Lecture Notes on CS 422 Software Methodologies
By Anthony Hornof

Last updated February 20, 2024

These are Anthony Hornof's notes from:

Sommerville (2015) *Software Engineering*, 10th edition, Pearson.
Usability.gov, captured March 6, 2023.
Merriam Webster and New Oxford American dictionaries.
van Vliet. (2008). *Software Engineering: Principles and Practice.*

The notes were taken to (a) learn and organize an understanding of the material and (b) prepare lectures. The notes summarize some, but not all, of the required reading. Please do the reading to learn the required course material.

Some of the notes are copied directly from the textbooks.

Part 1 in "Contents at a Glance" provides a good overview of most of the assigned reading.

The content in boxes, such as this, is *not* from the book.

These notes are primarily organized around the chapters in Sommerville (2015).

Table of Contents

(Click on a chapter to go to that chapter.)

Chapter 1 - Introduction

Four themes that pervade all aspects of software engineering. (A. Hornof)

1. Use abstractions. Find ways to summarize detailed specifications of concepts and ideas, and use these brief summaries in place of the more complex ideas. It is imperative that all team members understand the abstractions that other team members are using.

2. Divide and conquer. Break a large problem into smaller pieces that can be solved one at a time. This permits you to focus your thinking on one problem at a time.

3. Propose and consider alternatives. Nearly every activity in software engineering involves some form of *design*, in which *design* is the process of proposing and evaluating alternative solutions to a problem.

4. Collaborate. Most of the processes and techniques developed and used in software engineering are ultimately aimed at helping groups of people combine their brainpower to solve problems together.

These notes are primarily copied from Sommerville (2015) *Software Engineering*, 10th edition.

Read the first page in class.

1.0 - Intro to Chapter 1.

Software engineering is essential for the functioning of government, society, and national and international businesses and institutions....

Software systems are abstract and intangible. They are not constrained by the properties of materials, nor are they governed by physical laws or by manufacturing processes....

There are many different types of software system, ranging from simple embedded systems to complex, worldwide information systems....

There are still many reports of software projects going wrong and of "software failures."....

1. Increasing system complexity....
2. Failure to use software engineering methods....

What is software?

Computer programs and associated documentation, libraries, support websites, and configuration data that are needed to make these programs useful.

What is software engineering?

Software engineering is an engineering discipline that is concerned with all aspects of software production from initial conception to operation and maintenance.

What is the difference between software engineering and computer science?

Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.

If you are writing a program for yourself, no one else will use it and you don't have to worry about writing program guides, documenting the program design, and so on. However, if you are writing software that other people will use and other engineers will change, then you usually have to provide additional information as well as the code of the program.

1.1.1 Software engineering

Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use. In this definition, there are two key phrases:

1. Engineering discipline. Engineers make things work. They apply theories, methods, and tools where these are appropriate...., and they must look for solutions within constraints.

2. All aspects of software production. Software engineering is not just concerned with the technical processes of software development. It also includes activities such as software project management....

Software engineers adopt systematic and organized approaches to their work, but engineering also requires a creative approach to solving problems.

[Skipping 1.1.2 through the end of the chapter. Students read on their own.]

[Note that 1.3 introduces the case studies used throughout the book.]

What is Software Engineering? (from Stuart Faulk)

Software engineering is the process of gaining and maintaining control over the products and processes of software development. There are two kinds of control:

- "Intellectual control" means that we make rational choices based on an understanding of the effects of those choices on the qualities of the product and process.

  Such as understanding the implications of using C++ versus python.

- "Managerial control" is related but different in focus: The purpose is to gain and maintain control of software development resources (money, time, personnel).

  Such as figuring out whether to try to hire more programmers or delay the delivery date.

In practice, both are necessary and inseparable. It would difficult to have managerial control if you do not first have intellectual control.

In contrast to computer science (the broad study of the basis and behavior of computing machines), software engineering is an inherently pragmatic discipline.

Chapter 2 - Software Processes

These notes are primarily copied from Sommerville (2015).

Overview and Concepts

A *software process* is a set of activities (or "phases") that leads to the creation of a software system.

A *software process model* (sometimes called a *software development lifecycle model*) is an abstracted representation of the major activities required to build a software system, and the order of those activities.

They are "models" because they are representations, paper-based simulations.

The models are generic, high-level, abstract descriptions that help to explain different approaches to software development.

Note how the models (the diagrams) in this lecture show fundamentally different ways to do a project, but do so at a high level of abstraction (with little detail).

Note how the diagrams (1) use abstractions, (2) divide and conquer, (3) make it easy to consider alternatives, and (4) support collaboration.

The Major Lifecycle Activities (the rounded boxes in the diagrams)

The models in the chapter typically include the following activities:

1. *Software specification*. The functionality of the software and constraints on its operation must be defined. This can be further broken out into:
    (a) Requirements specification.
    (b) Design specification.
2. *Software development*. The system is implemented to meet the specification.
3. *Software validation*. The system is tested to ensure that it does what the customer needs (validation) and that it works correctly (verification).
4. *Software evolution*. The system is modified to meet changing customer needs.

Three Examples of Software Lifecycle Models

Chapter 2 introduces three general software process models:

1. *The waterfall model*. The major activities are requirements specification (or definition), software design, implementation, testing, and evolution. Each activity is completed separately, in order.
2. *Incremental development*. Specification, development, and validation are all done in parallel (interleaved). The system is developed as a series of versions (increments), which each version adding more functionality.
3. *Integration and Configuration*. Reusable components or systems are combined (integrated) and set up (configured) to work together to meet system requirements.

Key: In each model from Sommerville, the rounded boxes show activities, the square boxes show deliverables, and the arrows show the order of the activities and the output of the deliverables.

Example Model #1 - The Waterfall Model

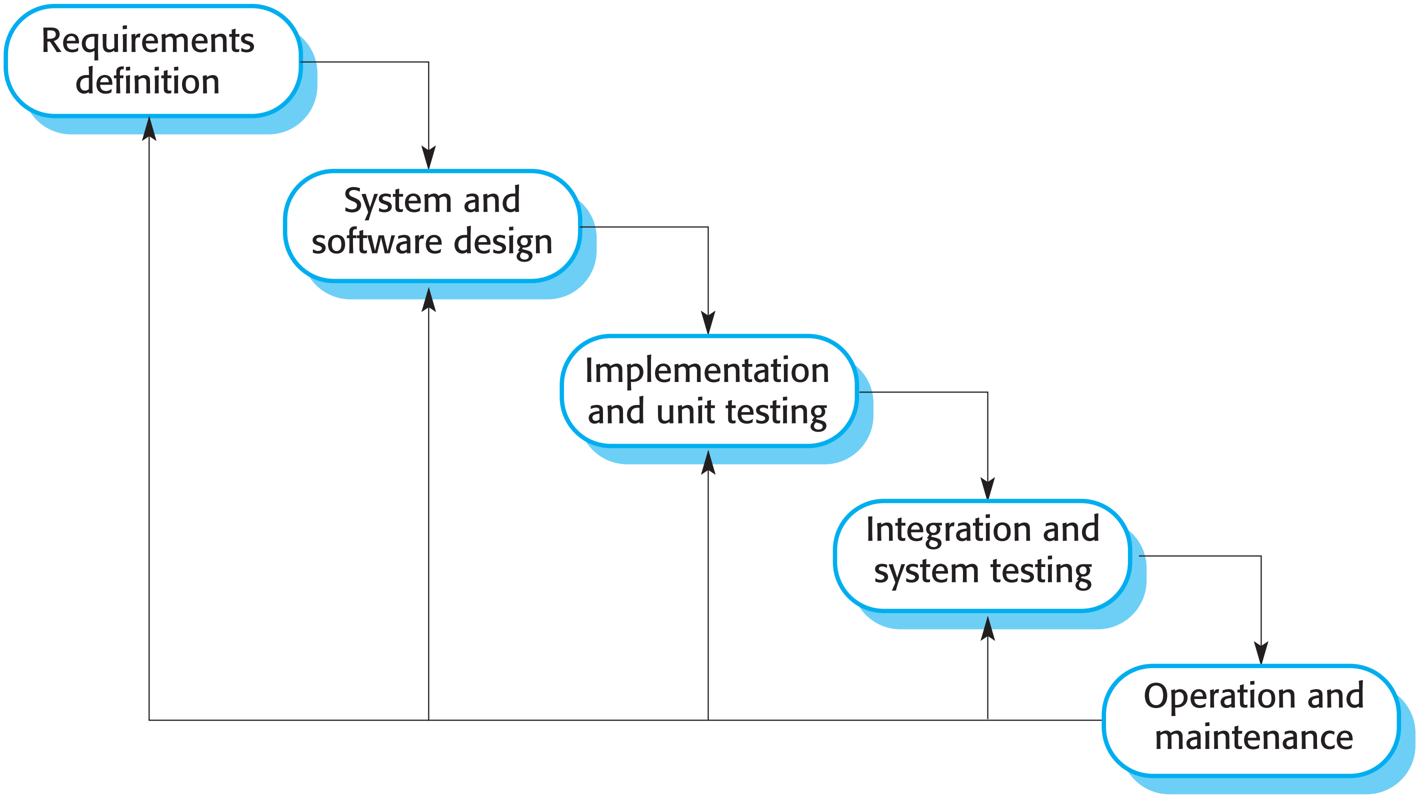

Requirements definition
System and software design
Implementation and unit testing
Integration and system testing
Operation and maintenance

Figure 2.1. The waterfall model.

In the waterfall model, activities are done in order. Based on the waterfall model shown in Figure 2.1, you only go back to a previous phase (or activity) during the maintenance phase.

Example Model #2 - Incremental Development Lifecycle Model

Incremental development is based on the idea of developing an initial implementation, getting feedback from users and others, and evolving the software through several versions until the required system has been developed (Figure 2.2). Specification, development, and validation activities are interleaved rather than separate, with rapid feedback across activities.

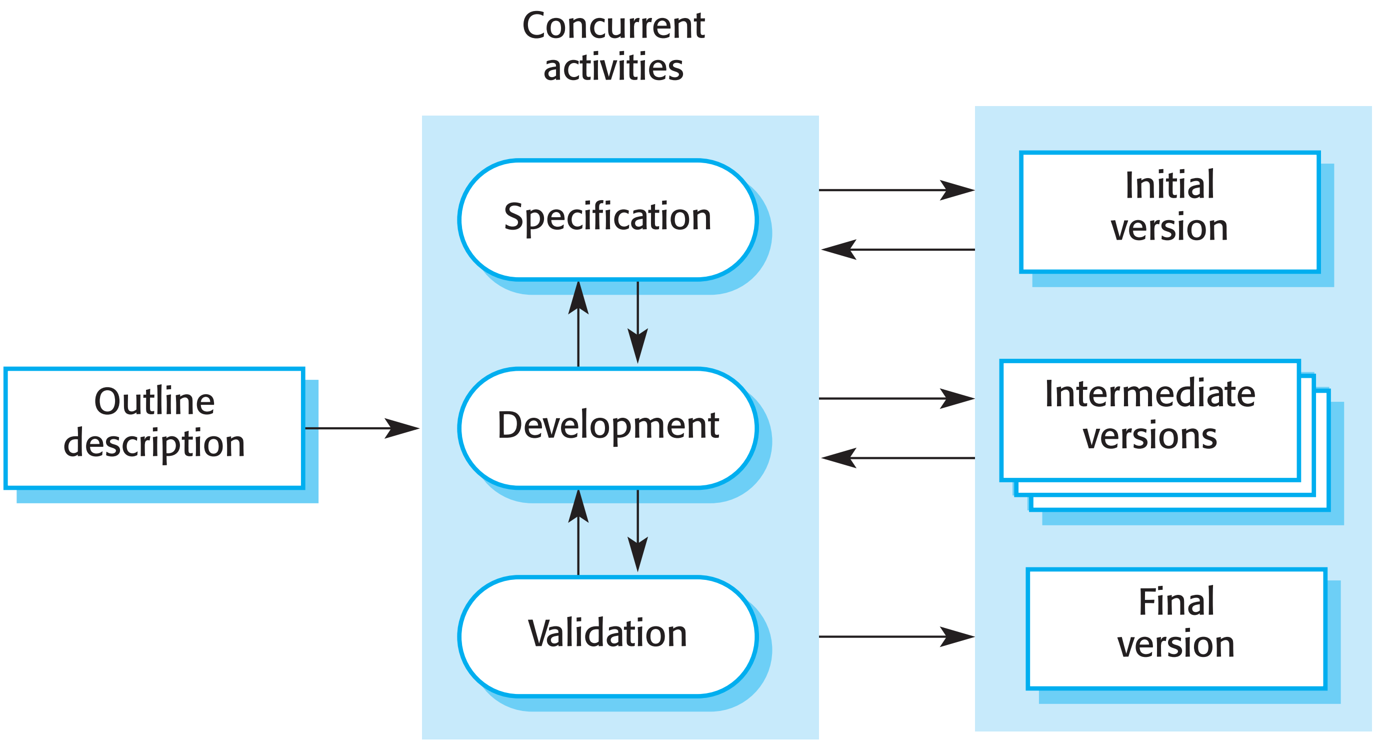


Figure 2.2. Incremental development lifecycle model.
All activities are interleaved (done in parallel, concurrently).
All of the deliverables on the right, including the final version, are produced by the concurrent execution of all activities.

Example Model #3 - Integration and Configuration

In the majority of software projects, there is some software reuse. The integration-and-configuration software-process-model breaks down the steps involved in looking for code, modules, or components; modifying and configuring them as needed; and integrating them.

The process produces a working system.

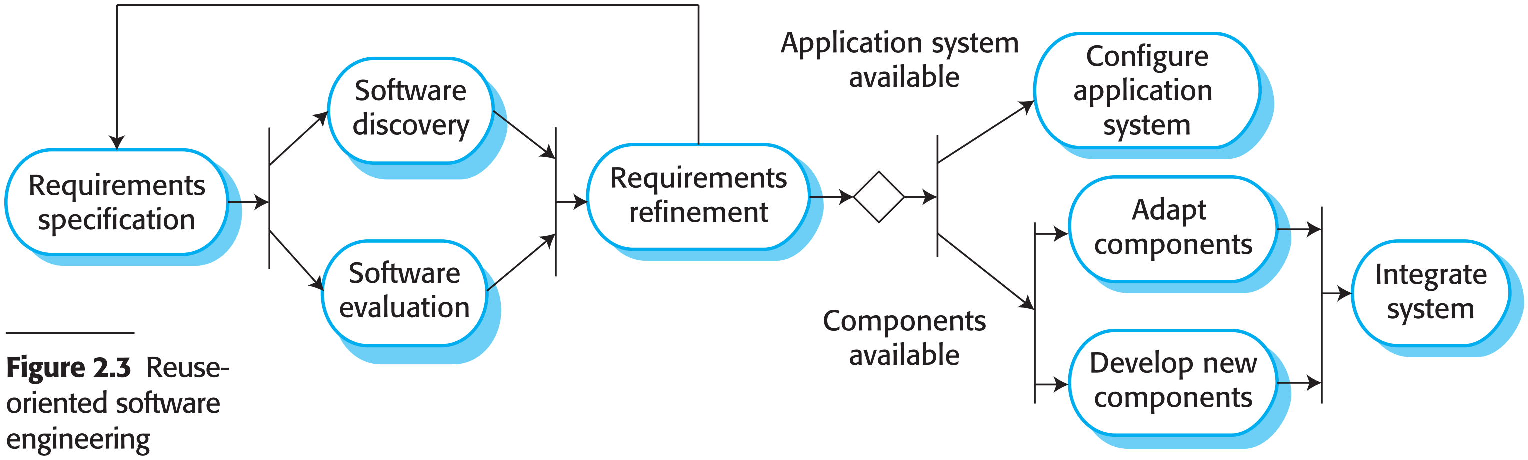


Figure 2.3 Reuse-oriented software engineering

Figure 2.3. Integration and configuration lifecycle model.

Note that, in Figure 2.3, some of the activities are different from the activities in the other software lifecycle models.

Section 2.2 - The Typical Activities in a Software Development Lifecycle

This important section describes the major activities in the software development lifecycle. This will be left to the students to read and take notes.

Section 2.3 - Coping with Change

Change is inevitable in all large software projects. Processes should be organized to anticipate changes that will likely occur in the project.

The introduction of a new system into an existing workflow often reveals new possibilities for the system, and thus creates new software requirements.

One way to cope with an anticipated change in system requirements, is to adopt prototyping lifecycle model.

A **prototype** is an early version of a software system that is used to demonstrate concepts, try out design options, and find out more about the problem and its possible solutions. (from Sommerville, 2015)

Real-World Examples of Requirements Changing

Example #1: EyeDraw Version 1 permitted children with severe disabilities to draw pictures using their eye movements, but also revealed the need for numerous software improvements, thus leading to Version 2. https://dl.acm.org/doi/abs/10.1145/1054972.1054995

Possible Example #2: Facebook causes harm. There is perhaps a need for some new requirements. https://www.wsj.com/articles/the-facebook-files-11631713039

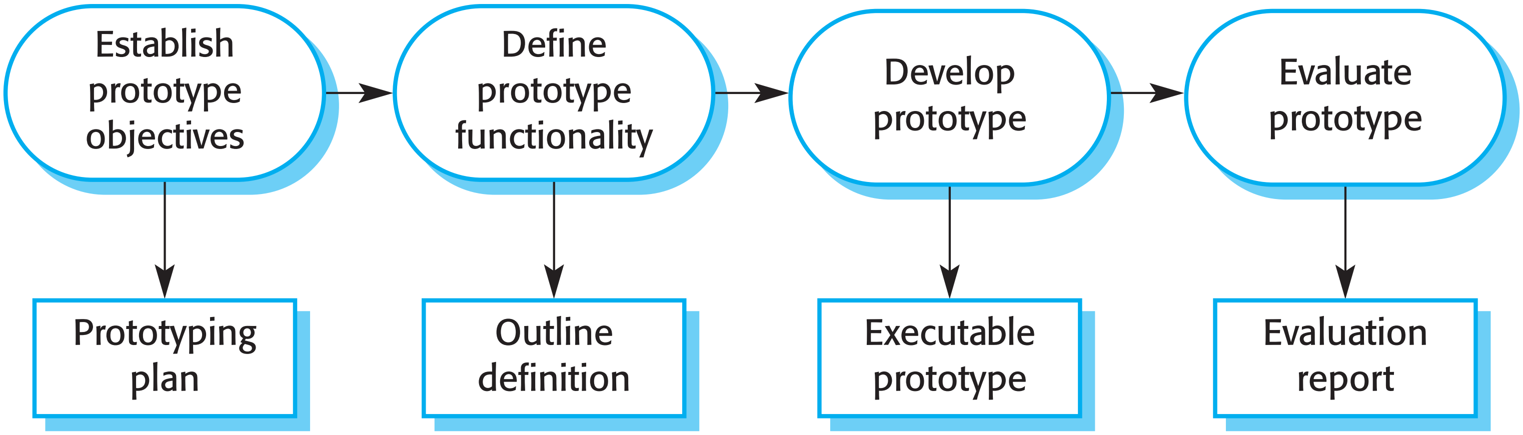


Figure 2.9. The prototyping phase of a lifecycle model. Note how this can be expanded into a full project lifecycle model, as is shown immediately below in this lecture.

Bonus Example Model #4 - Prototyping Lifecycle Model

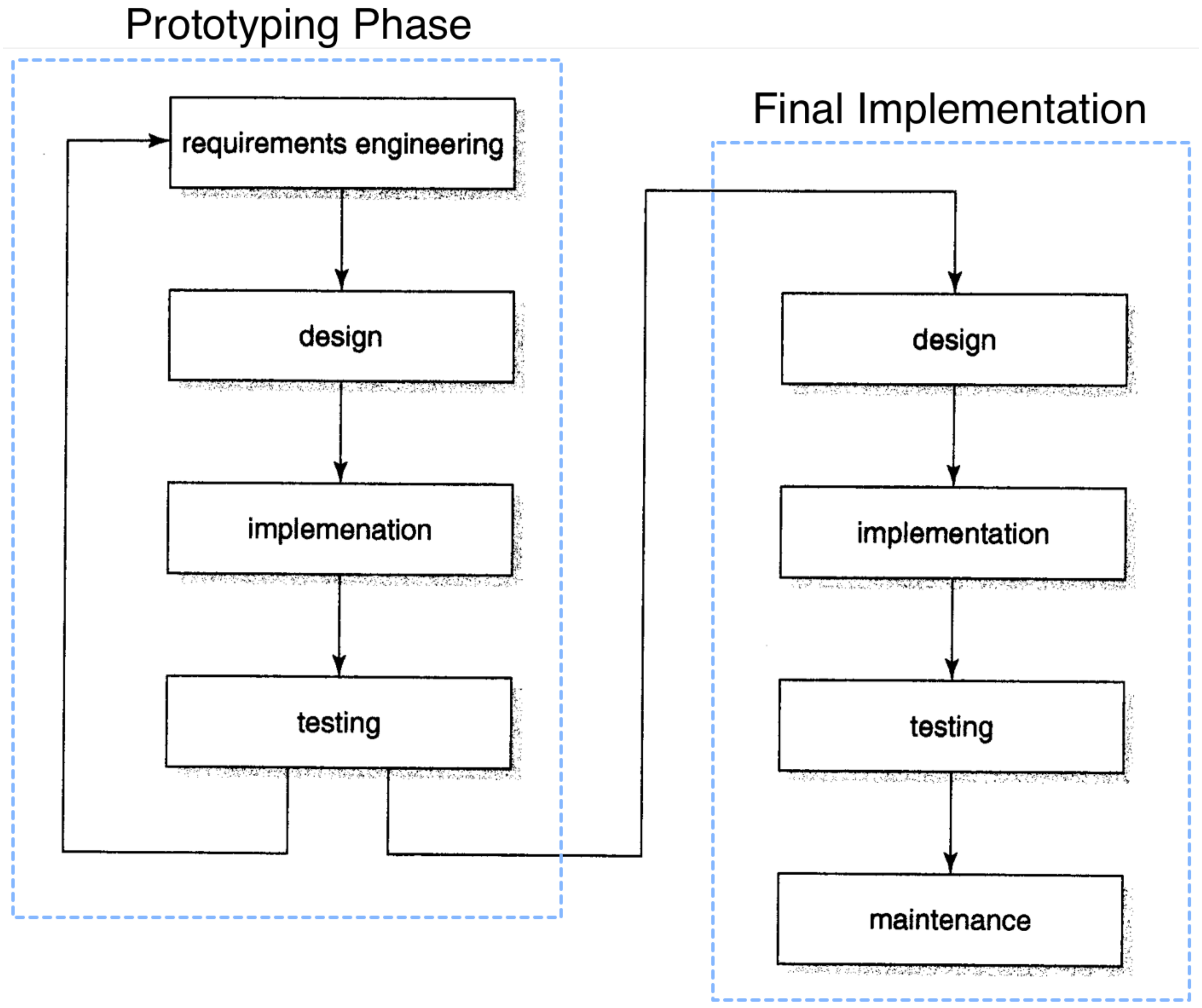


Figure 3.3 Prototyping as a tool for requirements engineering

Figure 3.3, adapted from van Vliet (2008), shows the entire software lifecycle model in which prototypes are iteratively developed. The prototyping phase ends when the testing of the prototype reveals that the system is ready for its final development.

Key: Boxes are activities, and the arrows are the order of activities. The dashed-line boxes show groups of activities, an additional layer of abstraction.

Chapter 4 - Requirements Engineering

(Most of the notes below are copied directly from Sommerville, 2015.)

In short:

Requirements describe what the system will do.

Design describes how the system will work.

Requirements engineering (or requirements analysis) is arguably the hardest phase, and the most important. The longer it takes to find a problem in a project, the more costly it will be to recover from that problem. Errors that are not discovered until after the software is operational cost 10 to 90 times as much to fix as errors discovered during the requirements analysis phase. If you are delivering the software and realize your software is not doing what the customer needs, that is a very costly problem. See the graph below.

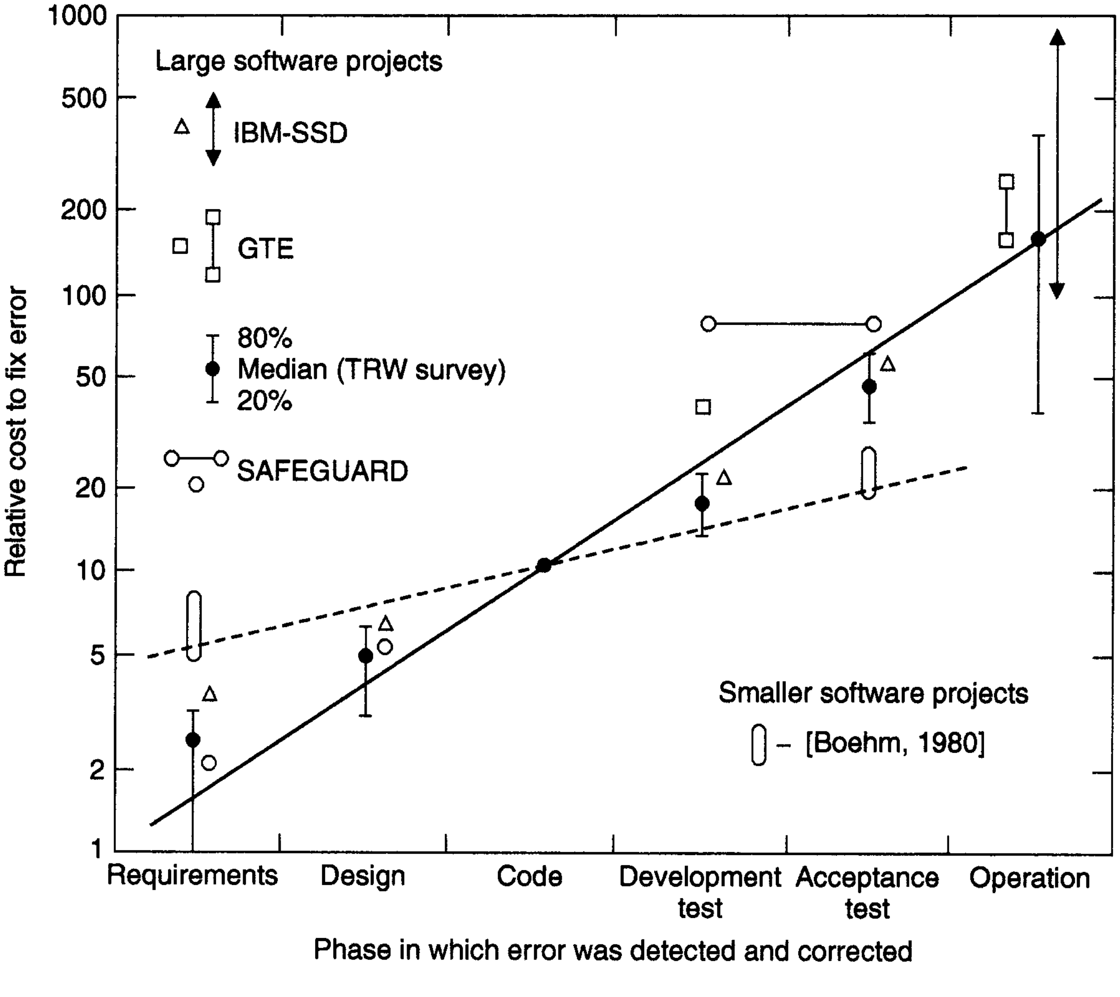


Figure 13.1 Relative cost of error correction (*Source*: Barry B. Boehm, *Software Engineering Economics*, Figure 4.2, p. 40, ©1981, Reprinted by permission of Prentice Hall, Inc. Englewood Cliffs, NJ.)

Example: My Mom worked as an export specialist at Tektronix in Beaverton, Oregon, from 1975 to 2000. Some time in the 1990s, Tektronix hired a major consulting company to build a new export order-entry system.

To establish the requirements, the consultants met with the export managers, but not the export specialists who actually did the order entry.

The system was deployed. On its first day of use, my Mom called the special hotline to ask how to split an order across two invoices (to accommodate bureaucratic needs for foreign customers). The consultants told her to just put it onto one order. My Mom explained "Hmm, I can't do that. In order for me to sell this product to this customer in this country, I have to split the order."

The functionality was not implemented because the consultants did an inadequate requirements analysis. They only met with the managers, not the export specialists. And the managers did not know how the export specialists did their jobs, or even how to enter an order.

The consultants had to re-define the system requirements. They had to go back and re-implement major portions of the system. It was expensive.

4.0 INTRODUCTION TO REQUIREMENTS ENGINEERING

The ***requirements*** for a system is a description of (a) the ***services*** that a system should provide and (b) the ***constraints*** on its operation. These requirements capture what the customer needs in the system.

Requirements engineering (RE) is the process of finding out, analyzing, documenting and checking these services and constraints. This is a difficult process. It is often difficult to figure out, in advance, what the client needs.

One general approach to solving this challenge, as with many challenges in software engineering, is to break the problem up in to smaller pieces, and solve each problem, and do it separately. For example…

(from van Vliet, 2008)

Separate out the ***user requirements*** from the ***system requirements***, and be sure to cover both.

1. ***User requirements*****:** The services the system is expected to provide to users, and the constraints under which the system must operate. The user requirements should be understandable by any stakeholder familiar with the application domain. In the SRS Template, this is “2. The Concept of Operations (ConOps)”.

2. ***System requirements*****:** A more detailed descriptions of the software system’s functions, services, and operational constraints. This is a functional specification that defines exactly what is to be implemented. In the SRS Template, this is “3. Specific Requirements”.

See the SRS Template for this class.

…

4.1 FUNCTIONAL AND NON-FUNCTIONAL REQUIREMENTS

Within the system requirements, you can also “divide and conquer” by separating the ***functional requirements*** from the ***non-functional requirements***.

1. ***Functional requirements*****:** Statements of the services that the system must provide, or descriptions of how computations must be done. For example, how the system should respond to specific inputs, or compute statistics.

2. ***Non-functional requirements*****:** Constraints on the system and the development process used to build it. For example, timing requirements, display parameters, ranges of inputs accepted, or the requirement to use a particular IDE or software development lifecycle model.

See the SRS for the NRL Dual Task software.

It is useful to separate ***functional*** and ***non-functional*** requirements, but this creates the challenge of positioning related requirements near each other in the SRS, so they can be considered side-by-side. You can solve this problem by structuring your SRS around major functions, such as shown here, and listing the functional and non-functional requirements by major function.

System Requirements for the HTML editor.
...
3. Editing
3.1. Functional
3.1.1. The user can edit the preview of the web page.
...
3.2. Nonfunctional:
3.2.1. The preview rendering must be compliant with HTML5.
...
4. Printing
4.1. Functional:
4.1.1. The user can send the web page to the printer.
...
4.2. Nonfunctional:
4.2.1. The default printer settings (set within the OS) will be used.
…

4.2 REQUIREMENTS ENGINEERING PROCESSES

The ***requirements engineering*** process includes:

1. Requirements ***elicitation*** (Section 4.3).	Pulling it out.
2. Requirements ***specification*** (Section 4.4).	Writing it down.
3. Requirements ***validation*** (Section 4.5).	Double-checking it.
4. Requirements ***management*** (Section 4.6).	Updating it.

Each is defined below.

…

4.3 REQUIREMENTS ELICITATION

Requirements elicitation is the process of drawing out information from relevant stakeholders.

“Elicit” means “to bring out” or “to extract”. For example, raising your hand and asking a question usually elicits a response from the instructor.

Requirements elicitation is an iterative (repeating) process that includes a spiral of activities—requirements ***discovery***, requirements ***classification and organization***, requirements ***negotiation***, and requirements ***documentation***.

Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:

1. Stakeholders often don’t know what they want from a system, have a hard time expressing it in a useful way, or have unrealistic hopes and dreams.

   For example, requests might be too general (such as “a system to track our inventory”) or too specific (such as “a button that prints a report…”) or unrealistic (such as “a system that permits our daughter to communicate”).

2. Stakeholders express requirements with implicit knowledge of their work.

   For example, asking why deejays in Berlin still used vinyl in 2008, a store owner offered a few reasons including ‘you don’t kill your mother’, ’you can just look at the vinyl’, and ‘because I can afford my rent’.

3. Different stakeholders express their requirements in different ways.

   An assistive technology device can “provide a sense of agency”, “enable her to do something for herself”, or “help her to learn cause and effect”.

4. Office politics can influence the requirements of a system.

   For examples, managers might want reports that make them look good.

5. The economic and work environment can change.

   Funds become available or dry up. New stakeholders join the project.

…

4.3.1 Requirements elicitation techniques

To successfully elicit requirements, you must be curious.

Sommerville does a good job in this section but leaves out a number of big ways to elicit requirements:

1. **Read.** Read scholarly articles written by experts, interviews of experts, and books written by experts. The experts would have first-hand knowledge on the task your system aims to support.

   For example, if you want to build a system to track a global pandemic, you do not start by trying to interview experts. Instead, you start by reading scholarly literature on how to track global pandemics.

2. **Study existing systems.** Acquire software that accomplishes the tasks, or close to the tasks, that you want your system to support. You can discover requirements you had not thought of, and identify functionality that is missing, thus motivating the system you aim to build.

There are three fundamental approaches to requirements elicitation:

1. Interviewing, in which you ask people about what they do.

   For example, interview parents of children with severe disabilities.

2. Observation, in which you watch people doing their job to see how they do their job, what artifacts they use, how they use them, and so on.

   For example, observe researchers collecting data at a Tetris competition.

2. Ethnography, in which you *get a job* in the work environment where the system would be used, and gain first-hand knowledge of exactly how the work really gets done, and how people truly function in that environment.

   For example, volunteer at a home for children with severe disabilities.

…

4.3.1.1 Interviewing

To do a good interview, you must be curious.

How to prepare for an interview

1. Before the interview, learn everything you can from published materials.
2. Recruit appropriate people to interview. These will ideally be people who are established experts at doing exactly the task your system is designed to support. (But not just any random people for a new social networking app.)
3. Prepare a script of all of the questions you might like to ask. But don't follow the script precisely. Make it a conversation. But be sure to mark the essential questions that you really want to ask.

Two kinds of questions:

1. Open-ended questions such as:
   "Please tell me about working here."
   "Please tell me about the people that you interact with here."
   "Please walk me through a typical day."
   "Please tell me more about that."
2. Specific questions such as:
   "How many people do you typically feed in any given day?"
   "What motivates your customers to install solar panels?"
   "Please walk me through the patient intake process."

Start with open-ended questions and gradually transition to specific questions.

Listen and take short notes on things they say that you want to follow up on.

Get people to talk about *how they do their job*—the thing they are truly expert at—not their guesses of what they want in a system, or how it should work.

How to capture the interview:

1. ***One team member leads*** the interview, and ***other team members furiously take notes***. Review the notes together immediately after the interview.
2. ***Audio record*** the interview and transcribe afterwards (time consuming).

…

4.3.2 Stories and Scenarios

People find it easier to relate to real-life examples than abstract descriptions. Stories and scenarios (the two are essentially the the same thing) are ways of capturing how a system would work in a specific real-world setting.

…

Figuring out the appropriate requirements for a system is challenging.

You must learn as much as you can about the task, the users, and the environment in which the system will be used, and figure out ways that a system could do a good job supporting that task.

…

4.4 REQUIREMENTS SPECIFICATION

Requirements specification is the process of formally documenting the user and system requirements and creating a software requirements document.

The software requirements document is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.

It should be easy to reference, and easy to update.

The SRS Template on the course web site provides an excellent start.

…

4.5 REQUIREMENTS VALIDATION

Requirements validation is the process of checking the requirements for validity, consistency, completeness, realism, and verifiability.

Requirements should be:

1. ***Valid***. The requirements should reflect *the real needs* of system users.
2. ***Consistent***. Requirements should not contradict each other.
3. ***Complete***. The requirements should define all functions and constraints needed by the system user.
4. ***Realistic***. It should be possible to implement the requirements using existing technologies.
5. ***Verifiable***. The requirements should be written so that it can be objectively determined, such as with a defined test, whether each requirement been met.

Requirements can be validated through:

1. **Requirements reviews.** A team systematically analyzes the requirements, making sure each requirements meets all five of the above criteria.
2. **Prototyping.** An executable model of the system is built. Customers use it, ideally in a real-world setting. It is determined whether it meets the user's needs and expectations. Requirements can be updated.
3. **Test-case generation.** For each requirement, ask "How will I test this?" and include that test as part of the requirement. If you cannot describe a test for a requirement, the requirement needs to be revised.

   For example, "The system will be easy to use" cannot stand on its own, but needs to be defined in terms of specific terms, such as the speed and accuracy with which a typical user can accomplishing specific tasks.

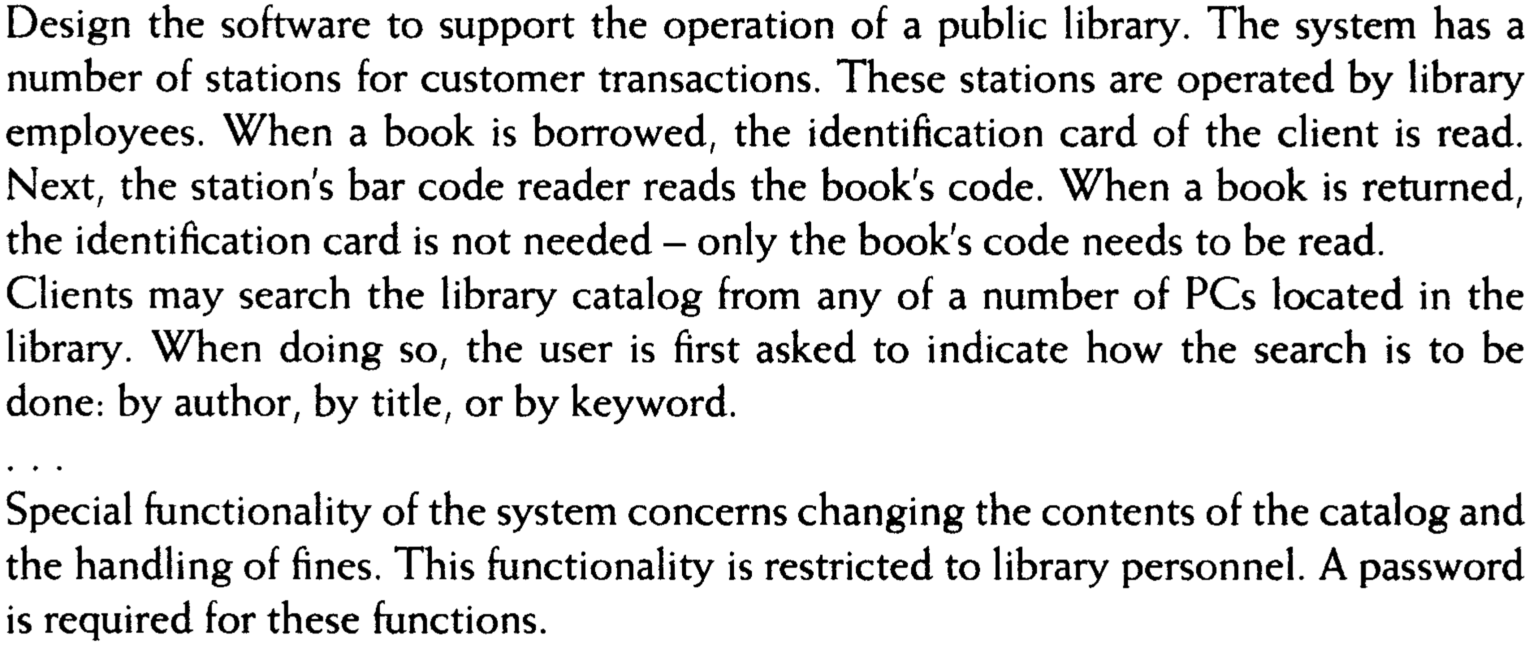
…

4.6 REQUIREMENTS MANAGEMENT

Business, organizational, and technical changes inevitably lead to changes to the requirements for a software system. ***Requirements management*** is the process of controlling, and making decisions about, these changes.

A ***problem statement*** provides a high-level description of the requirements.

Problem Statement for a Library Catalog (van Vliet, Figure 12.24)



Design the software to support the operation of a public library. The system has a number of stations for customer transactions. These stations are operated by library employees. When a book is borrowed, the identification card of the client is read. Next, the station's bar code reader reads the book's code. When a book is returned, the identification card is not needed – only the book's code needs to be read.
Clients may search the library catalog from any of a number of PCs located in the library. When doing so, the user is first asked to indicate how the search is to be done: by author, by title, or by keyword.

. . .

Special functionality of the system concerns changing the contents of the catalog and the handling of fines. This functionality is restricted to library personnel. A password is required for these functions.

Problem Statement for an Automated Teller Machine (from Rumbaugh et al., 1991, p.151)

Design the software to support a computerized banking network including both human cashiers and automatic teller machines (ATMs) to be shared by a consortium of banks.

Each bank provides its own computer to maintain its own accounts and process transactions against them. Cashier stations are owned by individual banks and communicate directly with their own bank's computers. Human cashiers enter account and transaction data.

Automatic teller machines communicate with a central computer which clears transactions with the appropriate banks. An automatic teller machine accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints receipts.

The system requires appropriate recordkeeping and security provisions. The system must handle concurrent accesses to the same account correctly.

The banks will provide their own software for their own computers.

You are to design the software for the ATMs and the network. The cost of the shared system will be apportioned to the banks according to the number of customers with cash cards.

Chapter 5 - System Modeling

__ Take blank overheads and pens to class. In class, students create dynamic models/diagrams to explain something new about how their project will function.
__ Perhaps print and handout UML quick reference.
__ Take printout of problem statements.

The chapters introduce a number of diagramming techniques that are commonly used to communicate aspects of a system design.

The diagrams are called “models” because they serve as small-scale representations, or paper-based simulations, of aspects of the system.

‘The fundamental driver behind graphical modeling languages is that programming languages are not at a high enough level of abstraction to facilitate discussions about design.’ (Fowler, 2004.)

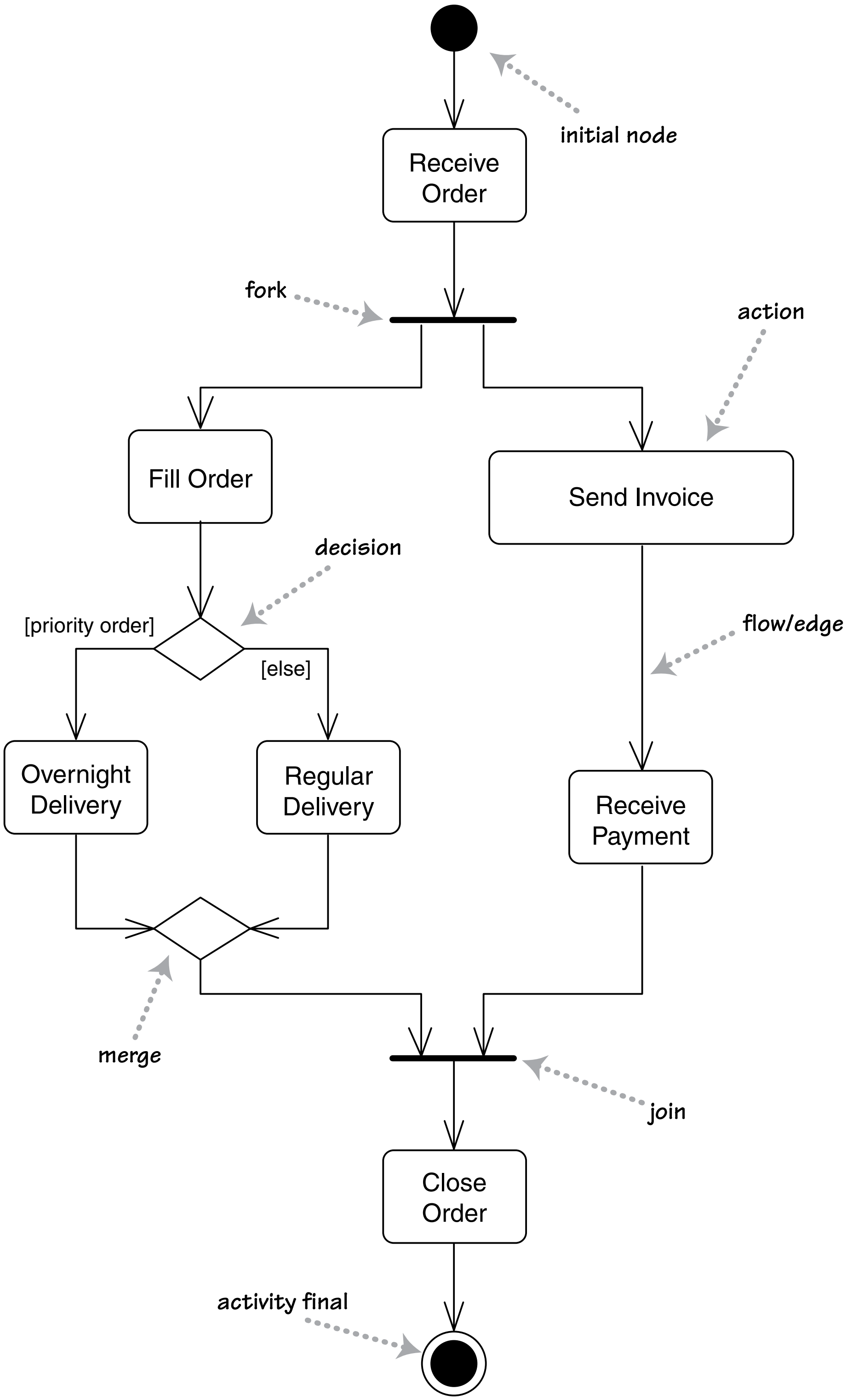
The models are **static** or **dynamic**.
Static show structure.
Dynamic show behavior.

Flowcharts are a classic *dynamic* model to show the flow of control of an algorithm. UML **activity diagrams** are very similar.

UML uses the terms "flow" and "edge" synonymously. (Fowler)

In any diagram, you generally need a key that explains what the boxes and lines represent.

However, if you are correctly using an established diagramming technique, citing a source can suffice.



A simple activity diagram, ***annotated (Fowler, 2004)***

The Unified Modeling Language

Diagramming techniques used in OOA and OOD (analysis and design).

Integrates and “unifies” the notations and methods of Booch, Jacobson, and Rumbaugh (object modeling technique, OMT), late 1980s and early 1990s.

There is also a UML *process*, but the language is still useful without the process.

(UML notes adapted from Sommerville, 2000, Software Engineering.)

There are other standard diagramming (modeling) techniques such as:

1. Entity Relationship Diagrams (ERDs) - similar to UML class diagrams.
2. Data Flow Diagrams - similar to UML sequence diagrams.

This lecture focuses on UML.

There are 13 different UML Diagrams, in the following hierarchy:

- *Structure*:
  - Class
  - Component
  - Composite Structure
  - Deployment
  - Object
  - Package
- *Behavior*:
  - Activity
  - Use Case
  - State Machine
  - *Interaction*:
    - Sequence
    - Communication
    - Interaction Overview
    - Timing

The underlined diagrams are those that are perhaps most commonly used.

Boxes and lines mean different things in each type of model.

Note how there is a fundamental distinction between static and dynamic.

Major diagrams used in UML:

Class diagrams: Static. Descriptions of the types of objects in the system, and the various kinds of static relationships that exist among them.

State-transition diagrams: Dynamic. Show all possible states (modes) that an object can get into as a result of events that reach that object.

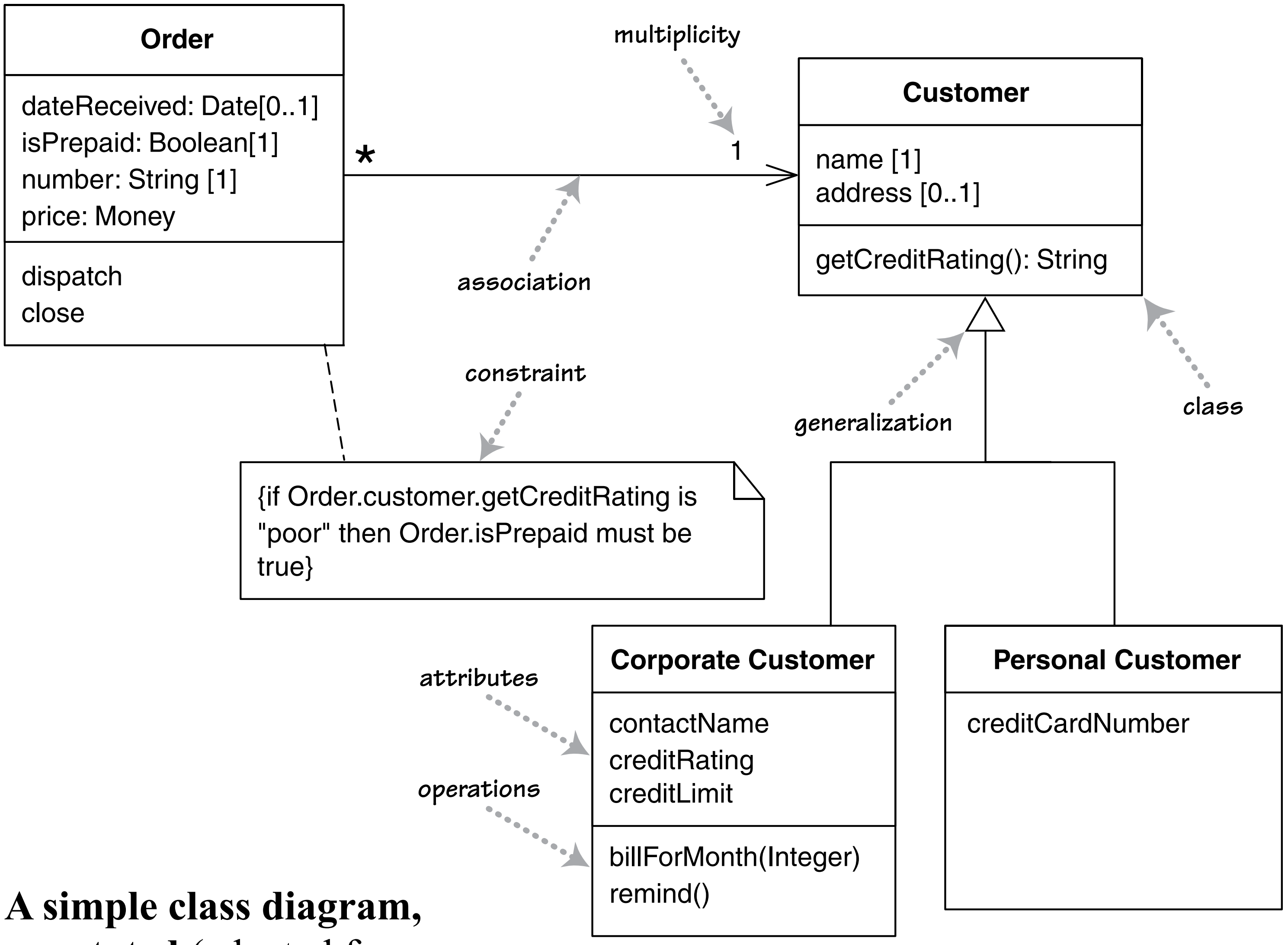
Sequence diagrams: Dynamic. Describe how groups of objects collaborate in some behavior. Show the sequence of object interactions

(UML notes adapted from Sommerville, 2000, Software Engineering.)

UML Class diagrams

Descriptions of the types of objects in the system, and the various kinds of static relationships that exist among them. Static model.

Include: Name of class, attributes and operations, inheritance (specialization). Associations, such as is-a-member-of, cardinalities.

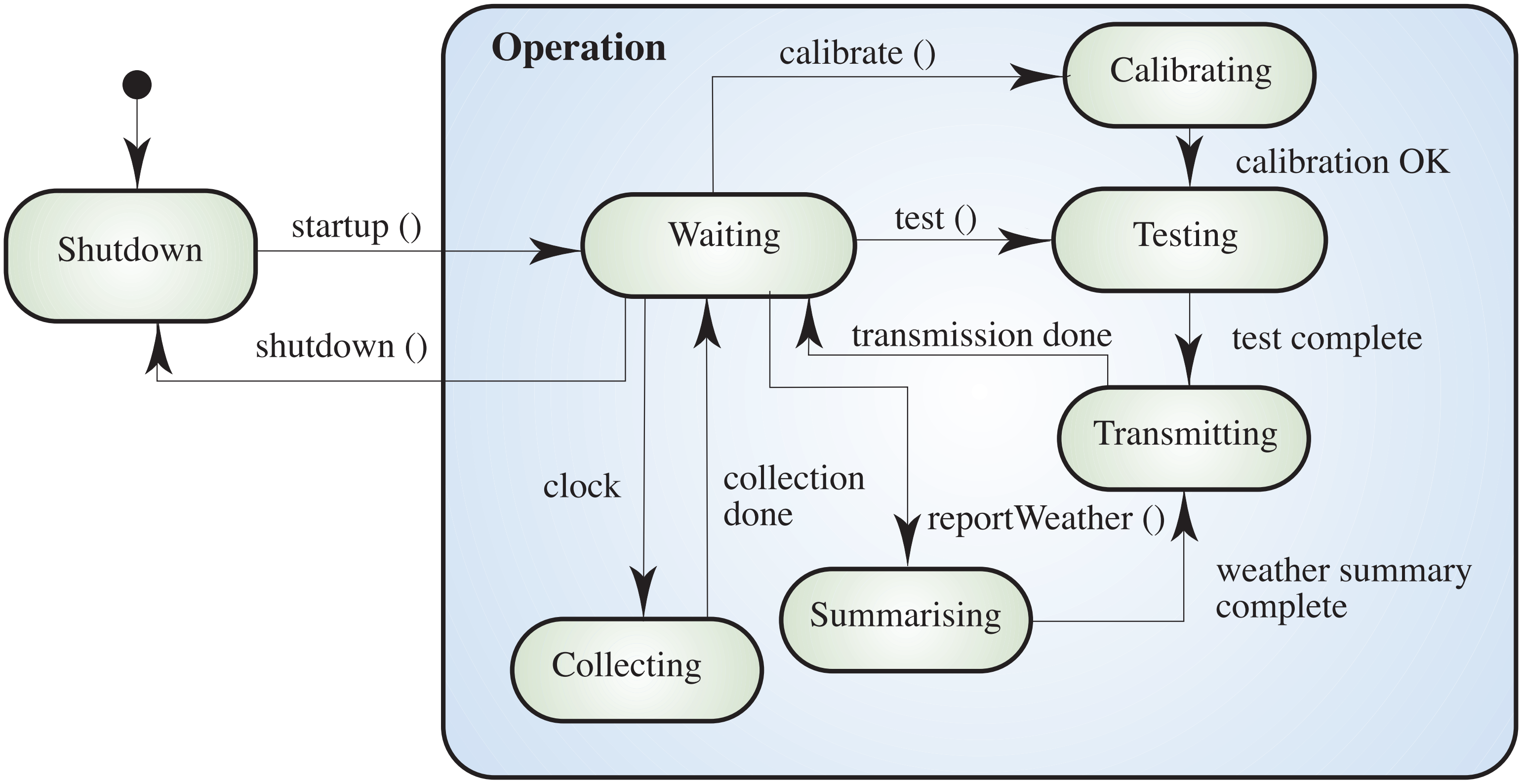


A simple class diagram, annotated (adapted from Fowler, 2004, Figure 3.1)

But ***dynamic*** models are also necessary to describe how a computer program works because a program executes over time. (A screenshot does not describe a user interface; you also need to describe the dynamic aspects.)

UML State Diagrams

A *dynamic* model illustrates the restricted states of an object or system.
The ovals are states and the arcs are events that cause the state to change.
Can have hierarchies of states, introducing abstraction.
Permit stakeholders to understand the of dynamic aspects of the system.

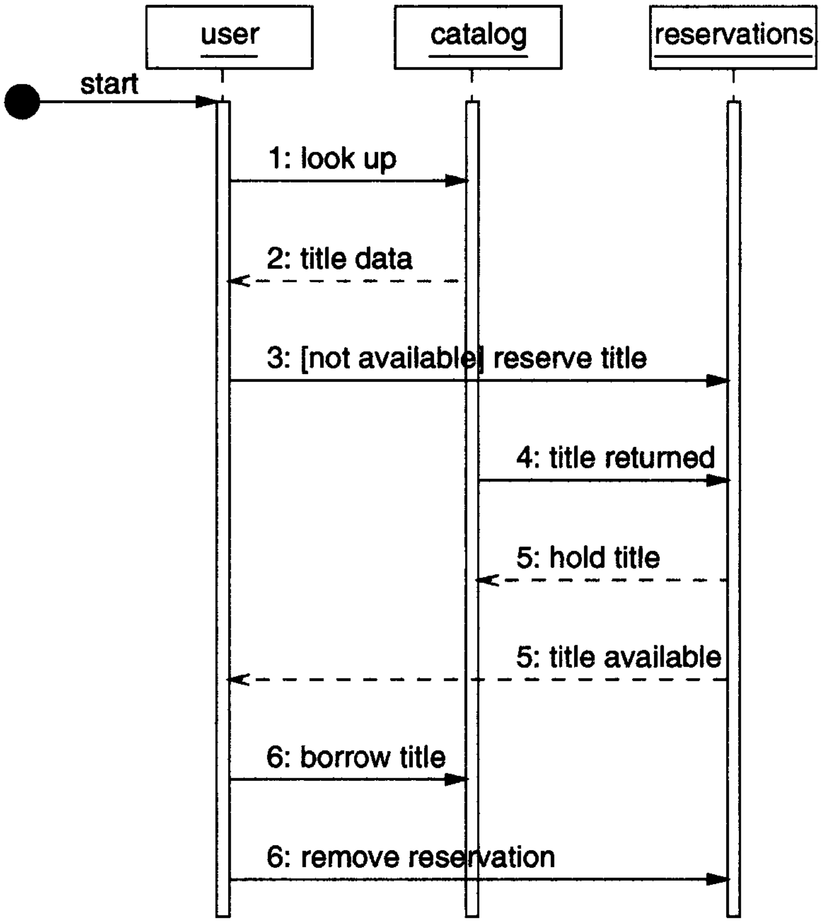


A state diagram for a weather station that, every five minutes, collects data, performs some data processing, and transmits this data.

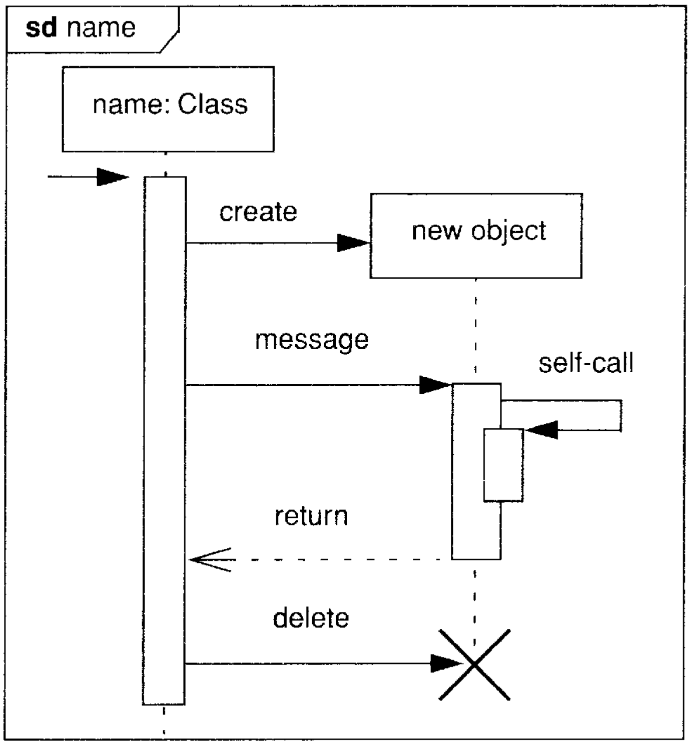
(From Sommerville, 2000, Software Engineering)

UML Sequence Diagrams

Dynamic models that describe how groups of objects collaborate to produce a system service or behavior. Shows the sequence of object interactions
Objects and users are shown at the top, each with vertical dashed *lifelines*.
Rectangles on the lifelines show when the object is active. Time moves down.
Solid lines show messages between objects. Dashed lines indicate a return.



A UML sequence diagram for reserving a title (vanVliet, 2008)



The template or key for a sequence diagram (Fowler, 2004)

The difference between a State Diagram and a Sequence Diagram

A state diagram says "All allowable sequences *must* conform to this state machine" whereas an interaction diagram says "Here is one possible sequence of actions." (Prof. Young, 11-9-2010)

Conclusion: UML evolved from earlier OOA and OOD methods, which evolved from earlier non-OO diagraming and design techniques.

All diagramming (modeling) techniques arrive at roughly the same models.

When you think about a piece of code that you are going to write, you think about the static and dynamic aspects of how that code will work.

Use standardized diagramming techniques to sketch out your ideas, both for yourself to think things through, and to communicate, record, and evaluate ideas with other team members and stakeholders.

Use these techniques to creatively propose and consider alternative designs.

Software design modeling is an important aspect of software engineering, the study of the full lifecycle of writing the code that run on computers.

Sommerville Chapter 5

(Some material is copied directly from the chapter.)

The chapter is consistent with the lecture above, but just offers a different presentation of the material.

The chapter focuses on five UML diagrams:

1. Activity diagrams.
2. Use case diagrams.
3. Sequence diagrams.
4. Class diagrams.
5. State diagrams.

UML diagrams can be developed to show **different perspectives of a system.**

1. An ***external*** perspective, showing the context or environment of the system.
2. An ***interaction*** perspective, showing the exchanges between a system and its environment, or between the components of a system.
3. A ***structural*** perspective, showing the organization of a system or of the data processed by the system.
4. A ***behavioral*** perspective, showing the dynamic occurrences of the system and how it responds to events.

The following organization is offered:

UML Models
- Context models: shows the environment of the system.
    - Activity diagrams
    - Context model" (which is not UML, but looks like an architecture).
- Interaction Models: between a system and its environment.
    - Use case diagrams (overly simple, but do focus on user task)
    - Sequence diagrams
- Structural Models
    - Class diagram
- Behavioral Models
    - Activity diagram
    - Sequence diagram
    - State diagram.

(Section 5.5 Model-driven engineering can be skipped.)

Chapter 6 - Architectural Design

(Most of the text below is copied directly from Sommerville, 2015.)

Overview

Software architecture: The large-scale (or top-level) decomposition of a system into its major components together with a characterization of how those components interact.

We are not talking about building architecture.

In computer science:
"Architecture" refers to the the design of the logic circuits in the chips.
"Software architecture" is what we are talking about today.

An architecture is typically a static (not dynamic) diagram.
"Module" implies static.
Specifying a software architecture for a system is an example of *modular programming*, which has long been understood as a key component of good programming.

Sommerville Section 6.0

Section 6.0 provides such a good overview of software architectures that I want to read it to the entire class, or have students take turns reading it to the class.
(The next sentences are topic statements copied from Sommerville, 2015.)

Architectural design is concerned with understanding how a software system should be organized and designing the overall structure of that system.... Architectural design is the first stage in the software design process
In agile processes, it is generally accepted that an early stage of an agile development process should focus on designing an overall system architecture. Incremental development of architectures is not usually successful.
To help you understand what I mean by system architecture, look at Figure 6.1. This diagram shows an abstract model of the architecture for a packing robot system.

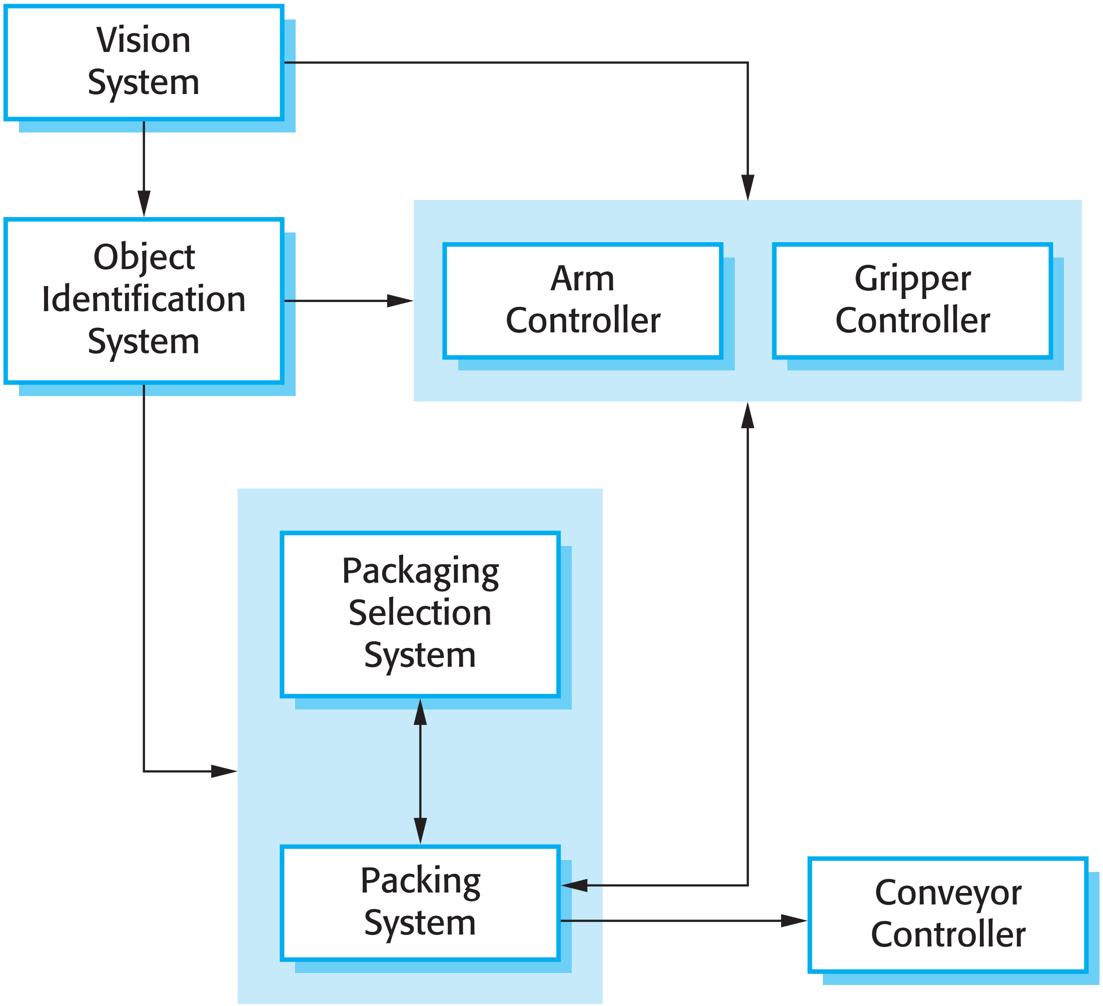


Figure 6.1. The architecture of a packing robot control system.

In practice, there is a significant overlap between the processes of requirements engineering and architectural design. Ideally, a system specification should not include any design information. This ideal is unrealistic, however, except for very small systems.

You can design software architectures at two levels of abstraction, which I call *architecture in the small* and *architecture in the large*:

1. *Architecture in the small* is concerned with the architecture of individual programs.
2. *Architecture in the large* is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components.

Software architecture is important because it affects the performance, robustness, distributability, and maintainability of a system (Bosch 2000).

Designing and documenting software architecture has three advantages:

1. Communication among stakeholders.
Stakeholders are all people with an interest in the system.
Q: Who are the stakeholders in the systems you are building now?

2. Captures design decisions.
The global structure of the system. Can provide insights into the *software qualities* of the system (reliability, correctness, efficiency, portability, ...) and work breakdown.

3. Transferable abstraction of a system.
A basis for reuse. Captures the essential design decisions. Provide a basis for a family of similar systems, or a product line, a "valued business entity" (Faulk).

System architectures are often modeled informally using simple block diagrams, as in Figure 6.1. Each box in the diagram represents a component. Boxes within boxes indicate that the component has been decomposed to subcomponents. Arrows mean that data and or control signals are passed from component to component in the direction of the arrows.

In spite of their widespread use, Bass et al. (Bass, Clements, and Kazman 2012) dislike informal block diagrams for describing an architecture. They claim that these informal diagrams are poor architectural representations....

The apparent contradictions between architectural theory and industrial practice arise because there are two ways in which an architectural model of a program is used:

1. As a way of encouraging discussions about the system design. A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail.

2. As a way of documenting an architecture that has been designed. The aim here is to produce a complete system model.... The argument for such a model is that such a detailed architectural description makes it easier to understand and evolve the system.

Block diagrams are a good way of supporting communications between the people involved in the software design process. They are intuitive, and domain experts and software engineers can relate to them....

They are widely used in industry.

6.1 Architectural design decisions

"Architectural design is a creative process in which you design a system organization that will satisfy the functional and non-functional requirements of a system."

…

Some Software Architecture Examples from Chapter 6

6.3.2 Repository Architecture

The Repository architecture describes how a set of interacting components can share data. In this architecture, all system data is managed in a central repository that is accessible to all system components. Components do not interact directly, but only through the repository.

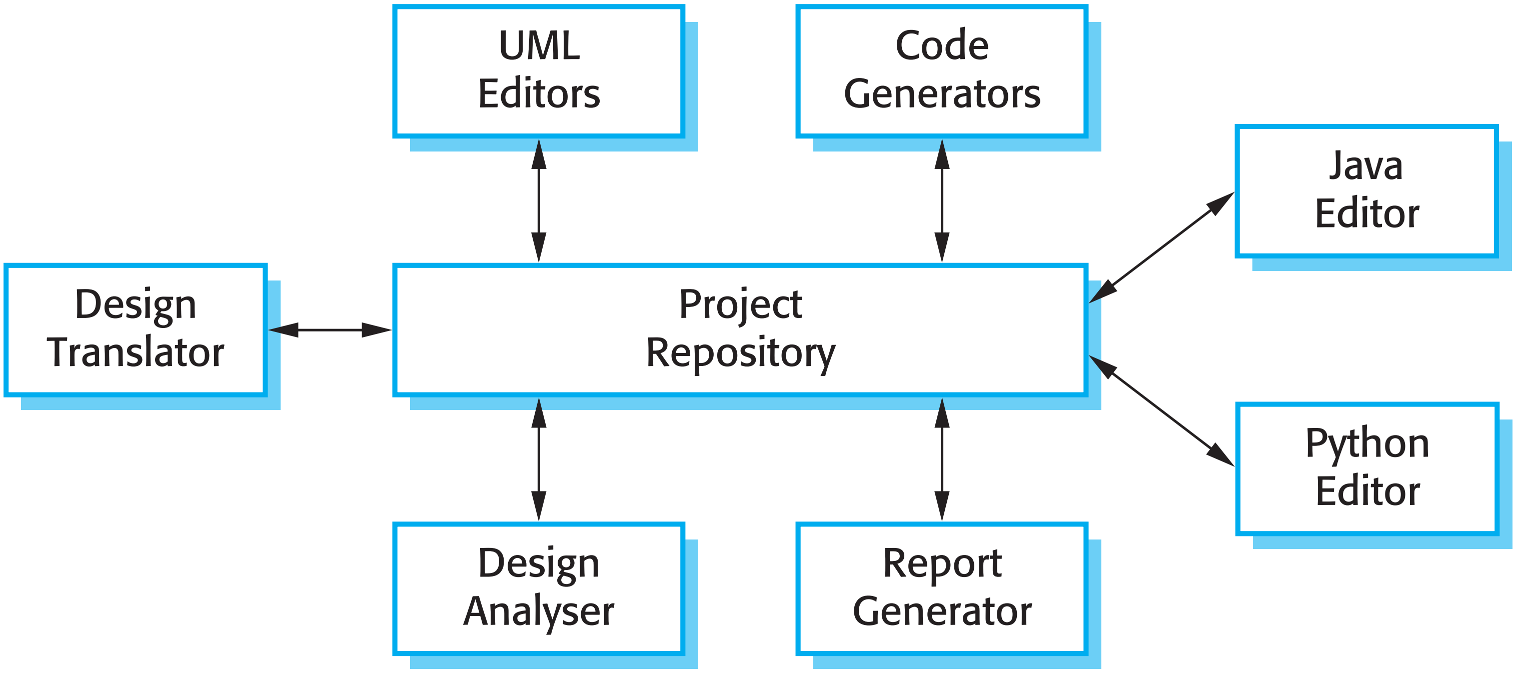


Figure 6.11 A repository-based software architecture for an integrated development environment (IDE) (Sommerville).

A Client-Server Software Architecture

A system that follows the Client–Server pattern is organized as a set of services and associated servers, and clients that access and use the services.

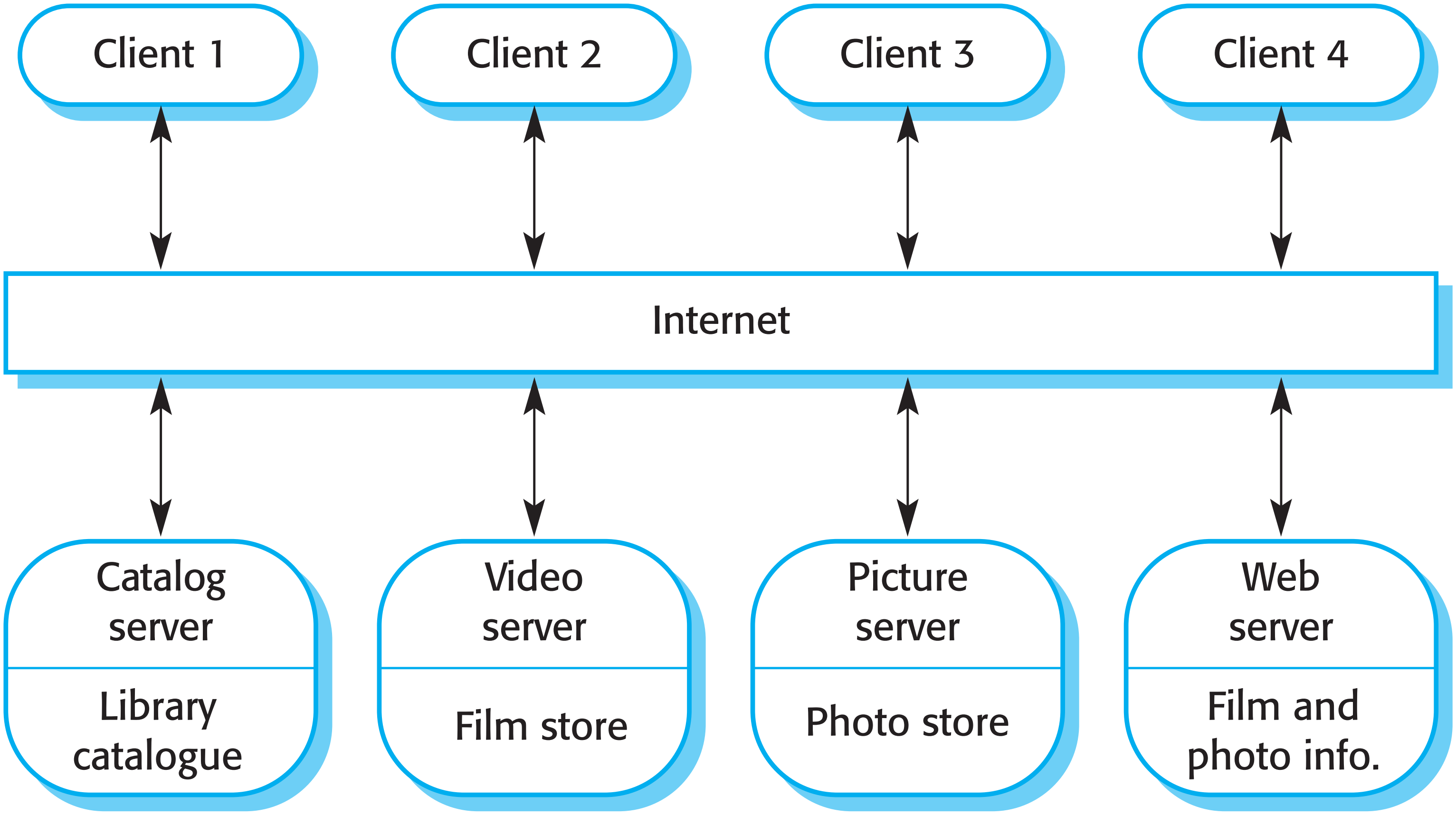


Figure 6.13 A client-server architecture for a film library

"Figure 6.13 is an example of a system that is based on the client–server model. [Though with a rather complex set of servers.] This is a multiuser, web-based system for providing a film and photograph library. In this system, several servers manage and display the different types of media. Video frames need to be transmitted quickly and in synchrony but at relatively low resolution. They may be compressed in a store, so the video server can handle video compression and decompression in different formats. Still pictures, however, must be maintained at a high resolution, so it is appropriate to maintain them on a separate server." (Sommerville, 6.3.3)

Most email usage follows a form of client-server architecture. You use one central email server, but many email clients to access that server. Each client needs to be configured for things like (a) the email host name and (b) having your full name appear in the "From:" header of emails you send.

Lecture on Software Design Principles

These notes are derived from Chapter 12 of van Vliet. (2008). *Software Engineering: Principles and Practice*.

This lecture introduces concepts that should help to guide your consideration of how to best break up a software system into modules.

Design Considerations

1. Abstraction
2. Modularity (coupling and cohesion)
3. Information hiding
4. Complexity (size based, structure based)
5. System structure

Abstraction

Abstraction is the process or outcome of concentrating on the essential properties of, and ignoring the details of, a set of related things.

Concentrate on the essential features and ignore—abstract from—those irrelevant to the current level. (For example, a sorting module sorts. You don't always care how.)

Procedural abstraction: The process or outcome of concentrating on the essential properties of, and ignoring the details of, *services or functions*. Examples: a *read*, *sort*, or *compute* module.

Data abstraction: The process or outcome of concentrating on the essential properties of, and ignoring the details of, *information or information structures*. Examples: a queue, a customer class. Object-oriented design identifies an abstract hierarchy in the program's data. Primitive structures such as booleans, ints chars, strings, are a form of data abstraction.

More examples of abstractions:

(This list is possibly from Michal Young.)

Interface	Provides abstract service	Abstracts over
TCP (Transmission Control Protocol)	Reliable communication.	Routing, transport, comm. protocols.
SQL (Structured Query Language)	Relational database.	Storage structure, concurrency control.
Java Swing	GUI widgets, interaction.	OSs, window system, graphics toolkits.

Modularity

Modules are separable pieces of code. The function of each module and each interface between modules needs to be defined precisely.

Parnas (1972) states the benefits of modular design:

(1) *Managerial*: Development time should be shortened because separate groups can work in parallel, with minimal communication.

(2) *Product flexibility*: It should be possible to make drastic changes to one module without changing the others.

(3) *Comprehensibility*: It should be possible to study and understand one module at a time.

Comparing different modular decompositions and interfaces reveals two structural design criteria: ***Coupling and Cohesion***.

Coupling is a measure of the strength or number of intermodule connections. In general you do not want strong dependence between modules. Rather, you want "loose" coupling between modules so that modules can be understood and developed independently. Tight coupling would result in any changes creating a large ripple effect across other modules.

Loose coupling might be achieved in different ways for different programs, such as sometimes by grouping similar services (putting all the reading and writing functions in one module), and sometimes by grouping services for a particular kind of data (putting all the functions for modifying customer records in one module).

Cohesion is a measure of the similarity, or mutual affinity, of the components within a module. You want "strong" cohesion within a module, meaning that similar components are grouped together. Cohesion is like the "glue" the holds a module together.

There are many ways to group components into modules: logical (input versus output), temporal, procedural, communication with other systems. You should be able to write down a single purpose for each module.

Information Hiding

Information hiding is the process or outcome of keeping implementation details *known only within* a component, such as within a module, function, or data structure. It does *not* relate to data security, such as making sure that certain users don't have access to certain data. It *does* relate to the data and functions in a component (such as a class or a module) that are made available to other components, such as through "getters" and "setters", or through an application programming interface (API). It helps you to organize your code.

(It is like the hints or mnemonics you use to remember someone's name. Don't tell them! You are the module. Keep that information hidden inside you.)

It is usually easier to use a software interface if its behavior is well-specified, and you only need to know how to use it, not how it works internally.

When designing a program, you need to decide what can be kept a secret, and what other components "need to know".

Information hiding is related to abstraction, cohesion, and coupling. Information hiding can improve cohesion and decrease coupling.

Complexity (This is *not* "Big-O" complexity.)

It is a measure of how complicated is the system. For example:

- intra-module connections (attribute of individual module)
- inter-module connections
- size-based (i.e. LOC == lines of code). Perhaps limit the size of modules.
- structure-based (complicated control structure)

System Structure

An outcome of design: modules and dependencies.

<u>Relations can include:</u>

Module A ***contains*** Module B

Module A ***follows*** Module B

Module A ***delivers data to*** Module B

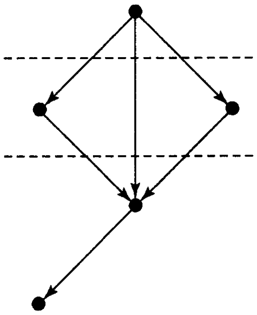
Module A ***uses*** Module B ——————————>

The use-relations shown in the "call graph".

If acyclic, we can identify a hierarchy.

We can measure the *size*, *depth*, and *width* of graph.

We (pre)tend to follow a top-down decomposition.



In-class exercise:

(a) Work in pairs and focus on one or more of these design considerations as you design or re-design your architecture or a component.

(b) Identify how these design considerations have already influenced your architectures or the plan for a component.

Section 7.2 - Design Patterns

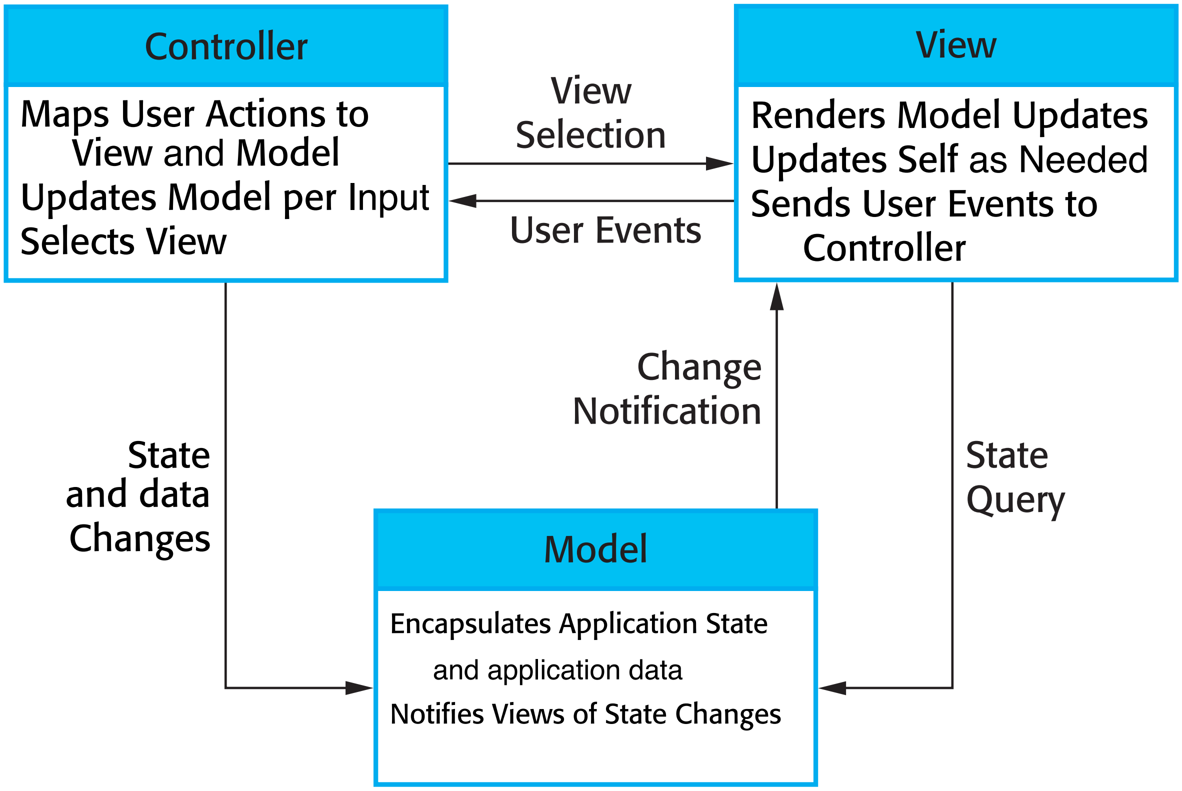
“Software Design Patterns” are solutions to recurring problems in computer programming, usually object-oriented computer programming.

"Patterns have made a huge impact on object-oriented software design. As well as being tested solutions to common problems, they have become a vocabulary for talking about a design. You can therefore explain your design by describing the patterns that you have used. This is particularly true for the best known design patterns that were originally described by the 'Gang of Four' in their patterns book, published in 1995 (Gamma et al. 1995)." (Sommerville Section 7.2)

“Design Patterns” in *building* architecture refer to an approach to design, and a book by Christopher Alexander (“A Pattern Language,” 1977). For what it's worth, the book is embraced by some architects, dismissed by others. (See Saunders, 2002, A Pattern Language. Harvard Design Magazine, Winter/Spring 2002, 16.)

Software design patterns, however, are widely accepted by programmers.

The model-view-controller architecture is the archetypal software design pattern. ("archetypal" == "that which captures the essence of")



The model-view-controller (MVC) software architecture (Sommerville).

Other Patterns: Observer (in the book), State (in CIS 443), Singleton, Factory.

Chapter 8 - Software Testing

The lecture is derived, and copies directly, from:
A guest lecture in this class on 11-7-04 by from Greg Foltz, a software tester from Microsoft.
Sommerville (2015) *Software Engineering*, 10th edition, Pearson.
van Vliet. (2008). *Software Engineering: Principles and Practice*.

Topics:
- Concepts and Terms
- Testing across the lifecycle. (Draw it and check off the boxes.)
- Microsoft interview question.
- Three approaches to testing.
- First Principles

Concepts and Terms

Testing is intended to show that a program does what it is intended to do, and to discover program defects before it is put into use.

When you test software, you are trying to do two things:

1. Demonstrate to the developer and the customer that the software meets its requirements. (***Validation*** testing: Show that it does the right thing.)
2. Find inputs or input sequences where the behavior of the software is incorrect, undesirable, or does not conform to its specification. (***Defect*** testing: Expose problems.)
   [Example: helping software developers fix errors with Bookends.app]

"Testing can only show the presence of errors, not their absence."
Testing cannot demonstrate that the software is free of defects or that it will behave as specified in every circumstance. It is always possible that a test you have overlooked could reveal further problems with the system.

Validation and Verification (or V&V):
Validation: Are we building the right product?
Verification: Are we building the product right?

Commercial software typically goes through *three stages of testing*:

1. ***Development testing***, in which the system is evaluated during the implementation to discover bugs and defects. System designers and programmers are likely to be involved in the testing process.
2. ***Release testing***, in which a separate testing team evaluates a complete version of the system before it is released. The goal is to make sure that the system meets the system requirements.
3. ***User testing***, in which real users use the system to do real tasks in a real-world environment. ***Acceptance testing*** is one type of user testing in which the customer formally tests a system to decide if it should be "accepted" from the system supplier, or if further development is required.

Testing across the lifecycle

The conventional breakdown of the software development process puts testing as a phase that occurs between implementation and maintenance.

The fact is, testing is an activity that occurs throughout the entire process.

The longer it takes to find an error, the more costly it is, and the cost goes up exponentially with each phase. Excellent graph. Conveys a lot of information, but is drawn to make a central point. (The median is the value that separates one half from the other.)

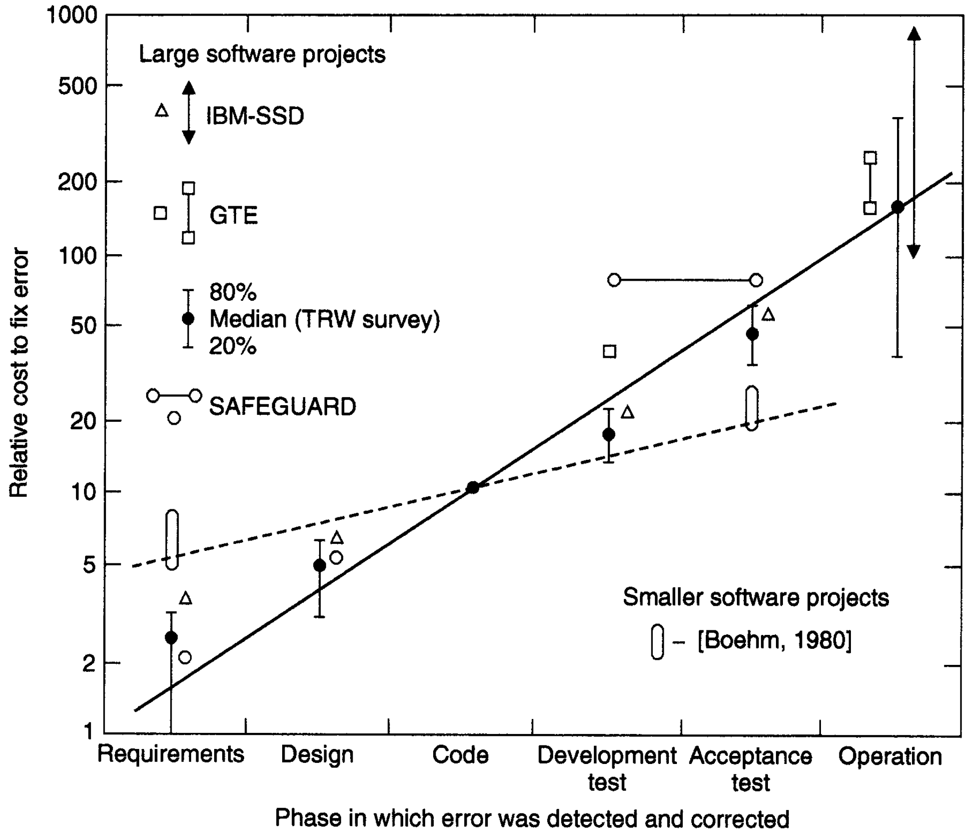


Figure 13.1 Relative cost of error correction (*Source*: Barry B. Boehm, *Software Engineering Economics*, Figure 4.2, p. 40, ©1981, Reprinted by permission of Prentice Hall, Inc. Englewood Cliffs, NJ.)

The graph reminds us how even the waterfall model has V&V in every phase.

Validation - Are we building the right product? Will it satisfy the requirements, the customer's needs?
Verification - Are we building the product right? Will it work? Will it accept the correct range of inputs, and map them to the correct outputs?

Requirements: What the system will do.
Design: How the system will do it.
Microsoft hires roughly one tester for each developer. The test team becomes the model user, the lead advocate for the user.

Testing in the Requirements Phase - V&V

Requirements: Is this what the customer wants? Are the features correctly prioritized? Do we have a good set of requirements to start the design? (Validation)
Requirements must be

- feasible (can it be built? tested? Easy to develop ≠ easy to test.)
- testable (objectively verifiable),
- consistent (internally (no conflict w/ others) and externally (w/ other components))
- complete (covers all cases, hardest to accomplish)

(Verification, or at least *preparing for* verification.)

When I critique your requirements and tell you to make them more objectively verifiable, it's not just an exercise in documentation. I'm trying to help you learn how to build better software systems by showing you how to evaluate, you might say *test*, your requirements.

How do you do it with these projects? As a group, have a session where you go through every single requirement, discuss whether it meets all of the above criteria. That is what we did with the NRL Dual Task Experiment software. It had to be implemented, and the main programmer and unit tester was one of the stakeholders—he or she needed to know what to do.

Verification: Testing in the requirements phase is mostly *planning* for verification. For every requirement, you should think ahead and plan on *how* to verify that requirement.

Designing test cases is creative work.

A description of the test case can serve as part of the requirement.

This points to the need for requirements to be precise and objectively verifiable.

Testing in the *Design* Phase

Design must also be

- feasible
- testable
- consistent
- complete

When I critique your designs and ask for more diagrams and specification of how the system is going to work, how it is going to be built, it's not (just) an exercise in writing specs or diagrams, it is to give you the opportunity to evaluate whether the thing will actually work. Many problems that come up near the end (such as a difficulty in both recording and listening to Skype audio) could have been identified earlier on through a rigorous design process, and consistency-checking with external components.

Testing in the *Implementation* Phase

This is where we typically think of testing being done.

Unit testing: Evaluate individual components such as methods or classes, called with different input parameters, and in different program states.

- Use test-case design techniques to develop thorough tests.
- Usually done individually in conjunction with coding. Usually.
- Done during implementation.

Component testing: Evaluate multiple components that interact with each other. Similar to unit testing, but you are now testing some integration.

- For example: evaluate programming interfaces, shared memory, called procedures, and messages that get passed.

System (or Integration) testing: Join components together to create a version of the system, and evaluate that integrated (joined-together) system.

- Done during the Implementation or Testing phases.
- Usually involves multiple team members.

8.1.2 Choosing unit test cases

- **Coverage-based:** Makes sure that some aspect of the product is evaluated exhaustively. Such as, *every* function call is called at least once, or *every* requirement is specifically evaluated, or *every* state-transition gets tried.
- **Error-based:** Focus on situations or places in which problems are likely to occur. Such as looking at the boundary conditions (where errors likely occur).

In all cases, you compare the real output to the expected output:

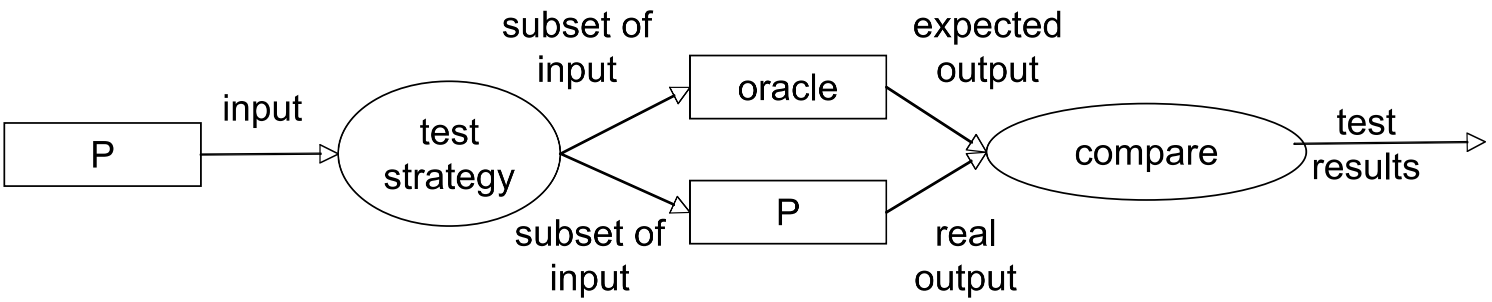


Figure 13.2 Global view of the test process. (vanVliet)

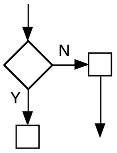
Interview question from Microsoft Interview:

A function takes a description of two rectangles in 2D space, and returns True if the two rectangles overlap, and False otherwise.

How would you test a function that returns the intersection of two rectangles? Specifically, what are all the inputs that you would provide to the test function? Presume that each rectangle is described by either (a) two (x, y) coordinates or (b) one (x, y) coordinate, an l, and a w. (droppeimage.pdf below in Pages)

Coverage-Based Techniques

In principle, every code segment that you write should have at least one associated test.

Path-testing or control-flow coverage. 

Branch coverage.

Data-flow coverage - how variables are treated down various paths.

Coverage of sequences of state transitions:

For example: Every possible path through the states in which every possible loop occurs 1 time.

Shutdown → Running → Shutdown

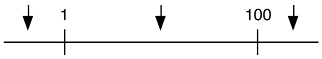
Configuring → Running → Testing → Transmitting → Running

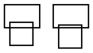
Running → Collecting → Running → Summarizing → Transmitting → Running

…

[See Chapter 5. System Modeling]

Note how a good design specification helps you to design good coverage-based tests.

Equivalence partitioning: Break the input into domains and assume that all inputs in a given range are equivalent. (You can do the same for ranges of output.)
For example, your function expects a number between 1 and 100, inclusive. You test in each region: 

You assume equivalence within the partitions, or walls. (For output, you might have three dialog boxes, and you just make sure that each will appear at one correct time.)
Same class: 

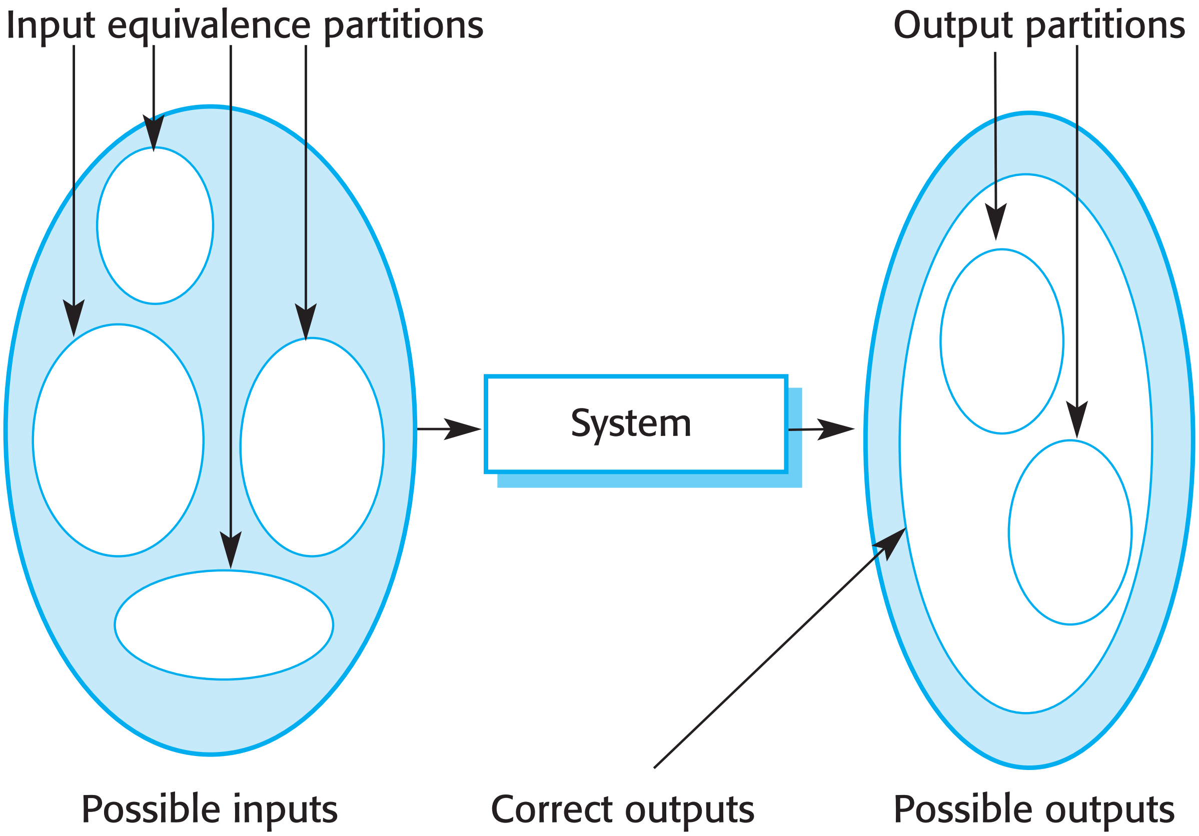


Figure 8.5: Equivalence partitioning. The large shaded oval on the left represents the set of all possible inputs to the program that is being tested. The smaller unshaded ovals represent equivalence partitions. The expectation is that a program being tested will process all of the members of an input equivalence partition in the same way. Output partitions are different classes of outputs that are possible.
[Sommerville, 2015]

Guideline-Based Techniques

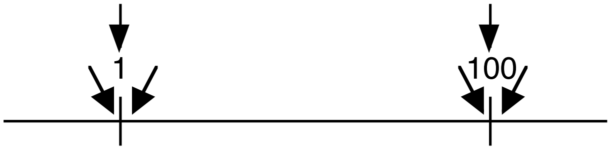
(Sometimes called “error-based”.)

Complementary to coverage-based.

Identify where errors are likely to occur based on the kinds of errors that programmers are likely to make.

Such as on the boundaries, “fencepost errors” and other “off by one” errors.

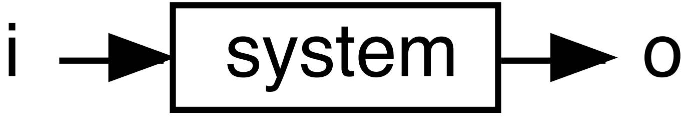
Test right on, and around each boundary:



Faults are likely to occur when two modules developed by different teams interact, so focus testing on the interaction between the these modules.

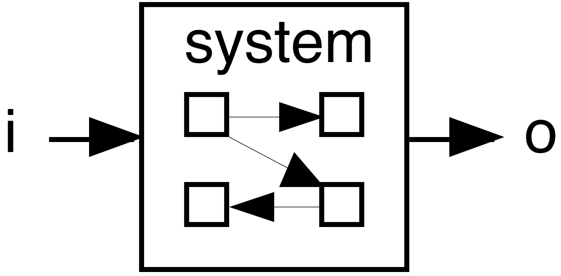
<u>Another way to organize testing approaches:</u>

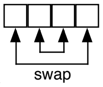
- **Black-box testing** (functional or specification-based). Test cases derived from specifications with little consideration of implementation details.



Examples: Equivalence classes and boundary testing.

- **White-box testing** (structural or program-based). Puts more emphasis on how the software works internally.



Example: You have to test a function that reverses a string. A naive way to program the function is to create a new string. A better way is to reverse in place. What are two different important test cases? Strings of even and odd length, to make sure the item in the middle is handled correctly in the strings of odd length. 

Testing in the *Test* Phase

"Code complete." All features are implemented. (Jargon. Book by McConnell.)

System testing or **Acceptance testing**, often driven by use case scenarios, how the system would likely be used.

System test days - at Microsoft, the developers or testers would try to do a real project with the system.

<u>Regression testing</u>: After a system is modified, you make sure new bugs were not introduced, that the code did not regress (go backwards). "Code churn causes bugs." 0.5 million bugs in building Microsoft Office.

Testing in the *Maintenance* Phase

Continue with all of the activities above as long as your software is being used. If your software is used, it will be modified.

First Principles

- **Bugs happen.** Faults are an integral part of the s/w development process. Anticipate them. But...
- **Impossible to test everything.**
- And... **Testing shows the *presence* of bugs, not their absence.** So...
- **Develop a plan.** Develop a system, an approach to do your testing.
- **Test early**: Early fault detection is important.
- **Test often**: In every phase.

Chapter 22 - Project Management

These notes are primarily copied from Sommerville (2015), and cover just a subsection of the assigned reading. Students should do all of the reading on their own following the SQ3R method.

Section 22.2 Managing People

Productivity is achieved when people are respected by the organization and are assigned responsibilities that reflect their skills and experience.

Four critical factors that influence the relationship between a manager and the people that he or she manages:

1. Consistency. People are treated the same, and held to the same expectations (given each individual's ability to contribute).
2. Respect. Different people have different skills. Everyone should be given the chance to contribute. [In this class, each student should be given a good opportunity to make a technical contribution.]
3. Inclusion. All ideas from all team members should be considered. Try to develop participation techniques to elicit contributions from team members who are more reflective, and less assertive in meetings. [Such as a quiet individual brainstorm followed by input from everyone.]
4. Honesty. Everyone should be clear and up front about what is going well, and what is not going well.
   "*The only thing worse than bad news is bad news late*."

Section 22.2.1 Motivating People

One way to think about motivating people is in the context of Maslow's hierarchy of human needs. In this hierarchy, each lower needs must be met before any of the higher needs can be met.

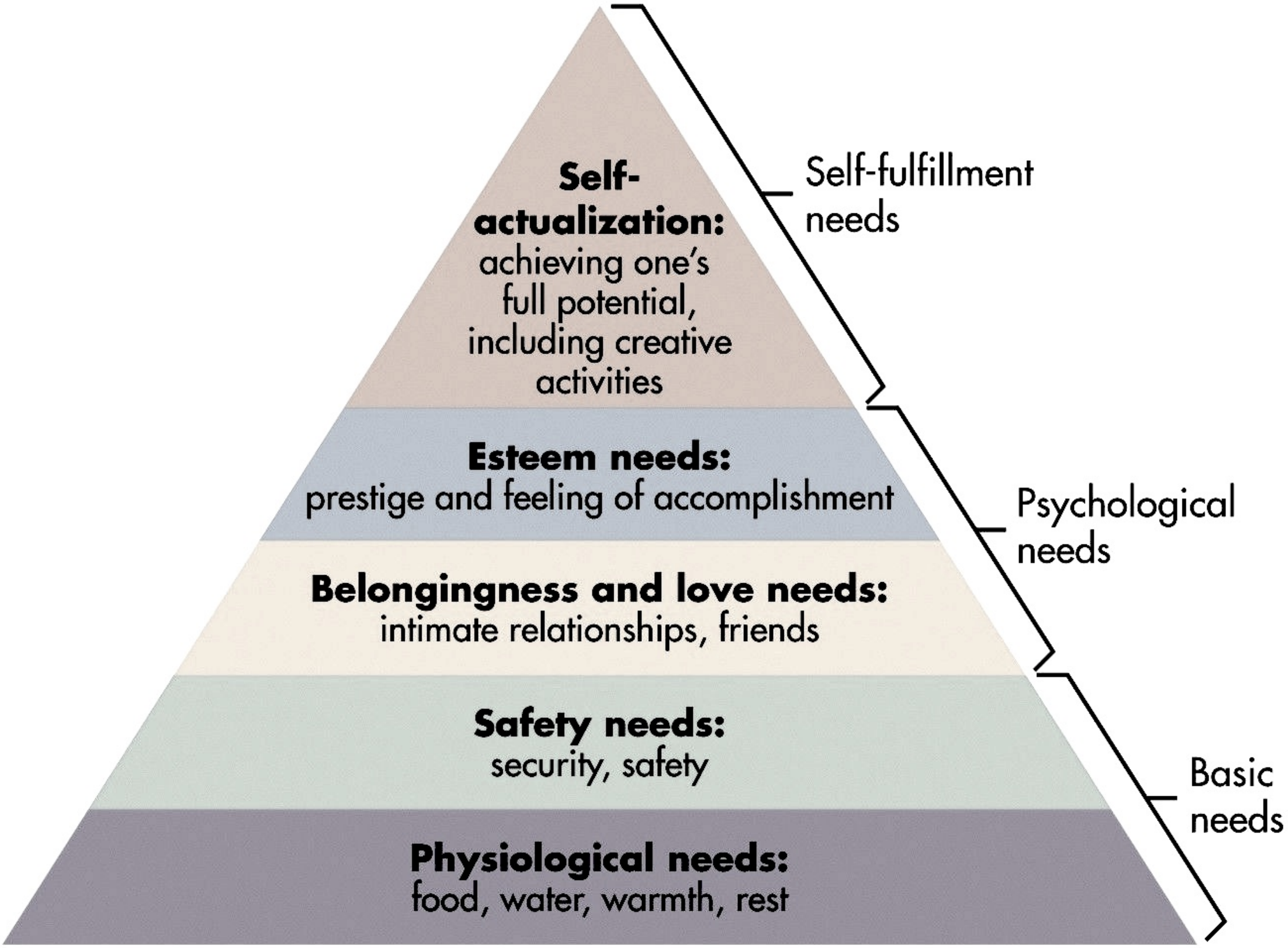

Self-actualization: achieving one's full potential, including creative activities
Esteem needs: prestige and feeling of accomplishment
Belongingness and love needs: intimate relationships, friends
Safety needs: security, safety
Physiological needs: food, water, warmth, rest
Self-fulfillment needs
Psychological needs
Basic needs

Figure X. Maslow's Hierarchy of Human Needs

(from https://www.simplypsychology.org/maslow-pyramid.jpg - 1-10-2022)

Human needs, starting from the bottom of the hierarchy.

1. Physiological. Team members must get enough sleep have adequate access to food. [Randy Pausch's Tips for Working Successfully in a Group.]

2. Safety. This includes team members feeling completely unthreatened in the workplace. ["Sexual Harassment In Silicon Valley: Still Rampant As Ever". September 15, 2020. *Forbes*.]

3. Belongingness. Team members should be recognized and appreciated as individuals. ["I see you." "I hear you."]

4. Self-esteem. People's contributions to the project, and to meetings, should be acknowledged.

5. Self-realization. People should be able to work at their level of ability, and to learn new things. [This does not mean to "follow your dreams".]

Maslow's hierarchy is a useful framework, but it does take a somewhat self-centered perspective, which can conflict with the need for a group to be cohesive and work well together. [Ask instead how you can contribute.]

Project Questions on Motivating People:

What are some ways that your group could better address either (a) Maslow's hierarchy of human needs or (b) Randy Pausch's "Tips for Working Successfully in a Group"?

Devise a specific proposal or request, and describe exactly how you will present it to your group.

Section 22.3 Teamwork

Teams need to be managed.
This is a task unto itself.
It requires consideration of alternatives.

cohesion means "sticking together tightly" or "forming a united whole".

A cohesive group values the group more highly than individuals in the group. Members of a well-led cohesive group are loyal to the group.

Benefits to a cohesive group include:

1. When the group makes decisions independently of outside influences, this contributes to a sense of independence and autonomy, and also of belonging.

2. Team members learn from each other.

3. Knowledge is shared so that people can help cover each other's tasks.

4. There is continual improvement to the overall product, not just parts of it.

Good project managers encourage group cohesiveness.

[Group cohesion among college football fans https://www.nytimes.com/interactive/2014/10/03/upshot/ncaa-football-fan-map.html]

"*One of the most effective ways of promoting cohesion is to be inclusive*." Treat group members as responsible and trustworthy, and make information freely available to everyone in the group. Everyone should know what is going on, should be able to name and contact all other group members, and have in mind at all times a general idea of what everyone is working on.

Project Questions on Teamwork:

What are some activities that your group does to promote cohesion? inclusion?

Does everyone in the group know everyone else in the group, and have one or more ways to communicate with that person? Such that the whole group can see the communication?

Does everyone in the group know what everyone else is working on? If not, what are a few different ways that could be improved?

What are some ways that any of the above might be improved?

Three factors that have a big effect on team work include:

1. **Who is in the group.** There should be a mix of skills.
2. **How the group is organized.** People should be able to contribute at their level of ability, and complete tasks as expected.
3. **Technical and management communication.** Good communication among all team members is essential

22.2.3 Group Organization

Important organizational decisions include:

1. Should the project manager and technical lead be the same person?
2. Who will be involved in making critical decisions, and how will the decisions be made?
3. How will interactions with external stakeholders be managed?
4. How will groups interact with team members who are not co-located?
5. How will knowledge be shared across the group?

Project Questions on Group Organization:

How are decisions being made about who will do what?

How are technical decisions being made?

Are there any policies about how to contact the professor?

How are co-located team members included (during Covid)?

What are some ways that any of the above might be improved?

Section 23.3 - Project Scheduling

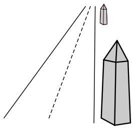
Some of these notes are from material that is not in Sommerville (2015).

Recall that software engineering is the process of gaining and maintaining control over the products and processes of software development.

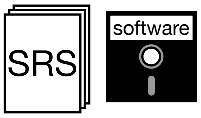
- "Intellectual control" ...
- "Managerial control" focuses on gaining and maintaining control over software development resources (money, time, personnel).

This lecture focuses on control of the resources of time and personnel.
Plans are nothing. Planning is everything. (Attributed to President Eisenhower)
"Begin with the end in mind". (Franklin Covey. 1989. The 7 Habits of Highly Effective People)

Project Planning Terminology



Milestones are distinctly identifiable points in the project timeline, named after stones that appear along the side of a road.



Deliverables are well-defined physical or digital objects that are handed over from one stakeholder to another.

Every deliverable can be a milestone, but every milestone does not necessarily have a deliverable associated with it (such as simply starting a task).

The **critical path** is the sequence of activities in a project such that, if any of these activities is delayed, the entire project is delayed.
(Not "the longest sequence of dependent tasks" as in Sommerville, 2015.)
(Yes "the longest-in-duration sequence of dependent tasks".)
You should always be working on activities that are on the critical path.
In the PERT chart below: How many days to complete project? What happens if the steam shovel breaks you have to dig the moat by hand, for 50 days?

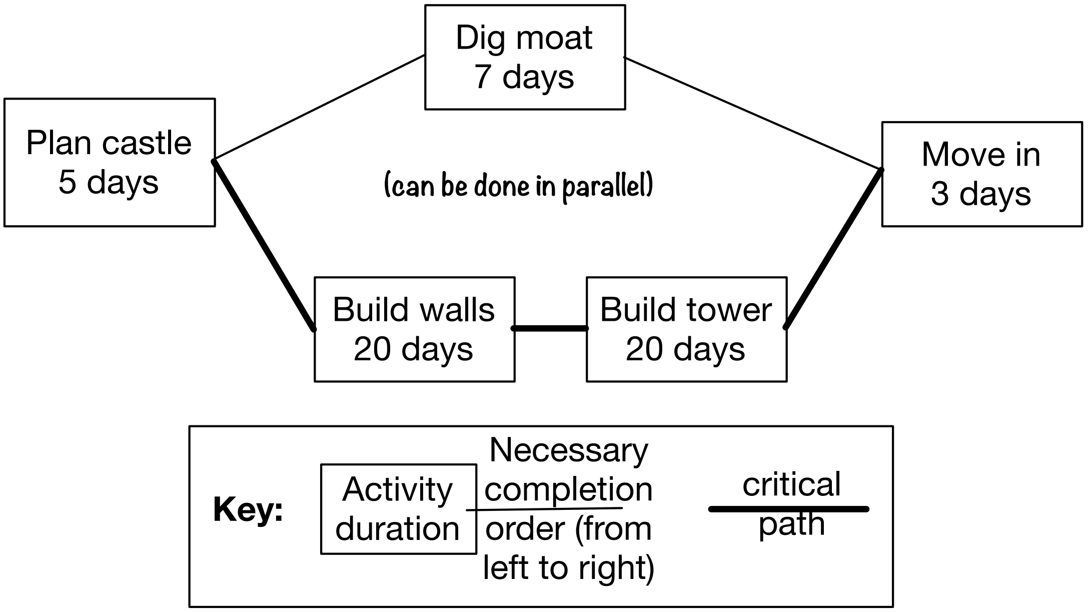
Slippage is the time a task (or project) is late compared to the original deadline. Slippage delays the project if the tasks with slippage are on the critical path.
Slack time is the time that a task can be delayed without delaying the project.

PERT Charts

Process Evaluation and Review Technique

(Developed during the 1950s Polaris missile program.)

The basic idea: Each activity gets a box. Lines indicate the necessary completion order (because of some kind of constraint).



A PERT chart for building and moving in to a castle.

Questions: How many days to complete the project? What happens if the steam shovel breaks you have to dig the moat by hand, for 50 days?

PERT charts emphasize the *critical path.*

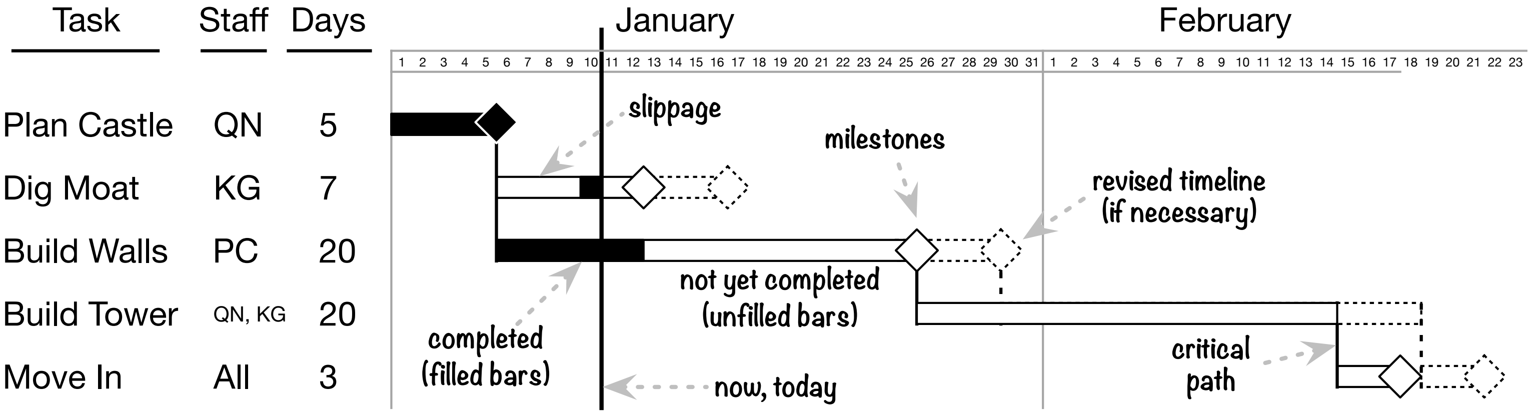
Gantt Charts (Timelines)

Named after Henry Gantt. (He developed them around 1910 to maximize the productivity of factory workers.)

The basic idea is as follows, though they can be drawn in many different ways.

Time moves from left to right.

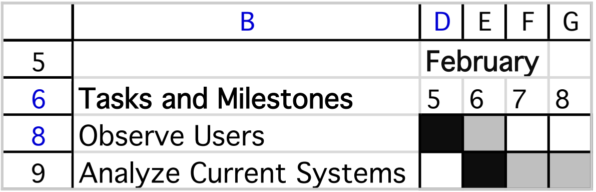
The time scale depends on the size of the project and the scope of the chart.



A Gantt chart for building and moving in to a castle.

The critical path can be drawn, but it is not as salient as it is in the PERT chart.
Other columns can be added, such as start and end dates, resources needed, etc.
A Gantt charts emphasize task duration, start/end dates, and task overlap.
Both diagrams are very useful but can be tedious to keep up-to-date.
Both both can be extremely useful for planning and communicating.
They must be updated regularly. Save a dated copy every time you update.

I recommend using a simple spreadsheet or direct drawing editor, not a complicated task management software such as Asana. You want easy and direct editing on a single page.
(I disagree with Sommerville, 2015, in this regard.)



	B	D	E	F	G
5		February			
6	Tasks and Milestones	5	6	7	8
8	Observe Users	■	▒		
9	Analyze Current Systems		■	▒	▒

The diagrams provide well-established conventions.
If you can communicate time and task needs, you can gain power and control.

(LTCB story: 1 week, 2 weeks, 3 weeks, 2 weeks.)

Project Questions on Project Scheduling:

Draw a Gantt chart to explain to a student in a different group (and then to the class) how your group is using its time. This can be explained at the level of the entire group, or at the level of each individual team member.

Propose two fundamentally different order of activities that could be done on this project. Each of the two should have real strengths. Start with a brainstorm of all of the activities that need to get done. Generate a different graphical view of each order of activities.

Each Gantt chart should include indications of the critical path.

Hierarchical Task Analysis

Notes in part from: Shepherd, A. (2001). *Hierarchical Task Analysis*.
Annett, J. (2003). *Hierarchical Task Analysis*.
The reading is compiled in a document entitled “HTA_Materials.pdf”.

Notes in part from: Shepherd, A. (2001). *Hierarchical Task Analysis*.
Annett, J. (2003). *Hierarchical Task Analysis*.
The reading is compiled in a document entitled “HTA_Materials.pdf”.

“Any effort to improve human performance in a work or recreational setting must start by some understanding of what people are required to do and how they achieve their goals.” (Shepherd, p.1)

Task analysis is the process of gaining this understanding, and writing it down.

Hierarchical Task Analysis (HTA) is a methodology for describing the procedural knowledge that a person needs to do a task, and showing that knowledge in a tree-like structure.

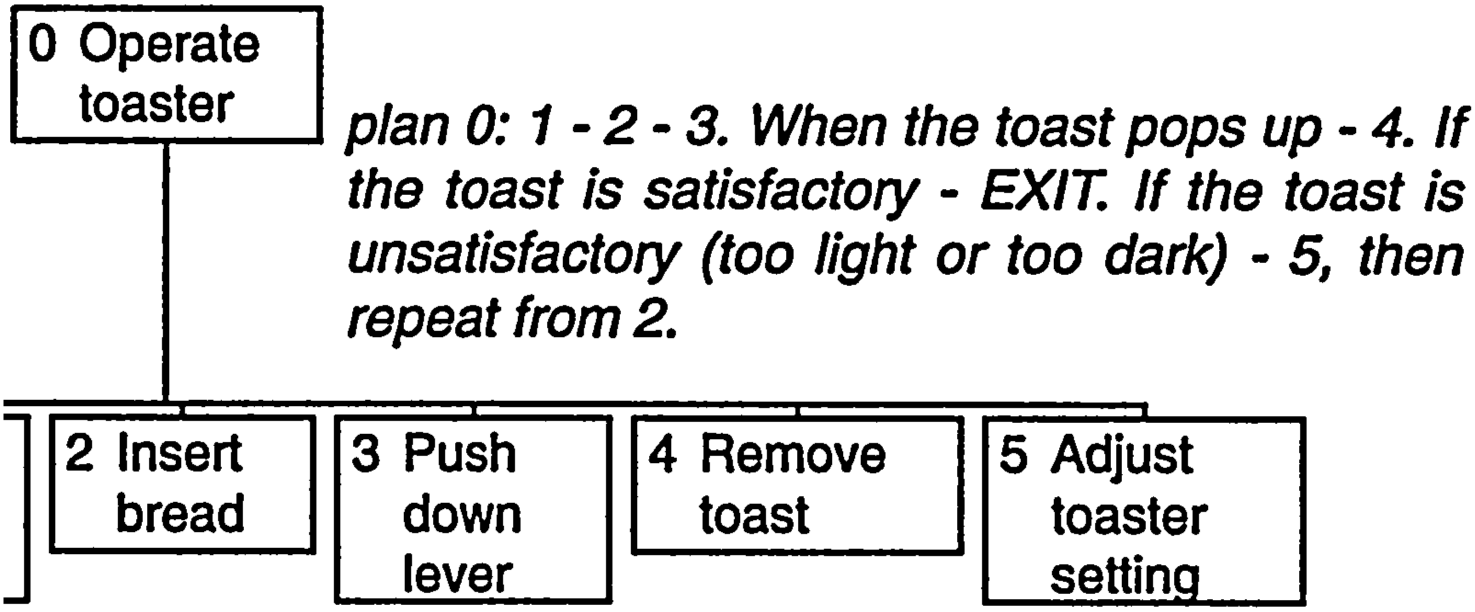
In HTA, tasks are represented in terms of hierarchies of *goals* and *subgoals*, using the idea of *plans* to show when subgoals need to be carried out.

‘In task analysis, it is always important to think of the reason why the task is carried out…. Thus, the task has a *purpose* or *goal* and *criteria* against which the outcome can be judged to be satisfactory or otherwise.’

Just as a task has a purpose, **the task analyst** (the person doing the analysis) also **has a purpose in doing the task analysis**. For example, the analyst might aim to understand the basic structure of a task, to design or improve a user interface, to create training materials, or to determine how people can coordinate activities. Knowing why you are carrying out the analysis affects how you should do the analysis.

For example, if you are trying to **use HTA to assist in the design of a user interface to support a specific task**, your analysis should perhaps (a) identify mid-level subtasks that reveal potential groupings of functionality and (b) take the analysis down to the level at which individual display and control decisions could be made, such as to specify exactly what information and inputs needs to be visible for a specific subtask. (Such as a thumb keyboard, and the text that you are typing.)

The *plan* is a very important part of the procedural knowledge. The boxes (?) tell you the actions to take, but the plans tell you the conditions under which you should take each action. Such as, if and when to take the action, and for how long.


0 Operate toaster
plan 0: 1 - 2 - 3. When the toast pops up - 4. If the toast is satisfactory - EXIT. If the toast is unsatisfactory (too light or too dark) - 5, then repeat from 2.
2 Insert bread
3 Push down lever
4 Remove toast
5 Adjust toaster setting

A plan for the steps required to make toast. (Figure 0.1 from Shepherd.) (The dash should be read as "then".) Note how the plan captures the conditions under which each step should be taken.

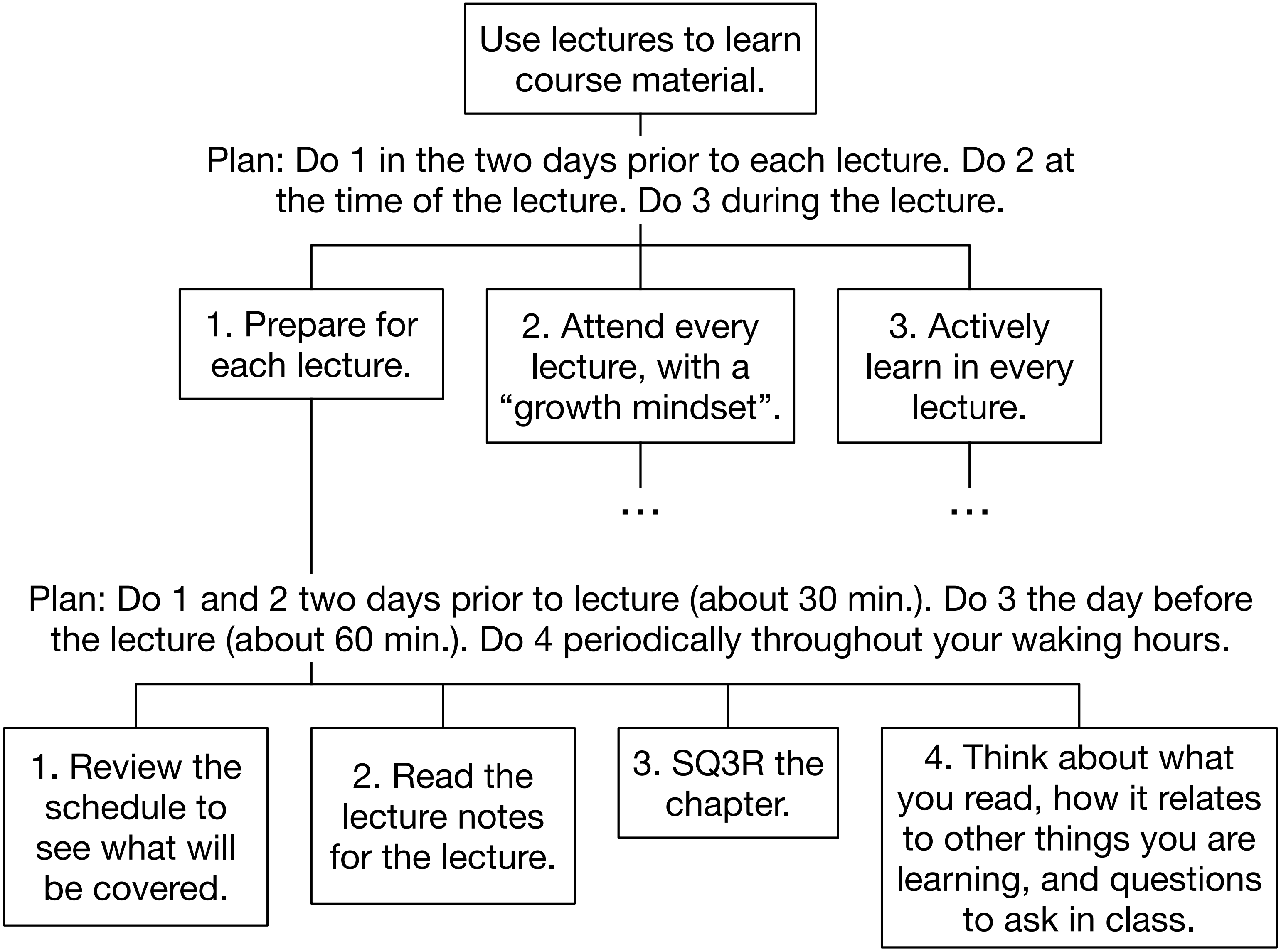
More elaborate versions of this procedural knowledge might include additional steps such as to scrape the burnt part off and serve it, or to monitor the toast while it is toasting.

Use the plan to capture the true expertise, the decisions of what to do next.

Three common types of plans in HTA:

1. *Fixed sequence* or *routine procedure*, such as “do this, then this, then this”.
2. *Selective rule* or *decision*. “If *x* is the case, do this; if *y* is the case, do that. These two types of plan are significant because they imply knowledge on the part of the operator. It may be simple *procedural knowledge*, or the plan may require extensive declarative knowledge of the environment….
3. *Time-sharing* or *dual task plan*. Two or more operations are pursued in parallel. That is, the superordinate goal cannot be attained unless two or more subordinate goals are attained at the same time. Not well-defined.

A final example:



Use lectures to learn course material.

Plan: Do 1 in the two days prior to each lecture. Do 2 at the time of the lecture. Do 3 during the lecture.

1. Prepare for each lecture.

2. Attend every lecture, with a “growth mindset”.

…

3. Actively learn in every lecture.

…

Plan: Do 1 and 2 two days prior to lecture (about 30 min.). Do 3 the day before the lecture (about 60 min.). Do 4 periodically throughout your waking hours.

1. Review the schedule to see what will be covered.

2. Read the lecture notes for the lecture.

3. SQ3R the chapter.

4. Think about what you read, how it relates to other things you are learning, and questions to ask in class.

A hierarchical task analysis (HTA) of an expert student’s procedural knowledge for using lectures to optimize learning. *SQ3R* is the active-learning technique *survey*, *question*, *read*, *recite*, *review*. The ellipses (“…”) show where the HTA would be further developed.

Sharp 14 - Usability Testing

Notes in part from Sharp et al. (2019). *Interaction Design*, Chapter 14.
Much of the following text are direct quotes from the chapter.

Essential Ideas in User Observation Studies.

You want to know: When my system is used in the real world, will people be able to use it, quickly and accurately, to accomplish their goals?

User observation studies are the "gold standard" of usability testing.
User observation studies are *empirical*, based on direct observation.
Analytic evaluation techniques also exist. Such as: Count the keystrokes and mouseclicks necessary to do a task. No users are needed.
But observing users trying to do tasks on your system is the gold standard.

There are Different Names for the Same Thing
"User Observation Studies" is a good phrase to use.
Also: "user testing" as in "testing the system with users", not "testing the user".
Remember: You are testing the product, not the user.

Remember the overall goal
Your overall goal is to create, as best as you can, the closest approximation to a real user doing a real task in a real-world task environment.
It is usually difficult or impossible, but you do your best.

14.3.1 Controlled Settings Involving Users

The studies control what users do, when they do it, and for how long.

User responses are observed and recorded. The observed data can be *quantitative* (such as speed and accuracy) and *qualitative* (such as user comments).

The primary goal is to determine whether an interface is usable by the intended user population to carry out the tasks for which it was designed. *The first question is: Could people use the device to do the task?*

A system design process produces a *software specification* (such as the Python functions that need to be written) as well as a *usability specification* (such as the time required to do representative tasks, and number of errors permitted).

A user observation study can determine if the usability specification is met.

A User Observation Study Looks for Causal Relationships.

Changes in Independent Variables
==== CAUSE ====>
Changes in Dependent Variables.

For example:
A UI design *causes* a system to be usable to do a set of tasks.
A button arrangement *causes* buttons to be clickable quickly and accurately

Experimental design is the creative process of trying to isolate and show causal relationships. This requires you to demonstrate that your findings are *valid*.

Validity

In experimental design, *validity* is the extent to which an experiment successfully isolates a causal relationship. Validity captures the extent to which an evaluation method measures what it is intended to measure.

Internal validity is the extent to which the experiment truly measures what it tries to measure; that is, within the context of this particular experiment.

External validity is the extent to which the experiment measures and shows something that is true about the world, beyond this one experiment.

Ecological validity is related to external validity. It is the extent to which the environment in which an evaluation is conducted influences or even distorts the results.

An example of validity: If you want to measure how long it takes someone to solve a Rubik's cube, you can measure their time by (a) using a stopwatch or (b) having someone count "one one-thousand, two one-thousand, ..."
You would feel more confident in the the stopwatch. It is more *valid*.

A challenge when learning how to run user-observation studies is developing your ability to make decisions that contribute to validity. This is not easy.

There are many potential “threats to validity”. You need to guard against them when designing and executing of a study; and when analyzing and drawing conclusions from the data.

Every aspect of a user observation study should be done with validity in mind, even if the word is never mentioned.

Some things to improve validity:

- Get your users into the “mental set” of a real user doing a real task.
- Present them with a real task, and motivate them to really do it.
- Vary the order of presentation of test conditions.
- Remove extraneous variables that may interfere with performance.
- Providing the same instructions to all of the participants.
- Train your users so they have the same expertise needed to start a task.
- Don’t help the user if they get stuck.
- Use many participants.
- Recruit your participants based on the kind of users you need.
- Screen your participants to make sure they have the qualities you need.
- Run your study in an appropriate setting (such as a quiet room, a loud room, or in everyday contexts “in the wild”).
- Much of Apple’s Guidelines for Conducting a User Observation Study.

How do you “prove” that your study has good validity?

You cannot. Also, you cannot *measure* validity.

What you *can* do is provide information that explains everything that you did to try to ensure that your study is valid.

That is one of the main goals of the Methodology section of a usability report.
Participants - Describe who participated in your study.
Setting - Describe the physical and social environment.
Materials - Describe the devices, instructions, and scripts.
Experimental Design - Describe the different conditions, and how presented.
Procedure - Describe the steps that you took to administer the study.

The information provided in these sections can helps to convince the reader of the validity of the study. Or can leave the reader unable to conclude whether the study was valid.

Informed consent

Participants must be told what they will be asked to do, the conditions under which data will be collected, and what will happen to their data when they finish the task. Participants must also be told their rights, for instance, that they may withdraw from the study at any time, for any reason, and with no penalty.

Informed Consent. Request this from your participants.

"Informed consent" refers to your participants permitting you to observe and record their performance, while knowing how their data will be used, and knowing their rights during the experiment.

Two important components of informed consent:

1. The participant can quit any time with no penalty whatsoever, including loss of access to services.
2. The participant will not be identified in the reporting of the data.

Informed consent is necessary for the ethical treatment of experimental subjects, and also to ensure that your participants can immerse themselves into the task without fear of retribution if they make a "mistake".

The goal of a user observation study is to get as close as possible to real users doing real tasks in a real-world task environment.

The Basic Idea in a User Observation Study

(From usability.gov)
(Also see "Apple's Guidelines for Conducting User Observations")

The basic idea in a user observation study is to figure out whether the actual intended users of your system will be able to figure out how to use the system, and will be able to use the system to do the tasks that it is supposed to support. Furthermore, the goal is to figure out whether the users will be able to do these tasks quickly and accurately, and with little confusion or frustration.

The general approach to running a user observation study includes:

1. **Identify the tasks that your system is designed to support.** Figure out how to present these tasks to the participants in your study (who are for now your representative "users"), ideally with no back-and-forth discussion between the participant and the experimenters (the people running the study).

   For example, create a worksheet of these tasks, and give it to the participants. Include blanks for the participant to fill in, such as the outcome of a task.

2. **Identify performance goals for the tasks.**

   For example: Using this system on a laptop, with a paper copy of the user manual next to the laptop, users familiar with laptops, but new to this system, should be able accurately figure out whether there are any beds available in any homeless shelters in Eugene, and where those beds are. It should take less than three minutes to do this task.

3. **Figure out how to record participant performance.** One common practice is to videotape the keyboard, screen, and voice of the user as they are using the system. When videotaping, it is important that you capture all of the user inputs, and all of the system outputs, in enough detail to understand exactly what the user saw, and exactly what commands they issued.

4. **Write a script that you follow during the study.**

5. **Recruit participants who are reasonable representative users**, such as users who work in the actual task environment. Ideally at least 5 users.

6. **Run the study.**
   a. Invite the participant in and explain what you are doing.
   b. Reassure the participant that you are testing the system, not them.
   c. Inform the participant that they can quit any time with no penalty.
   d. Explain that, if they get stuck, you will not be able to offer any help. They should just do the best they can, and let you know when they are done.
   e. Give the participants a list of specific tasks, and everything they need to do the tasks. The tasks should be clear enough such that the participants will know when the tasks are done.
   f. Tell them to start any time they want.
   g. You fade into the background, offering no help or guidance.
   h. When they are done, ask them debriefing questions such as how they did the task. Lastly, ask them if they have any questions. Thank them.

7. **Analyze the data.** Were most participants able to meet the performance goals? What problems did people encounter?

8. **Use what you learned** to update the interface design or implementation.