviews and audits plan.** Details as to how reviews are conducted are presented in this section.
   (f) **Problem resolution plan.** In the course of developing a software product, problems are all but certain to arise. For example, a design review may bring to light a critical fault in the analysis workflow that requires major changes to almost all the artifacts already completed. In this section, the way such problems are handled is described.
   (g) **Subcontractor management plan.** This section is applicable when subcontractors are to supply certain work products. The approach to selecting and managing subcontractors then appears here.
   (h) **Process improvement plan.** Process improvement strategies are included in this section.
8. **Additional plans.** For certain projects, additional components may need to appear in the plan. In terms of the IEEE framework, the appear at the end of the plan. Additional components may include security plans, safety plans, data conversion plans, installation plans, and the software project postdelivery maintenance plan.

23.4 Planning Testing

1. Include a detailed schedule for what testing must be done during each workflow. Potential problems if this is **not** done:
   (a) Capturing traceability between workflows (which is required to test effectively) may not be done correctly.
   (b) Missed opportunities to follow-up on later artifacts as suggested by unusually high numbers of faults in early artifacts of the project.
   (c) Black-box test cases should be selected at the end of the analysis workflow (while details are fresh in developers'/SQA members' minds). If not, then black box test cases may be hurriedly thrown together later on (less effective).
   (d) Etc.

23.5 Planning OO Projects

1. Planning / estimation tools (function points, intermediate COCOMO) work as well for OO as they do for classical, **assuming no re-use**.
2. To date there is little data on how re-use affects estimation.
3. We expect that in the long run, re-use will save effort, hence reduce estimates.
4. COCOMO II is better than COCOMO for OO, but it is much more complicated.

23.6 Training Requirements

1. Training requirements should be carefully considered for all staff, not just for the client. Reasons:
   (a) Developers may need training in project management (e.g. planning and estimating)
   (b) New development / testing techniques may necessitate training for all project staff.
   (c) New H/W may necessitate training for all operators.
   (d) Etc.
2. Training requirements must be documented in the SPMP.

23.7 Documentation Standards

1. Documentation is an integral part of any S/W project.
2. Hence it is crucial that standards be established (in the SPMP if nowhere else), understood and followed by all team members. Reasons:
   (a) fewer misunderstandings between team members
   (b) aids the SQA group
   (c) after initial training, no additional training will be needed when staff change teams internally,
   (d) etc.

23.8 CASE Tools for Planning and Estimating

1. There are many commercially available CASE tools for project management.
2. In all likelihood the organization that employs you will already have a project management suite in place for you to use.

23.9 Testing the SPMP

1. As pointed out earlier, it is crucial to neither underestimate nor overestimate the cost/duration of our S/W projects.
2. Hence SQA must test the SPMP before communicating any estimates to the client.
3. This must be non-execution based testing. Best technique: **inspection**.

24 Lecture 24 - Review and Wrap-Up

Outline

1. Course Review - Key Topics
2. Course Evaluations

24.1 Course Review - Key Topics

1. The Scope of Software Engineering
   (a) Historic / Economic Aspects

(b) Maintenance (esp Post-Delivery)
(c) Why There Is No Phase for
    i. Planning
    ii. Testing
    iii. Documentation
(d) The OO Paradigm

2. S/W Life-Cycle Models
(a) Change is Inevitable
(b) Iteration and Incrementation (which drives the remainder of the items in the list)
(c) Other Life-Cycle Models
    i. Code-And-Fix
        A. This was the prevailing model pre-Waterfall / Classical.
        B. Under this model there was no change management at all!
    ii. Waterfall / Classical
    iii. Rapid Prototyping
    iv. Open Source
    v. Agile Processes
    vi. Synchronize and Stabilize
    vii. Spiral
(d) No One Life-Cycle Model dictated by SW-CMM

3. The S/W Process
(a)

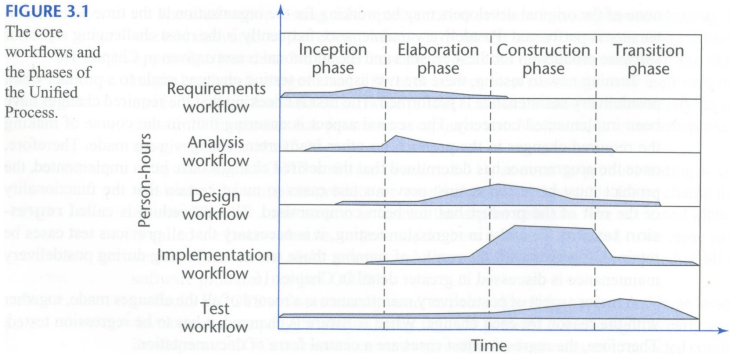


FIGURE 3.1
The core workflows and the phases of the Unified Process.

(b) Postdelivery Maintenance
(c) One- and Two-Dimensional Life-Cycle Models
(d) Capability Maturity Models (SW-CMM specifically)

4. Teams
(a) Democratic
(b) Classical Chief Programmer

(c) Modified Chief Programmer
(d) Teams for Life-Cycle Models
    i. Synchronize and Stabilize
    ii. Agile Processes
    iii. Open Source
(e) No One Team Organization dictated by P-CMM

5. The Tools of the Trade
(a) Stepwise Refinement
(b) Cost-Benefit Analysis
(c) Divide and Conquer
(d) Separation of Concerns
(e) S/W Metrics
(f) CASE tools
(g) Version/Configuration Control

6. Testing
(a) Quality Issues
    i. SQA
    ii. Managerial Independence
(b) Non-Execution-Based Testing (Reviews)
    i. Walkthroughs
    ii. Inspections
(c) Execution-Based Testing
    i. Best Practice: Determine expected results **before** you execute your first test.
(d) What to Test
    i. Utility
    ii. Reliability
    iii. Robustness
    iv. Performance
    v. Correctness
(e) Testing versus Correctness Proofs
    i. There will be **no** correctness-proving on the final exam.
    ii. When correctness-proving can be justified is fair game for the final exam.
(f) Who Should Perform Execution-Based Testing? Answer: SQA!

7. From Modules to Objects (N.B. Use our definitions from the Lecture Notes, NOT the text definitions here)
(a) Cohesion

(b) Coupling
(c) Encapsulation
(d) Abstract Data Types
(e) Information Hiding
(f) Objects
(g) Inheritance, Polymorphism, Dynamic Binding

8. Reusability and Portability
   (a) Reusability
      i. Impediments to Reusability
      ii. Objects and Reusability
      iii. Types of Re-use
         A. Library (toolkit)
         B. Application Framework
         C. Design Patterns
   (b) Portability
      i. Impediments to Portability
         A. Hardware Incompatibilities
         B. Operating System Incompatibilities
         C. Numerical System Incompatibilities
         D. Compiler Incompatibilities
      ii. Objects and Portability
9. Planning and Estimating
   (a) Estimation
      i. Metrics for Size of a S/W product - Function Points
      ii. Estimating Duration - Intermediate COCOMO
   (b) Project Management
      i. Testing
      ii. Training
      iii. Documentation
      iv. CASE Tools
      v. Testing the SPMP

24.2 Course Evaluations

- Please fill out your course evaluations at `http://perceptions.uwaterloo.ca`

- All the best on your final exams before ours!

Index