Notes on Reading Van Vliet (2008)

By Anthony Hornof

Last updated February 28, 2020

These are Anthony Hornof's notes from:

Hans van Vliet (2008) *Software Engineering: Principles and Practice*, 3rd edition, John Wiley & Sons.
Rosson, M. B., & Carroll, J. M. (2002). *Usability Engineering*. San Francisco: Morgan Kaufmann.

The notes were taken to (a) learn and organize an understanding of the material and (b) prepare lectures. The notes are not at all complete in that all chapters are not included here, and all of each chapter is not included. Some of the notes are copied directly from the book.

Table of Contents

Chapter 1 - Introduction

From 1955 to 1985, the percentage of total costs for computers shifted dramatically from hardware to software development and software maintenance. The amount of maintenance also increased relative to development.

Dramatic software failures cause all sorts of calamities. A few specific examples are offered.

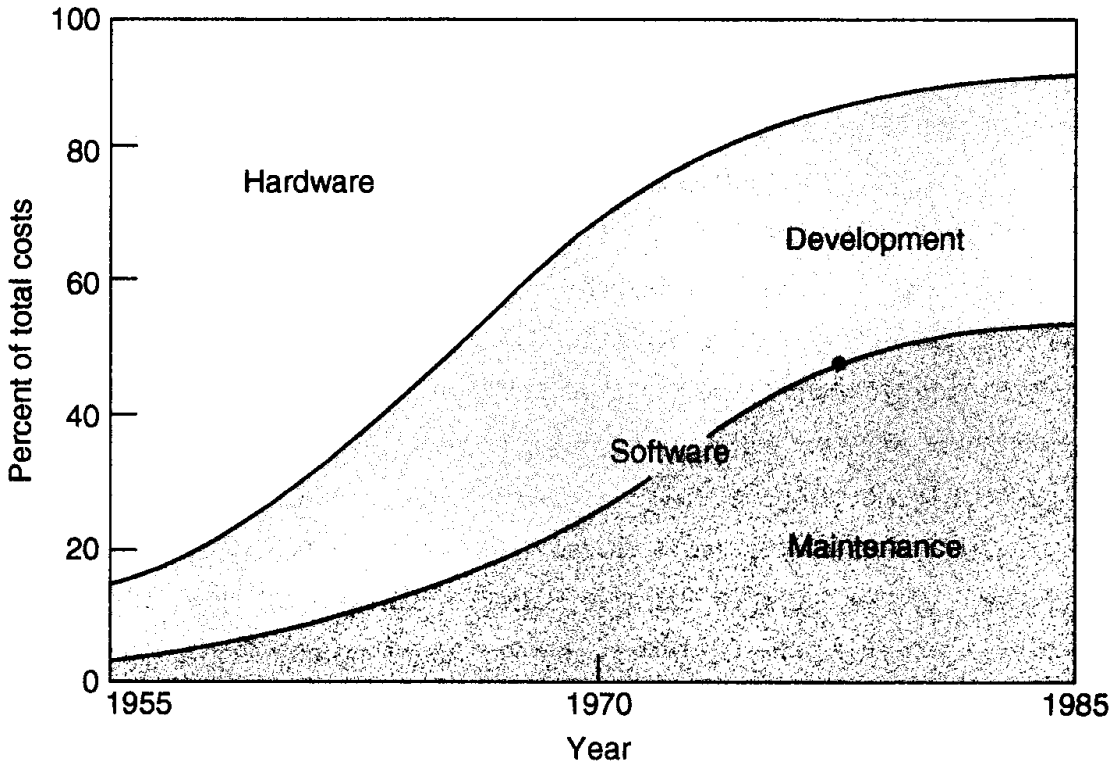


Figure 1.1 Relative distribution of hardware and software costs (*Source*: B.W. Boehm, Software Engineering, *IEEE Transactions on Computers*, © 1976 IEEE. Reproduced with permission.)

1.1 **What is software engineering?**

The methodological process of building reliable, robust, efficient, accurate, and useful computer programs. "The engineering metaphor is used to emphasize a systematic approach to developing systems that satisfy organizational requirements and constraints."

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.

Characteristics of the field of Software Engineering:

- It is concerned with "large" programs.
- It tries to master complex problems: people, processes, programs. A project must be broken up and managed.
- Software (s/w) evolves. For example: Y2K, Euro, internet, new CPUs, etc.
- Building and maintaining s/w is very time-consuming. "The last 10% takes 90% of the time."
- S/W development is a people problem. Efforts must be coordinated. People must be managed.
- Software development is partly a user interface (UI) problem. You must study people doing their work, understand the context of work, and provide user documentation and training.
- Developers are not domain experts. They usually lack factual and cultural knowledge of the target domain.

S/W does not wear out the same way as physical products. It breaks differently.
"90% complete" syndrome: s/w "almost finished" for endless amount of time.

1.2 Phases in the Development of Software

Process Model:
Requirements engineering => Design => Implementation => Testing => Maintenance

But it is rarely a truly linear process.

Phases of S/W Development:
1. Requirements engineering: Includes a feasibility study. Produces a requirements specification.
2. Design: Decompose into modules or components, and interfaces between. Wrongly seen by some programmers as getting in the way of the "real work" of programming.
   Architecture: global description of a system.
3. Implementation: Start with a module's design specification, or "spec". The first goal should perhaps be a well-documented, well-organized program, not necessarily an efficient one.
4. Testing: Not just a phase that follows implementation.
5. Maintenance: Keep the system operational after delivery.
6. Project management: Deliver on time and within budget.

System Documentation: Project plan, quality plan, requirements spec., architecture description, design documentation, test plan.
Start writing your documentation early.
User documentation: Task-oriented, not feature-oriented. (Write it first!)

Breakdown of activities: 20% coding. 40% requirements and design. 40% testing. 40-20-40 rule.

A software development lifecycle is a planned sequence of activities that are undertaken when building a computer program.
It is called a software life "cycle" in part because it is cyclic, activities loop back around.

Dramatic software failures:

Adrian 5 rocket blew up, $0.5 billion loss. Overflow converting from 64-bit float to 16-bit int.

Therac-25 radiation machine delivered radiation doses 100x the intended. Patients died. Software interlock replaced electromechanical interlock, and failed.

London Ambulance Service, Computer-Aided Dispatch. Bidder was not qualified for project. Dispatched ambulances outside of familiar areas. Memory leak crashed system.

1.6 Quo Vadis - "Where are you going?"

Not yet a fully mature discipline. Trends appear, such as agile methods, the integration of COTS (commercial off-the-shelf) software, open source software. But there is no magical solution to the difficulty of software development.

"There is no silver bullet."

Frederick Brooks "No Silver Bullet: Essence and Accidents of Software Engineering" landmark software engineering paper in 1987:

> "Of all the monsters who fill the nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these, one seeks bullets of silver that can magically lay them to rest.
>
> The familiar software project has something of this character (at least as seen by the non-technical manager), usually innocent and straightforward, but capable of becoming a monster of missed schedules, blown budgets, and flawed products. So we hear desperate cries for a silver bullet, something to make software costs drop as rapidly as computer hardware costs do.
>
> But, as we look to the horizon of a decade hence, we see no silver bullet. There is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement in productivity, in reliability, in simplicity."

Chapter 2 - Introduction to SWE Management

Some reasons that software is delivered late:
- Programmers did not accurately state the status of their code.
- Management underestimated the time needed for the project.
- Management did not allow enough time for project.
- Project status not made clear.
- Programmer productivity was lower than hoped.
- Customer did not know what they wanted.

Information Planning - the meta-project planning process; how this project fits into other projects and systems within the organization.

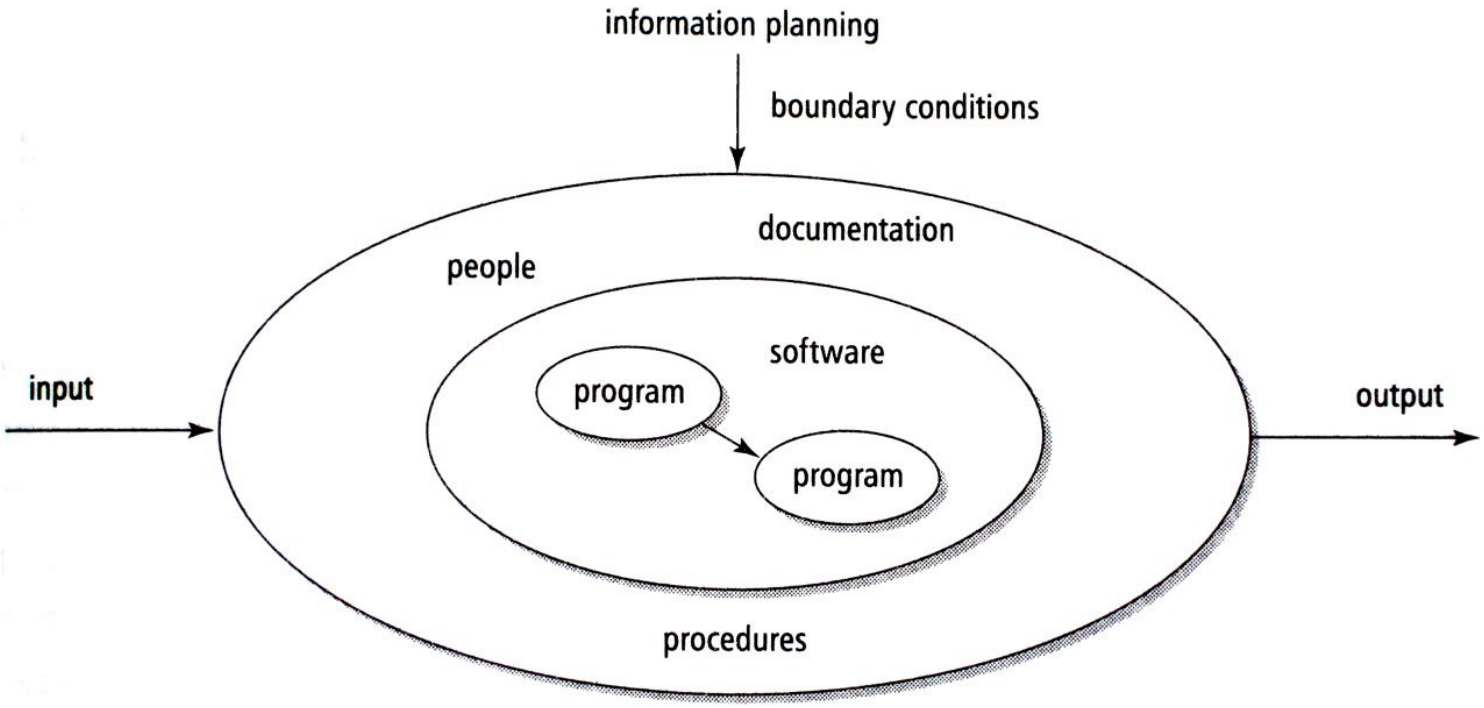


Figure 2.1 The systems view of a software development project

2.1 Planning a S/W Dev Project

Project Plan: A document that provides a clear picture of how the project will proceed, to both the customer and development team.

Major constituents of a project plan are:
1. Introduction - background, goals, deliverables, team members, summary.
2. Process model - activities, milestones, deliverables, critical paths.
3. Project organization - relationship of the project to the rest of the organization, project team roles, reporting structure, how stakeholders members will interact.
4. Standards, guidelines, procedures - configuration control, quality assurance, etc.
5. Management activities - status reports, resource balancing, etc.
6. Risks
7. Staffing
8. Methods and techniques
9. Quality assurance
10. Work packages
11. Resources
12. Budget and Schedule ***
13. Changes
14. Delivery

2.2 Controlling a SWD project

Control must be exerted along the following dimensions: time, info, organization, quality, money

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.

Chapter 3 - The Software Lifecycle Revisited

Chapter 1 introduced a simple model of the software life cycle. Phases included:
Requirements engineering, design, implementation, testing, and maintenance.
In practice, it is more complicated.

In this view, major milestones generally relate to documents, such as:

- Requirements spec.
- (Technical) specification
- Computer programs
- Test report

Document-driven. The client signs off. (I saw this at DRT Systems.)
Does not readily accommodate maintenance, or going back to previous phases.
Can have excessive maintenance costs. (World Tax Planner.)

Overview: The waterfall model model does not really take maintenance into account. Evolutionary models do. The model should ideally also take into consideration product families and long term business goals.

Choose a process model for your project. Making it explicit helps all of the stakeholders to anticipate what is going to happen, and helps you to gain control over the development process.

The Waterfall Model

A slight variation from the Chapter 1 model.
Emphasizes the interaction between adjacent phases, with testing in every phase.
Verification & Validation to compare what is needed to what is generated.
Verification: Building the system correctly.
Validation: Building the right system.
Emphasis on getting the client to "sign off" on each phase before proceeding.
Problem: It is difficult to anticipate all requirements. The validation in each phase may allow for slight adjustments, but not a wildly different direction.

The waterfall model, like Escher's waterfall on the cover of the book, is unrealistic.

The strict sequence of activities is not obeyed.

For example, you may do perhaps 50% of the design in the "design" phase, 33% in the "coding" phase, and then another 15% in the testing phase. (Figure 3.2)

Designers and programmers cross boundaries all the time.

But we teach it! And it is followed! Why? It is understandable. It is a good first approximation of the phases and the general order in which they are followed.

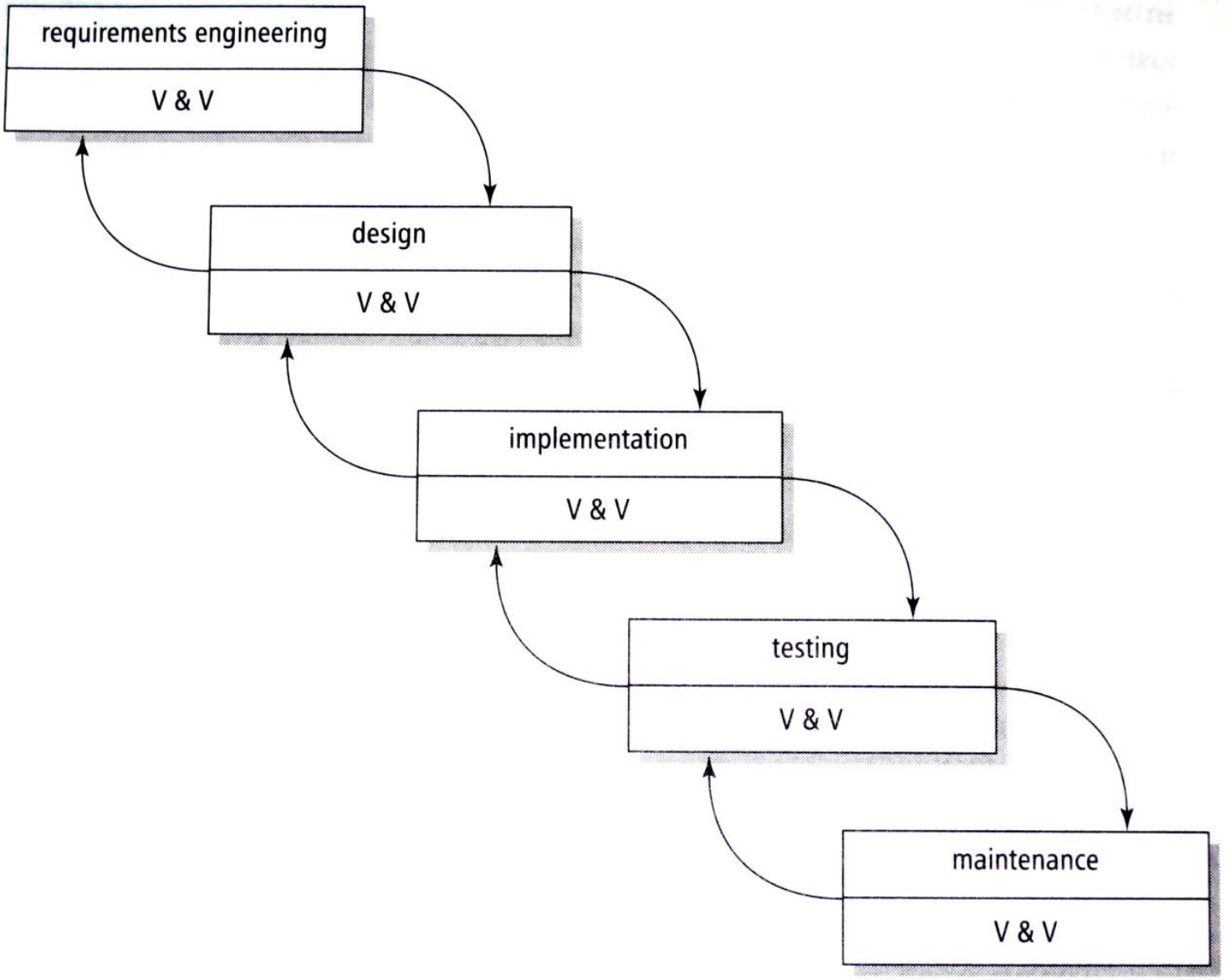


Figure 3.1 The waterfall model

Agile Methods (added 9/30/10)

Agile (able to move quickly and easily) methods resign themselves to the fact that the world is fundamentally chaotic, and cannot always be controlled. (Though, on the other hand, there are many natural forces that prevent complete entropy, at least in the near term, such as gravity, species survival, or people achieving goals).

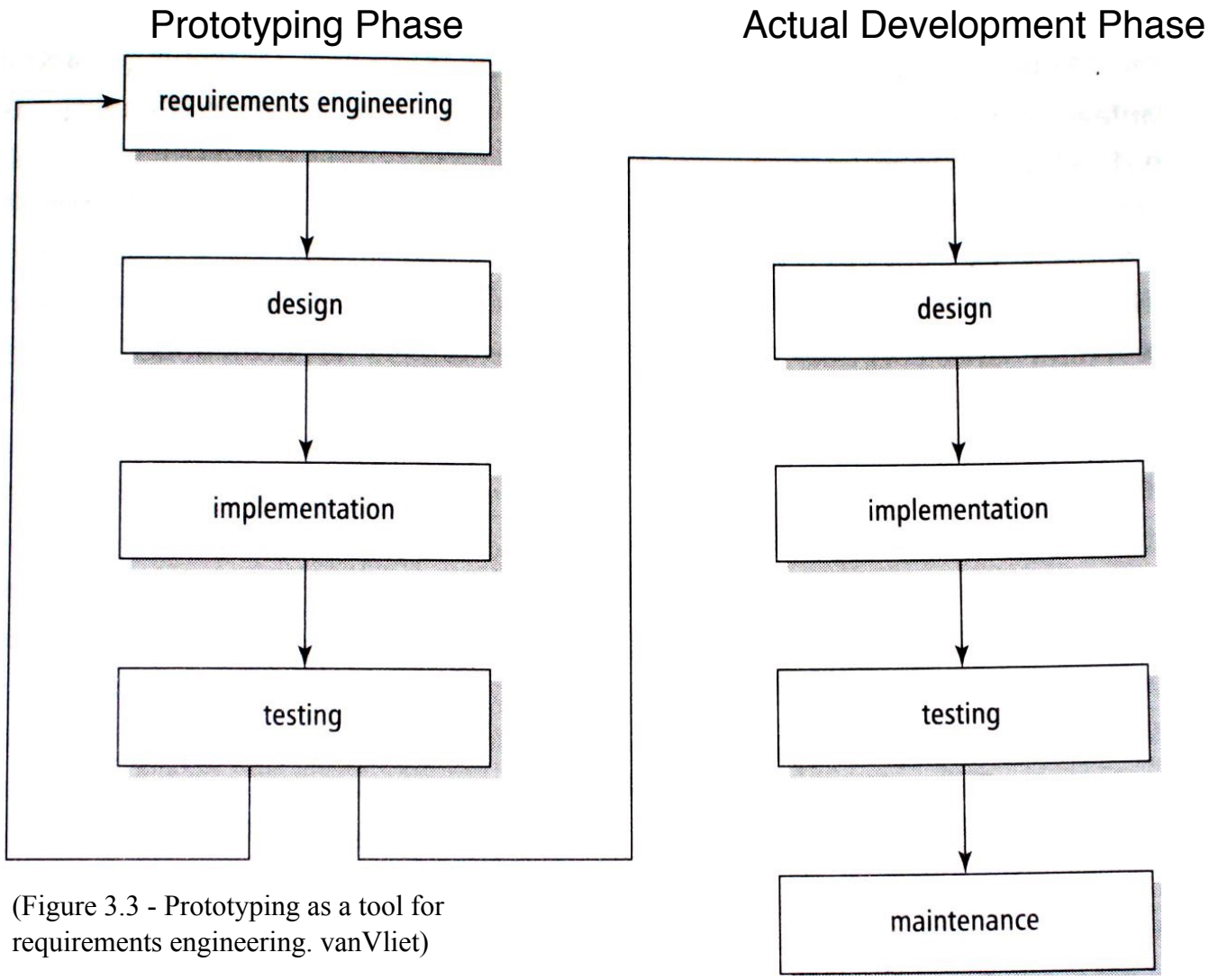
Agile methods emphasize:

1. People over processes.
2. Working software over documentation. (Some will think "Hooray!").
3. Collaboration over negotiation.
4. Responding to change over following a plan.

Similar to (the formerly popular approach of) RAD (rapid app. development).

"Extreme Programming" is an agile method, with two programmers working side-by-side on the same computer, like pilot and co-pilot. "Pair programing." (If you do this, take turns.)

Prototyping



(Figure 3.3 - Prototyping as a tool for requirements engineering. vanVliet)

A **prototype** is a working model of a proposed software system, or parts of such a system.

Often constructed with higher-level languages or tools that are constrained in what you can build, and that produce inefficient programs. (html is pretty limited, for example)

The functionality is typically limited.

Prototyping is extremely useful for addressing the problem that customers have a very difficult time expressing their requirements precisely.

Give the user a UI prototype, let them try it out it in the intended context, and see if the functionality accurately reflect the true system requirements BEFORE a huge investment in building a real system.

Potential problem: The client may think that this *is* the real system. Maintain user expectations.

"Throwaway prototyping" - No code is carried over (in Figure 3.3).

"Evolutionary prototyping" - More common, at least some code is re-used.

Pros and Cons of prototyping in Figure 3.2 (p.58)

Particularly useful when the user requirements are ambiguous, and when the UI is important.

Customer can get carried away with new features. You have to keep them focussed on what is truly needed, and limit the number of iterations.

Incremental Development

The system is produced and delivered to customer in small pieces, with each piece providing a set of independent functionality.

Essential functionality is delivered initially.

Rapid Application Development

Incremental development with "time boxes": fixed time frames within which activities are done.

Must be able to sacrifice functionality for schedule.

Requires, close, rapid communication cycles between developers and with stakeholders

Peer-to-peer communication between users and developers

Intense user involvement (and commitment) in negotiating requirements and testing prototypes Joint Requirements Planning (JRD) and Joint Application Design (JAD),

“Cutover” phase in which the system is installed (and abandoned?).
Best suited for small team development and modestly sized projects.

3.5 Maintenance or Evolution?
Can maintenance be thought of as a single box at the end of the lifecycle?

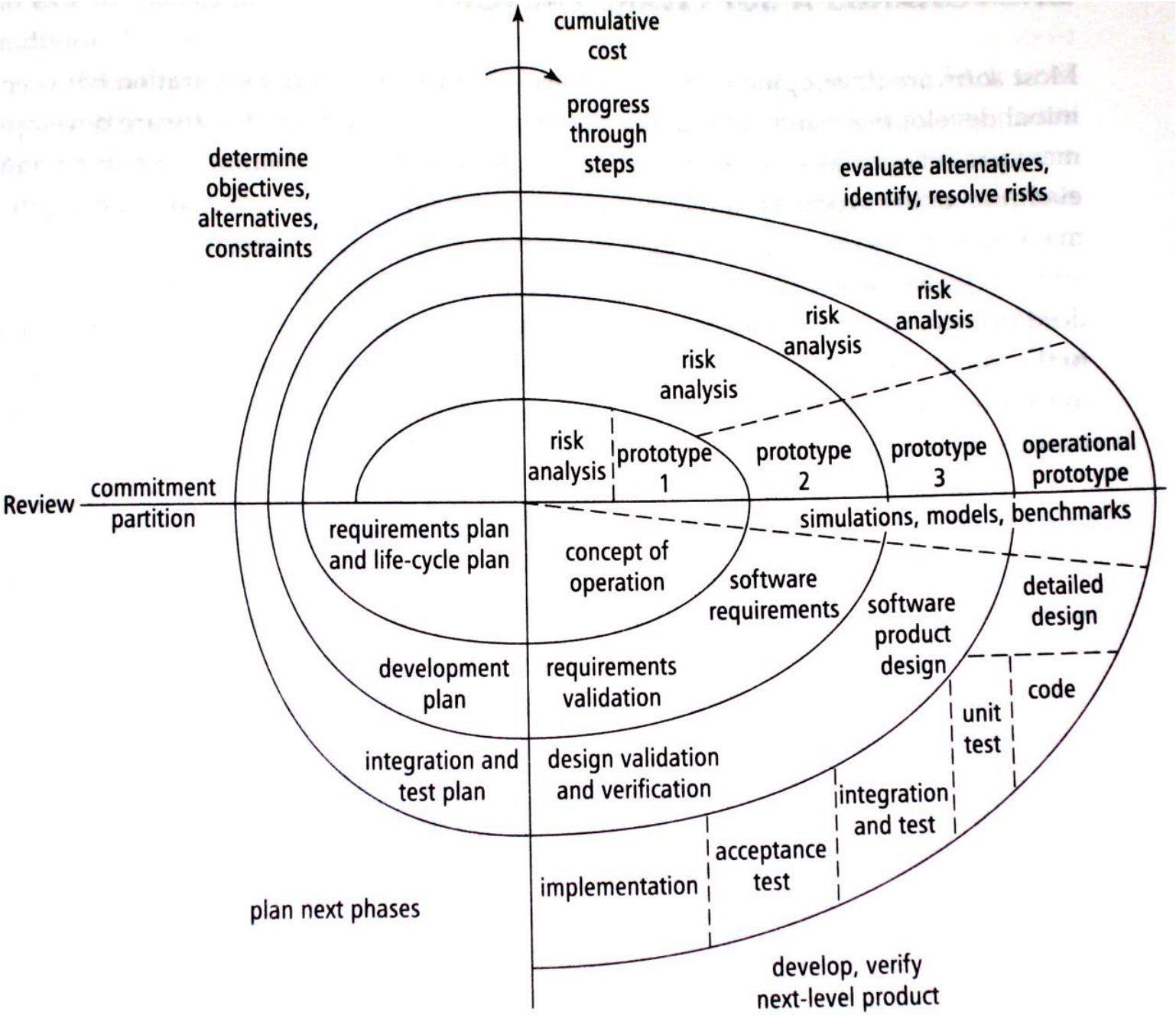
The laws of software evolution:
The law of...
1. ... continuous change: A system that is being used undergoes continuous change.
2. ... increasing complexity: A program that is changed becomes less structured. Entropy (disorder) sets in.
3. ... program evolution: Measurable aspects of the program (loc, number of modules, functions, etc.) may seem to grown in spurts because of short-term pressure. But in fact they can really only grow at a steady, linear rate, because after the spurt you need to go back and “clean up the code” and update the documentation, etc. (Figure 3.8 on p.74)
4. ... invariant work rate: Adding more staff does not increase the speed of development. Large systems proceed at a saturated rate. (Windows software is routinely released years late.)
5. ... incremental growth limit: A system can only grow to a certain size, or at a certain speed (clarify with Stuart) before major problems set in.
6. ... continuing growth: If you build it, they will come... and want bugs fixed, and new features...

(There are two more laws on on p.73)

Software engineering is in the news daily. “Windows Is So Slow, but Why.pdf”

3.6 The Spiral Model

Considered to be the idealized model for s/w development.
The conventional teaching is: waterfall bad, spiral good.
But much more complex, and more difficult to anticipate specific milestones and deliverables.
Big emphasis on risk assessment.



Subsumes the other process models discussed thus far.

3.9 Summary

Look at the trajectory we have followed:

Waterfall to prototyping to incremental to agile to spiral.

Perhaps increasing complexity, but also increasing realism.

In all cases, you are trying to model—or simulate—the processes necessary to develop a system, to gain control over the process.

Chapter 8 - Project Planning and Control

Notes on reading Van Vliet (2008) by Anthony Hornof.

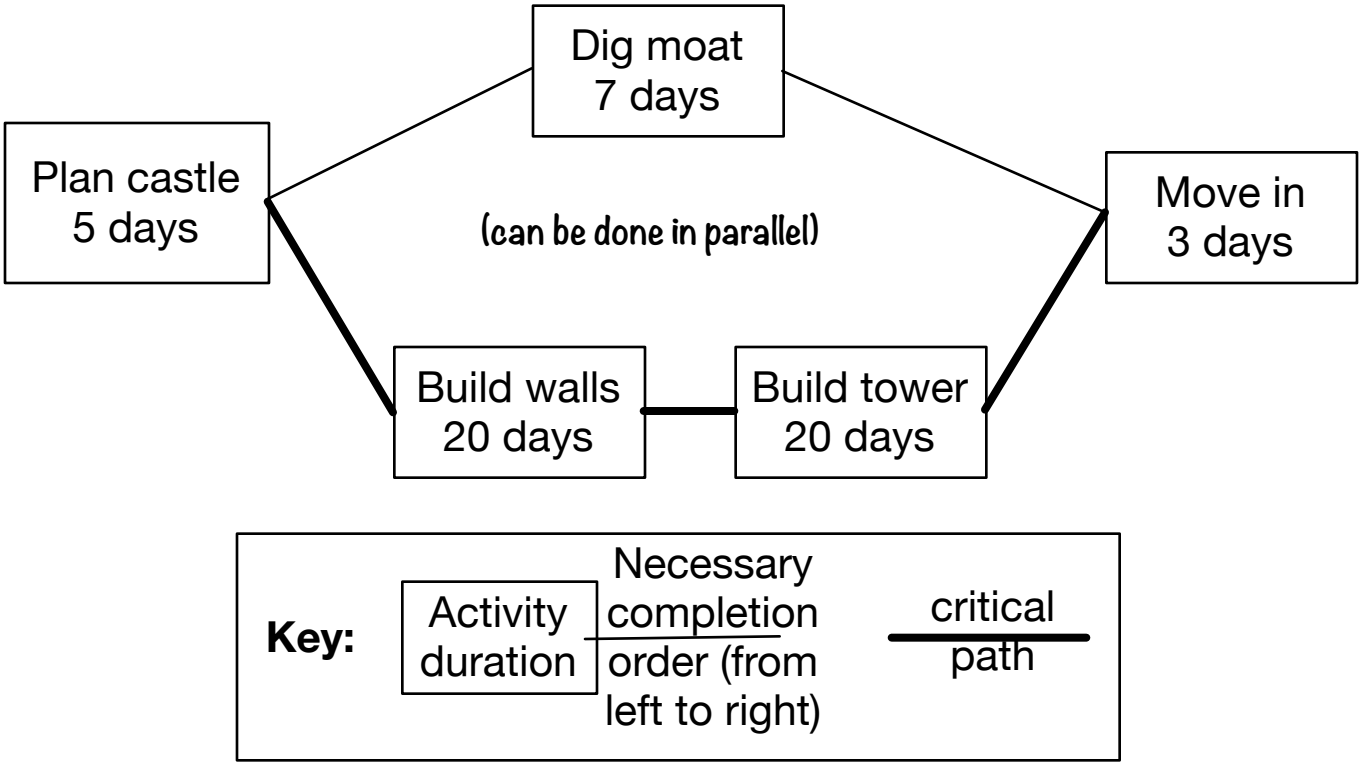
Recall that software engineering is the process of gaining and maintaining control over the products and processes of software development.

- "Intellectual control" ...
- "Managerial control" focuses on gaining and maintain 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)

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).



PERT charts emphasize the *critical path*....

Project Planning Terminology

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.

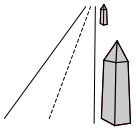
You should always be working on activities that are on the critical path.

In the example above: 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 only delays the project if it is on the critical path.

Slack time is the time that a task can be delayed without delaying the project.



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).

Gantt Charts (Timelines)

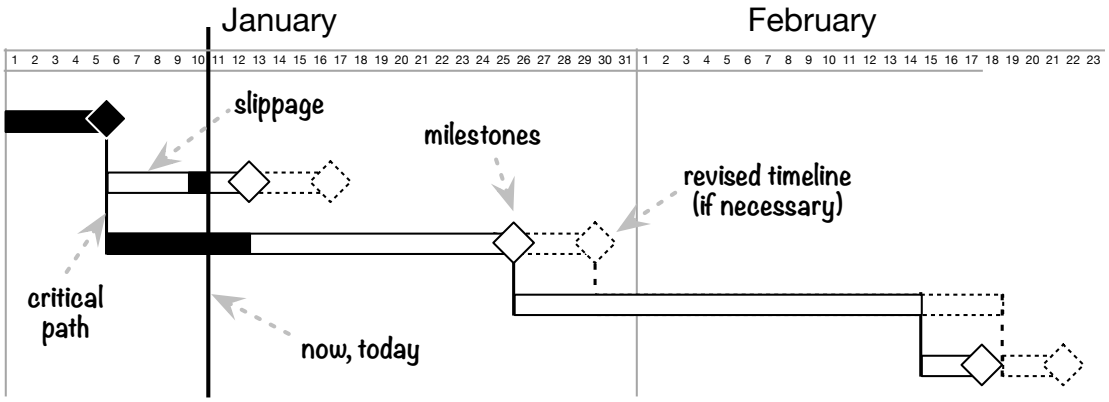
Named after Henry Gantt. (He developed them around 1910 to maximize the productivity of factory workers.)

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

Time always moves from left to right.

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

Task	Staff	Days
Plan Castle	QN	5
Dig Moat	KG	7
Build Walls	PC	20
Build Tower	QN, KG	20
Move In	All	3



The critical path can be drawn, but it is not as visually emphasized as it is in the PERT chart.

Other columns can be added, such as start and end dates, resources needed, etc.

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 are extremely useful for planning and communicating.
They must be updated regularly. Save a dated copy every time you update.

I recommend using a spreadsheet or direct drawing editor, not a complicated task management software such as Asana. You want easy and direct editing on a single page.

	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.)

Chapter 9 - Requirements Engineering

Notes on Reading Van Vliet (2008), continued. By Anthony Hornof.

Requirements describe what the system will do.
Design describes how the system will work.

Requirements engineering (or requirements analysis) is 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 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. (Figure 13.1, in S/W Testing)

Example: From Mom's work at Tektronix. Major consulting company came in and only met with managers. Managers did not know how the export specialists would split orders across invoices to accommodate bureaucratic needs in foreign customers. Did not get implemented. System was deployed. Export specialists explained the need. Consultants told them to just put it onto one order. "We cannot sell to this customer unless we can split it across invoices." They had to go back and re-implement major portions of the system.

How do you get it right?

Requirements engineering...

1. Elicitation - understanding the problem.
2. Specification - describing the problem.
3. Validation - agreeing upon the problem.

All three are critical.

Identify the Problem

A good statement of the problem is critical. Separate the problem from the proposed solution. This helps enormously to convince the client that you understand their needs.

See examples of Problem Statements below.

Elicitation Techniques

Ask: Interview users about the work and their tasks, not the system.

Task analysis: A technique to obtain a hierarchy of a goal-oriented set of activities. Work (or play) involves people, tasks, artifacts, context. Record and document these aspects. Watch, observe the users. Get them to "think aloud".

Scenario-based analysis: Generate usage scenarios. These are stories that tell brief narratives of different stakeholders using the system. The Project 1 handout has very brief usage scenarios. The sample SRSs on the course web page describe stakeholder scenarios. These should be sample stories of real users doing real tasks. They put the system into a context that helps to capture and convey some of the explicit and implicit requirements.

Ethnography: Submerge yourself into the foreign culture and learn its subtle ways.

(**Form analysis:** Study the paper associated with the current system.)

(Natural language descriptions.)

Derivation from an existing systems: This is certainly done in market-driven software development.

(Business Process Redesign.)

Prototyping - Ask: What is a software life cycle model that would lend itself to requirements elicitation?

Market-Driven versus Customer-Driven

"Unfortunately, most requirements engineering techniques offer little support for market-driven software development." (p.208) - Agree or disagree?

An example would be the challenges in developing an open source carpool program that would be useful to a range of different organizations. What is/was the problem? How did we think to solve the problem?

The conventional approach in software engineering is to discuss requirements engineering as the process of identifying, documenting, and validating user requirements.

This makes a huge assumption that users and stakeholders are available to participate in the process.

Market-driven software.

Book example: Develop a 'generic' library application rather than for a specific library.
COTS: commercial off-the-shelf.
Where does a good open source piece of software fall? Market-driven or customer-driven?

Specification
You need to organize the document. Section 9.2 on Requirements Documentation offers example structures.
Functional versus nonfunctional is a typical breakdown.
Functional: Services provided, or how inputs are mapped to outputs.
Nonfunctional: System properties, constraints, and qualities. (External interface requirements, performance requirements, design constraints, and software system attributes.)

Requirements document should be
* **Correct**. Solving the right problem in the right way.
* **Unambiguous**. At some level, to all stakeholders. Define all terms. Must be well-written.
The serial order problem, solved with overviews, organization (TOC, lists), some repetition.
See Slide (3).
* **Complete**. Should address all aspects of the system functionality and constraints.
* **Consistent** (internally). Should not contradict itself.
* Ranked for importance. Can be explicit or conveyed with words such as "must" vs. "should."
* **Verifiable**. Can objectively determine if each requirement is met. Not just "fast", "easy".
* **Modifiable**. *Requirements will change. You will always need to update your document.*
* **Traceable**. The origin of each requirement should be documented.

Conclusion: The requirements describe what the system should do and define the constraints on its operation and implementation.

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 10 - (Software Design) 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.)

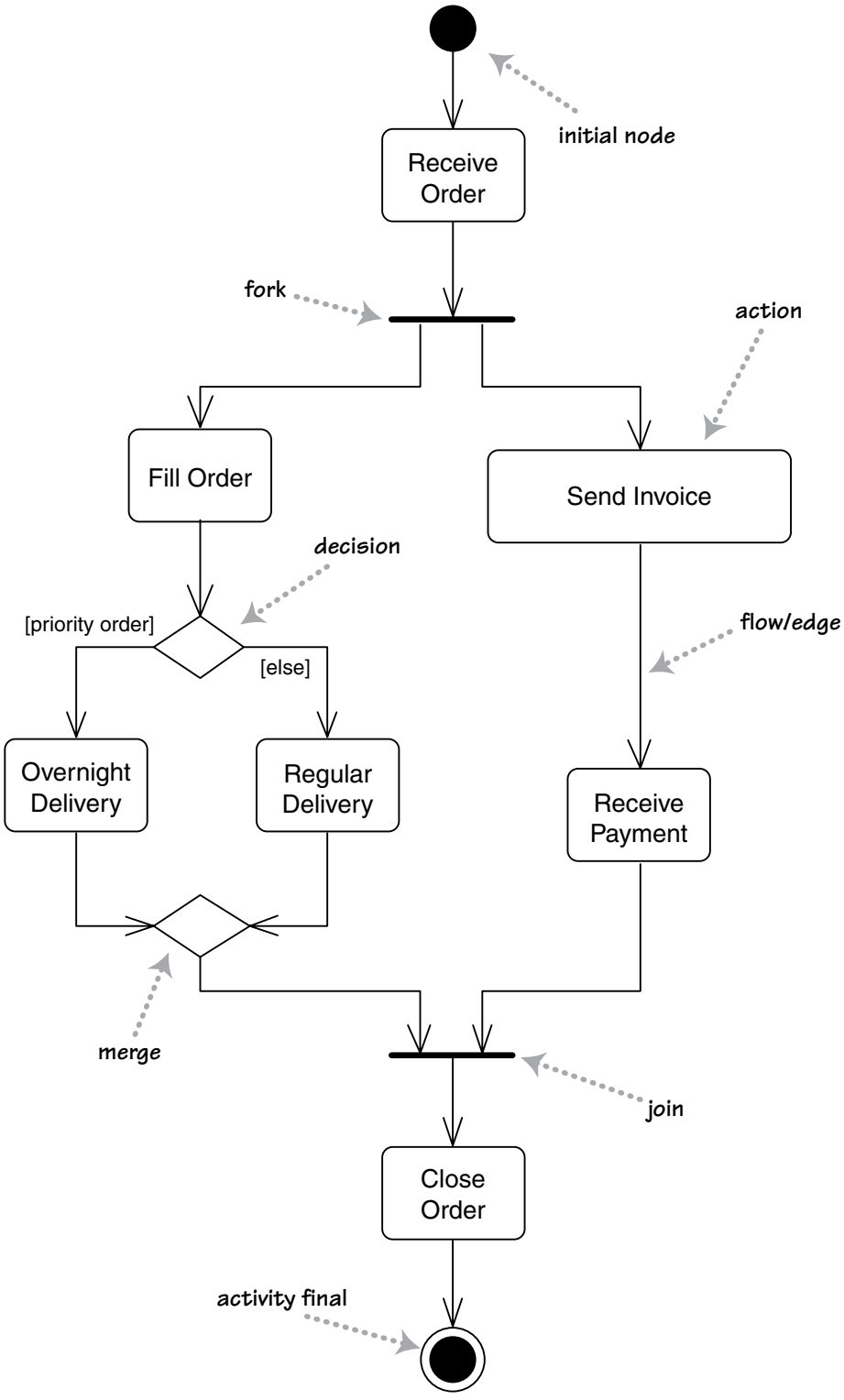
The models are **static** or **dynamic**.
Static show structure.
Dynamic show behavior.

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.

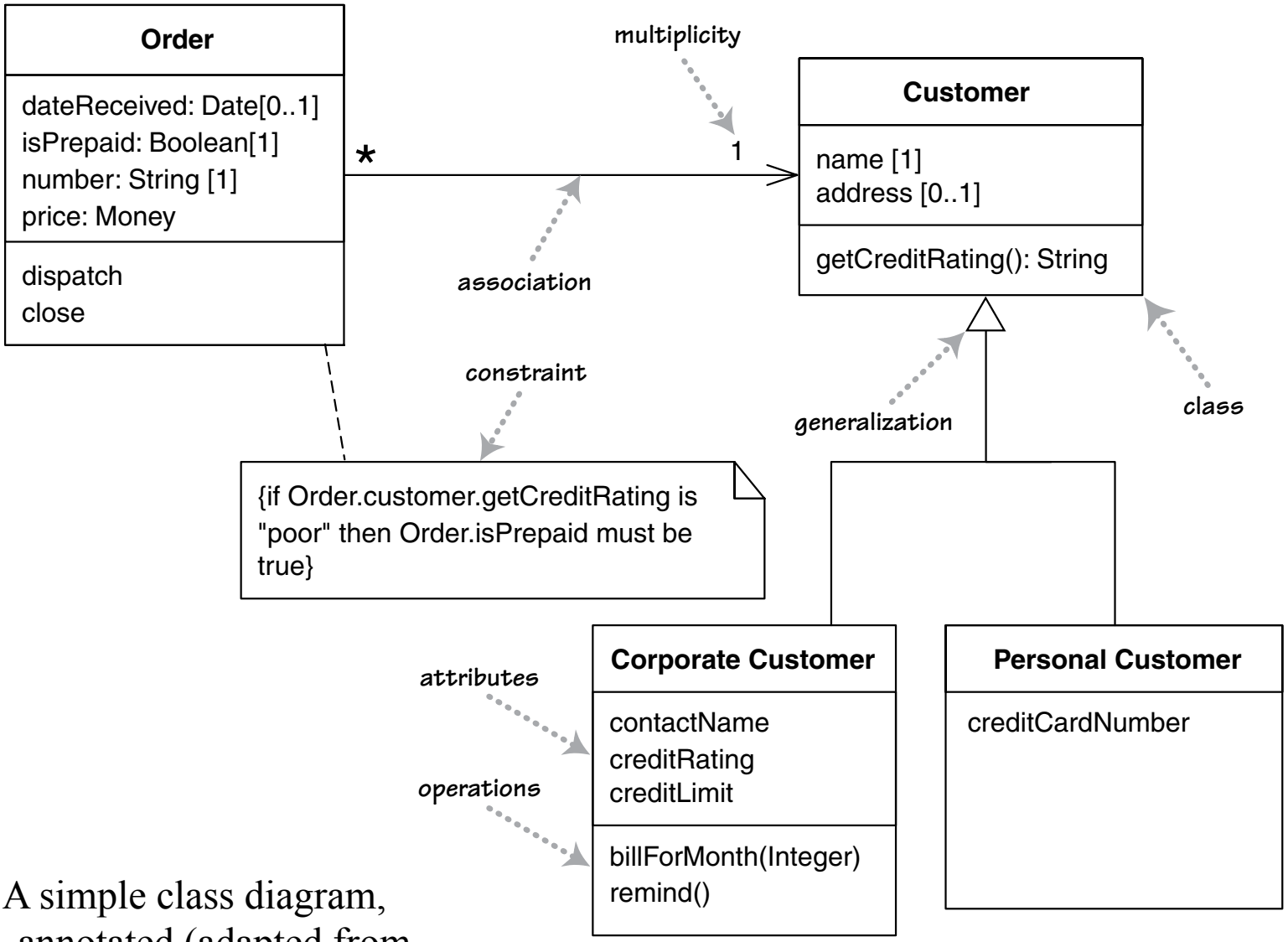
Sequence diagrams: Dynamic. Describe how groups of objects collaborate in some behavior. Show the sequence of object interactions

(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.

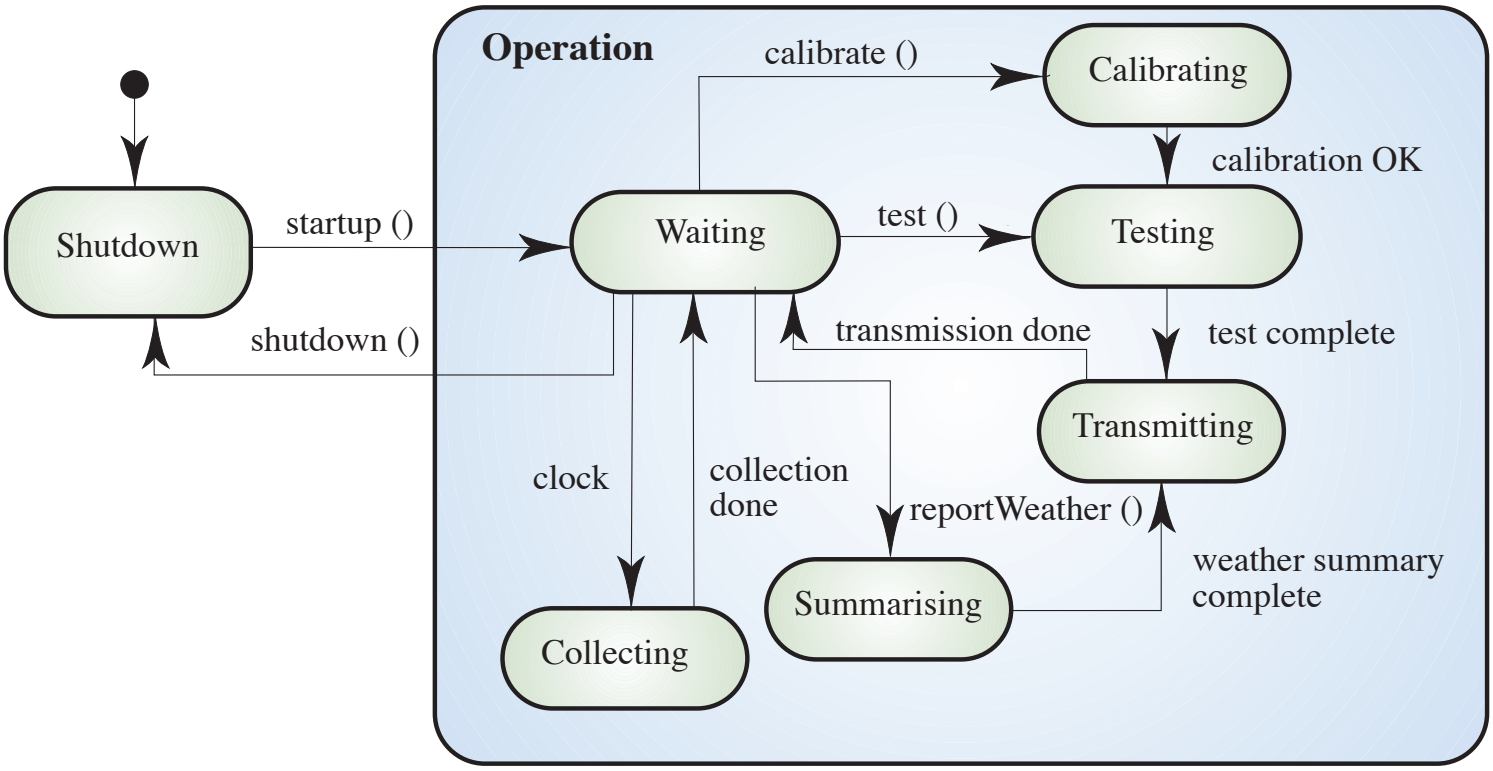


A simple class diagram, annotated (adapted from Fowler, 2004, Figure 3.1)

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

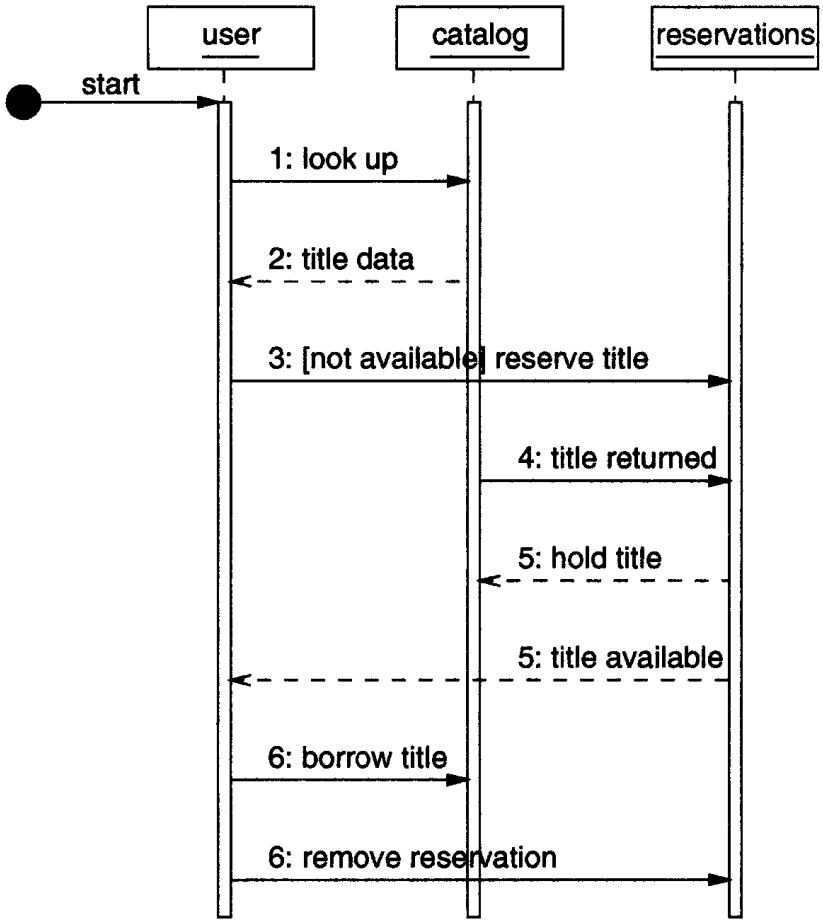
A *dynamic* model illustrates the restricted states of an object or system.
The ovals are states and the arcs are events that cause the state to change.
Can have hierarchies of states, introducing abstraction.
Permit stakeholders to understand the of dynamic aspects of the system.



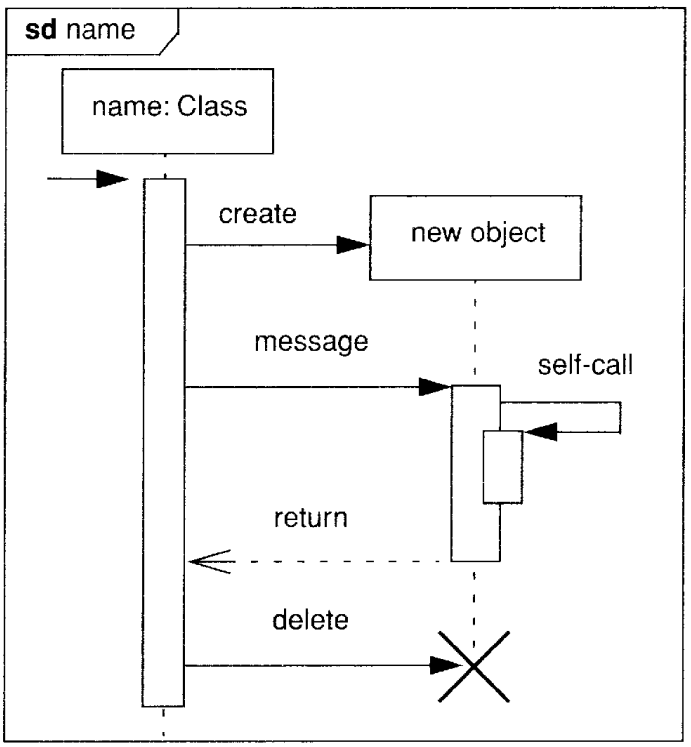
A state diagram for a weather station that, every five minutes, collects data, performs some data processing, and transmits this data.
(From Sommerville, 2000, Software Engineering)

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.

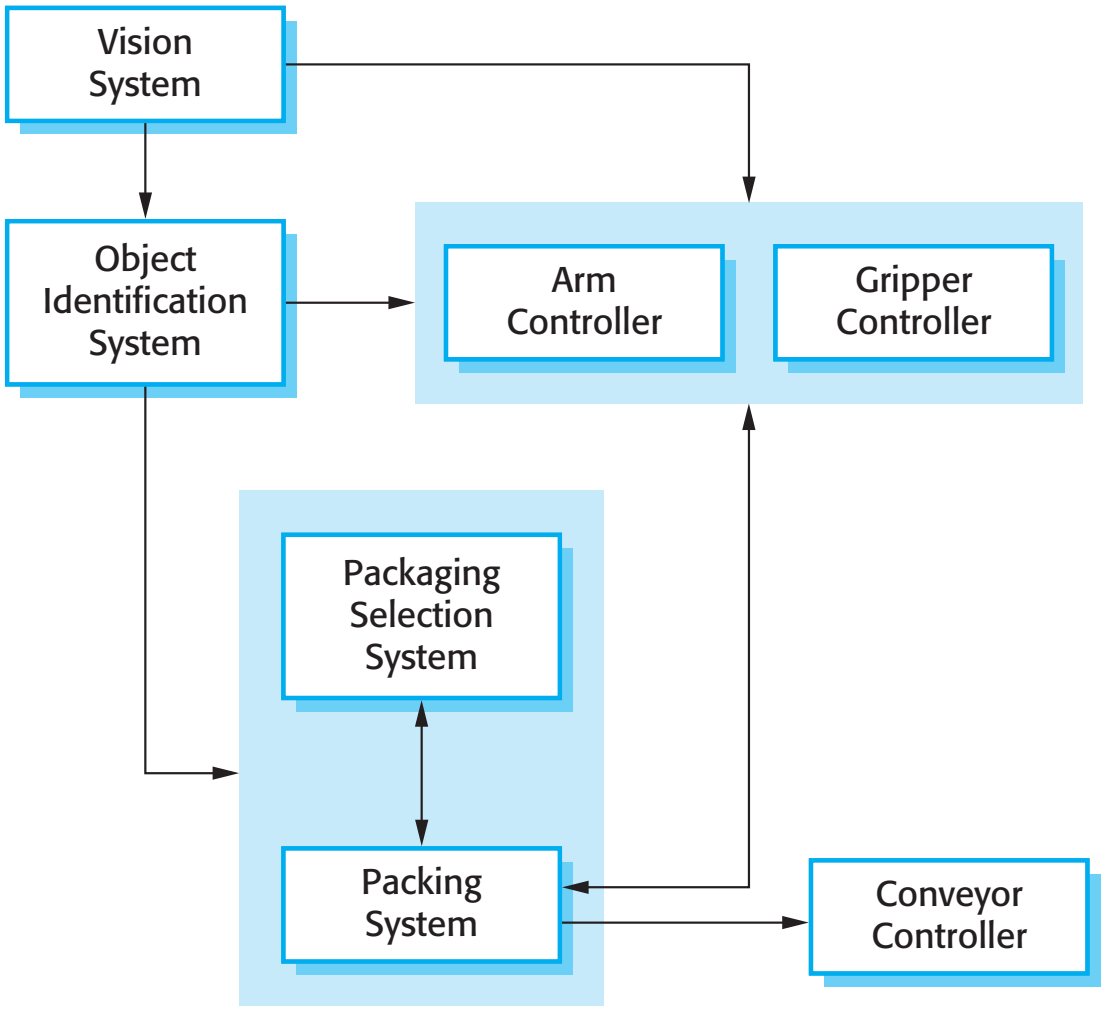
Software design modeling is an important aspect of software engineering, the study of the full lifecycle of writing the code that run on computers.

Chapter 11 - Software Architecture

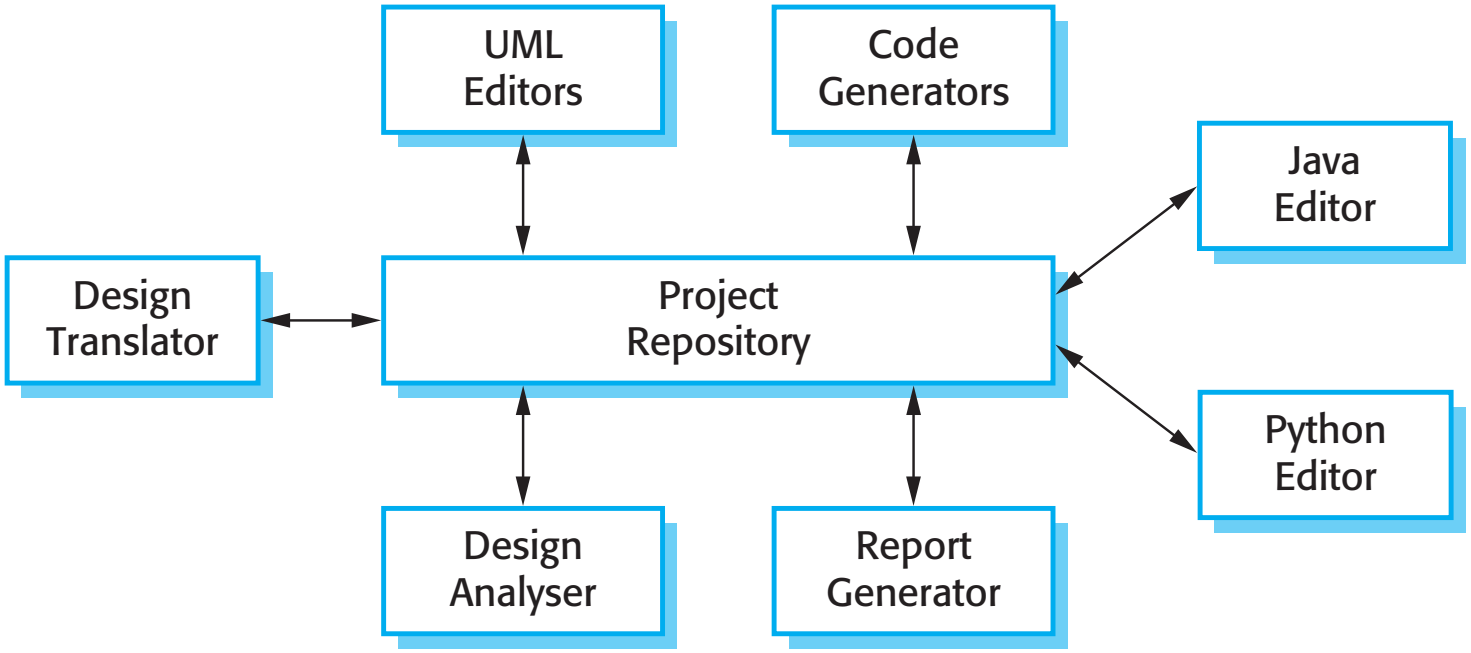
In computer science:
"Hardware architecture" refers to the the design of the logic circuits in the chips.
"Software architecture" is what we are talking about today.

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.
Typically a static (not dynamic) diagram. "Module" implies static.
Relates to modular programming.

"The design process involves negotiating and balancing functional and quality requirements on the one hand and possible solutions on the other hand." (Van Vliet p.290)



The software architecture of a packing robot control system (Sommerville).



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

Software architectures serve three purposes (from van Vliet):

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

The traditional view is that the requirements determine the structure of a system. It is increasingly recognized that other forces influence the architecture and design.

1. Organizational inertia. If you develop a really good code base for interacting with Google maps, you're less likely to switch to Yahoo maps.

2. Architect's expertise. When I have students use a MVC architecture on Project 1, they almost all use the same on Project 2, even if other architectures are superior.
3. Technical environment. If a Skype API is implemented, and it provides all of the telephony functionality that you need, you will incorporate it rather than build your own module.

The software architecture process is about both making and documenting design decisions. Not all of them. But all of the major decisions. This is why I have you explain your design rationale.

One of my goals is to get you to build into your design process a consideration of alternatives, including alternative architectural designs. See Figure 11.1.

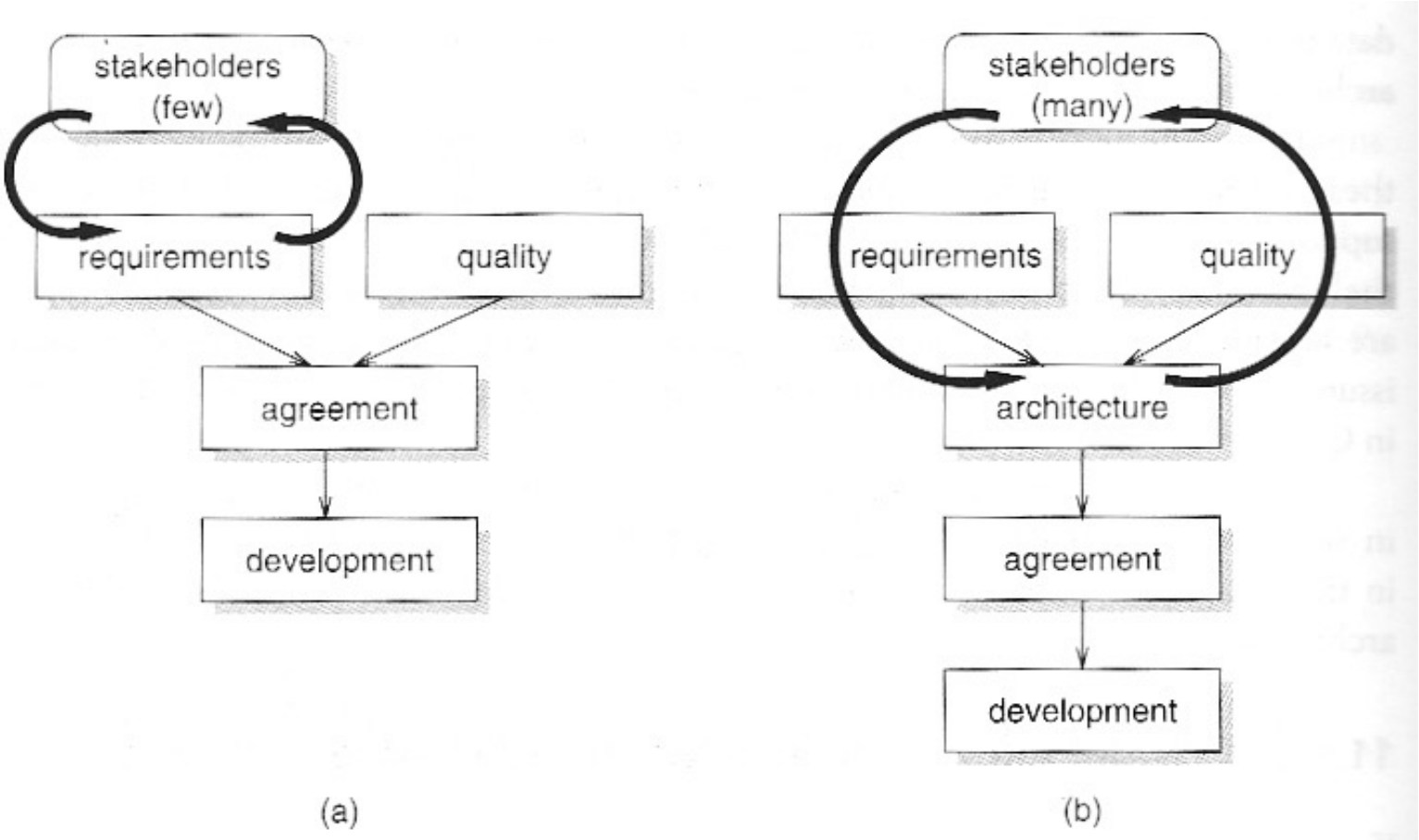


Figure 11.1 Software life cycle (a) without and (b) with explicit attention to software architecture

Architectural Views

Terms:

- Stakeholder: A person or group with interests in a system.
- View: A representation of a whole system (from the perspective of a stakeholder).
- Viewpoint: The purpose for, or the techniques for constructing, a view. Provides the syntax of the view.

Three classes of viewpoints:

- Module viewpoint - *static* views of the system. Examples: Decomposition (boxes of boxes), class diagrams. Boxes are components and lines some kind of relationship.
- Component-and-connector viewpoints - *dynamic* views of a system Boxes are components or processes, and lines represent some sort of temporal order. Example: flowchart.
- Allocation viewpoint - some relationship between the system and the environment, such as a work assignment chart.

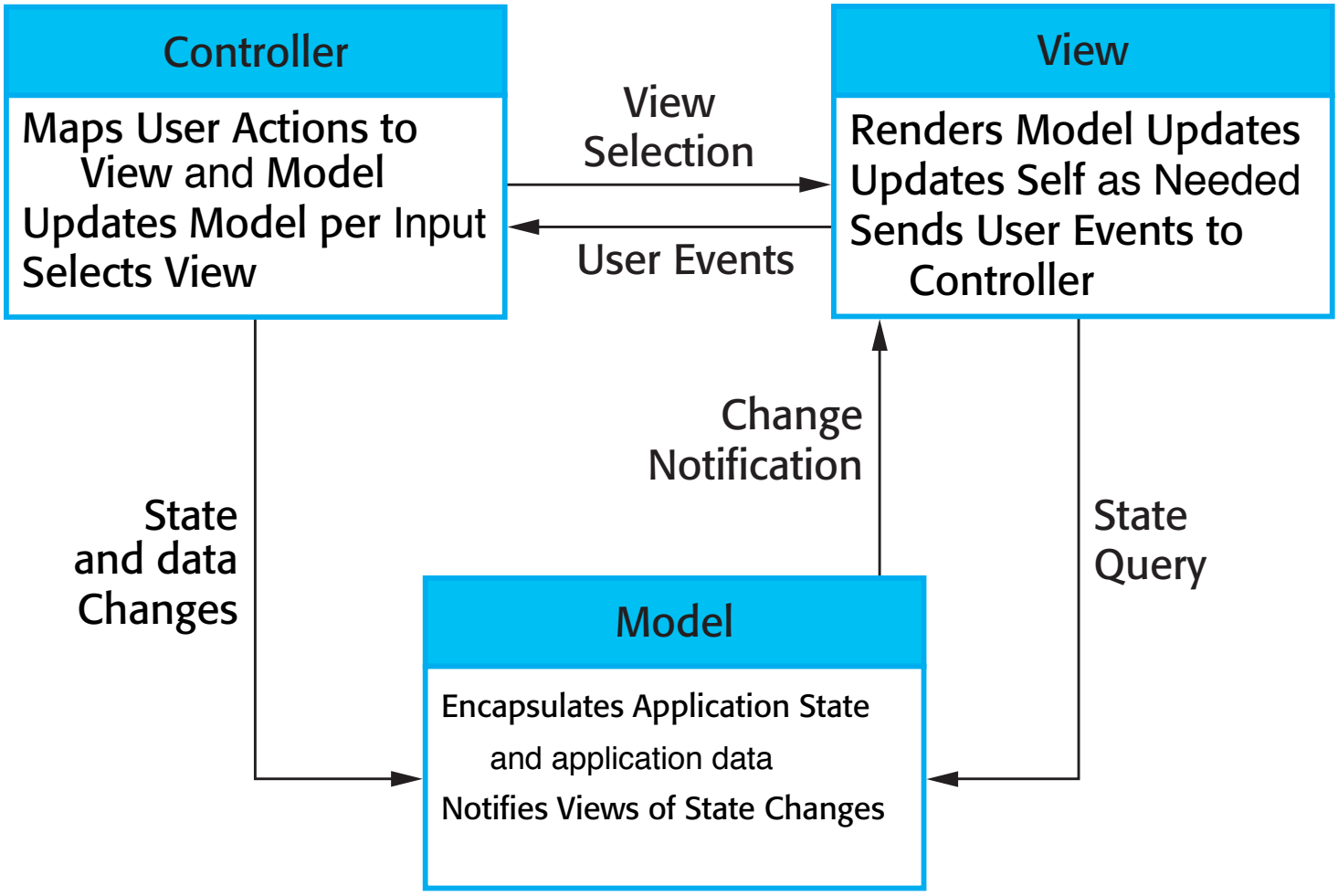
Van Vliet is trying to find abstractions and classifications that can encompass, tie together, and even prescribe a bunch of different architectural designs.

He wants you to learn the architecture, and also the situations in which you would use it.

"Design Patterns" in software architecture (See Section 10.3) refer to a book by Gamma et al. (1995) that discusses solutions to recurring problems in software construction.

"Design Patterns" in building architecture refer to an approach to design approach and book ("A Pattern Language," 1977) by Christopher Alexander. It is embraced by some architects, mocked and dismissed by others.

Design patterns in building architecture are overplayed a bit in this edition. VanVliet picked some questionable building design patterns for his examples, such as "high buildings make people crazy." I do not believe that the field of building architecture would generally endorse this design heuristic. It can be risky to derive ideas across disciplines without fully understanding the source discipline's analysis of the ideas, such as by Saunders (2002). A Pattern Language. Harvard Design Magazine, Winter/Spring 2002(16).



The model-view-controller (MVC) software architecture (and archtype design pattern). (adapted from Sommerville)

vanVliet tries presents a bunch of different architectures in the context of a recipe that incorporates the problem, the context, and the solution. This is how software patterns are used in computer programming, and they are used more precisely than in building architecture. The rough idea came from Alexander, but the precise implementation for computer programming came from Gamma et al. Building architects now point to Gamma et al. as validation of their patterns, which remain loose and imprecise.

Let me just explain what are the architectures that he is talking about. I will leave you to read the book to see the recipes. These diagrams are no longer in the book.

KWIC-Index Example

A classic example from Parnas (1972) though not thought of as an example of "software architecture" until 1996. (I think that I got this 2nd detail from the footnote at the bottom of p.259 of the second edition.)

The problem: You want a list of all of the titles in the collection such that all of the titles are included once for every word in the title, with every word featured once as the first word. And you wanted it sorted by the first word of every title regardless of its reordering. This way, you can efficiently find all of the titles that have a certain phrase in it by just going to that one part of the list.

So "Introduction to HCI" and "HCI Handbook" with both be next to each other:

...
Handbook HCI
HCI Handbook
HCI Introduction to
Introduction to HCI
to HCI Introduction
...

The input is a list of titles. The output is a sorted list of duplicated and shifted titles.

How do you do it? Perhaps have students draw them on the board, and try to critique.

Four tasks must be accomplished: Read input, determine shifts, sort shifts, write output.

Modular decomposition dictates one module per task.

But how do they communicate, coordinate, and share data?

These are architectural decisions.

Design #1. Shared Data - Main program and subroutines

Multiple modules share data structures.

Input into one table. Shift into another, keeping a reference back to the original title. Sort into a third table, drawing from the shift, but keeping a reference back into the original titles.

This is somewhat akin to a design in which you input the data into a single data structure, and then manipulate all the data within that structure.

Common approach. All modules need access to all data. Decisions about data representation have to be made very early. Procedural interfaces also have to be decided early.

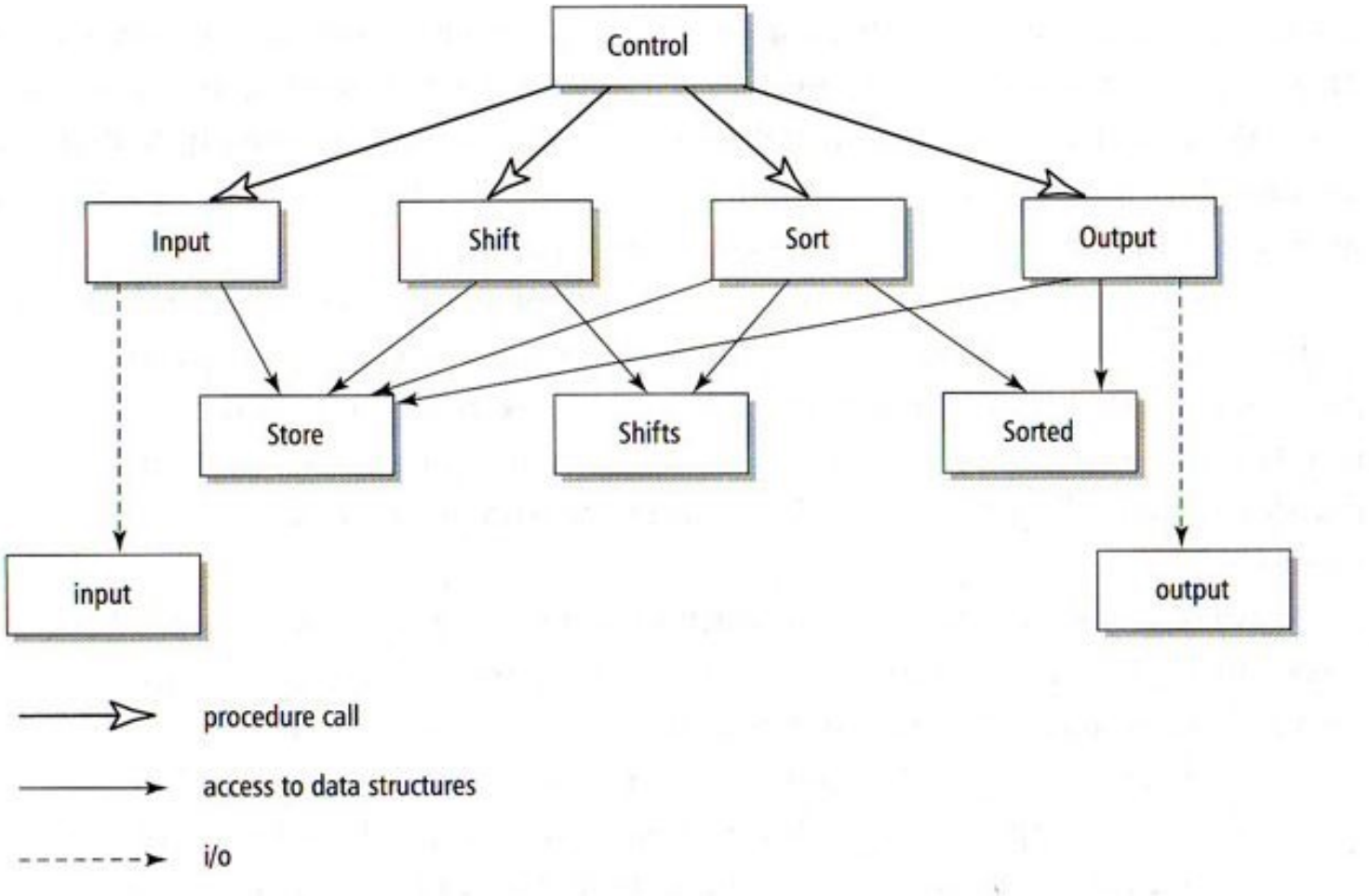


Figure 10.1 Main-program-with-subroutines solution of the KWIC-index program

Design #2. Abstract Data Type

Rather than all modules having an explicit agreement about the exact structure of each table, the modules have a shared understanding about the general, or abstract, way that the data will be stored. Such as a set of numbered lines, with each line have a set of numbered words.

The procedures access and manipulate these abstract data types.

For example: lines() returns the number of lines, and words(r) the number of words in line r.

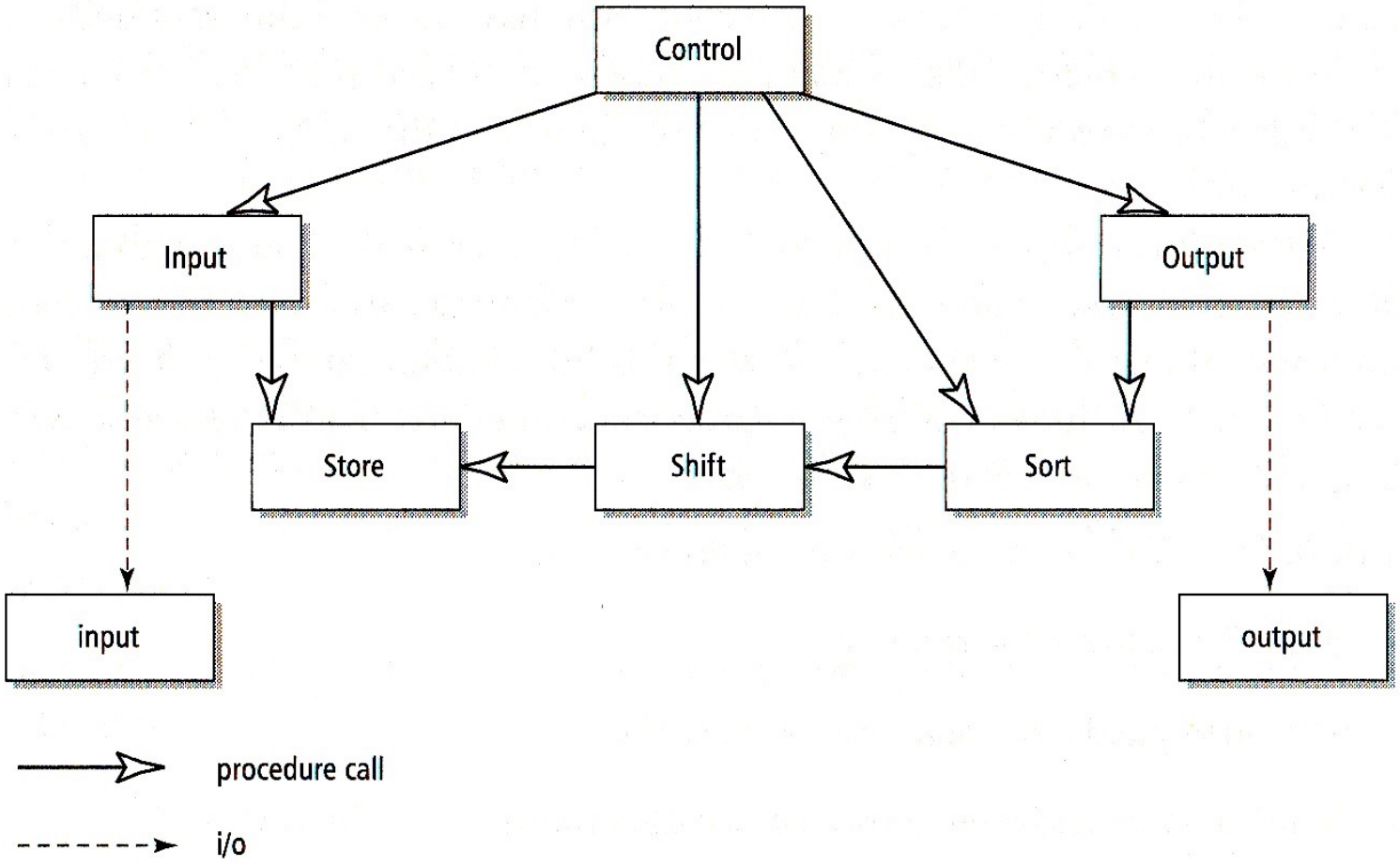


Figure 10.2 Abstract-data-type solution of the KWIC-index program

Design decisions made locally.

It is relatively easy to change the data representations and algorithms, but hard to change the functionality.

To not output the lines that start with "the", you would either (1) add a module between sort and output (which would waste time because the shifts have already been made) or (2) change the shift module to skip over the lines (but the module starts to move further from its simple functionality).

Design #3. Implicit Invocation

Event-based. Each module processes a line, or a batch, and deposits into a store. The next module down the line is listening for that event and when it happens, processes the new data.

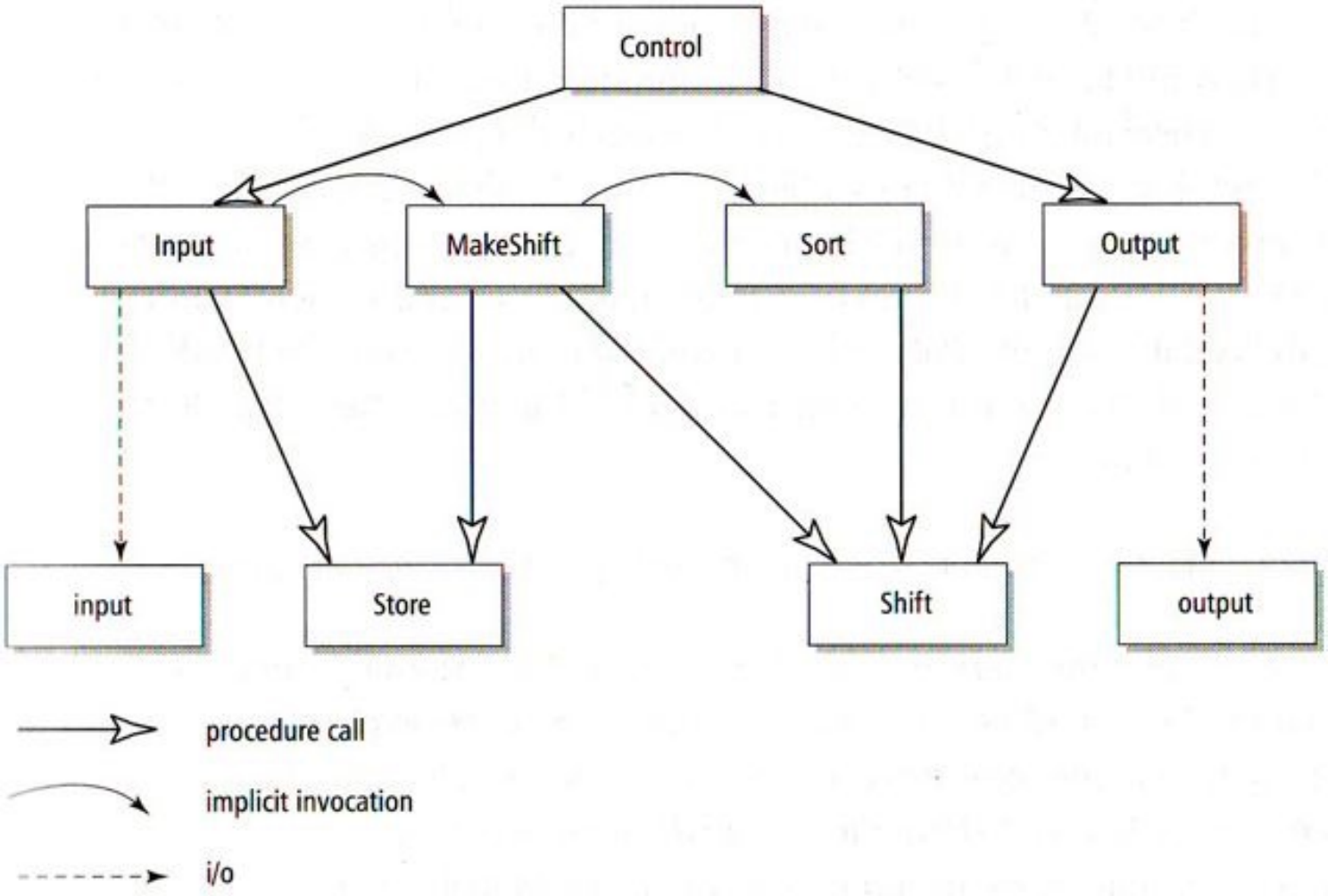


Figure 10.3 Implicit-invocation solution of the KWIC-index program

This can perhaps handle changes in functionality better.

Design #4. Pipes and Filters.

Separate program, or filter, for each. Batch processing.

The final program, Unix: Input < input | Shift | Sort | Output > output

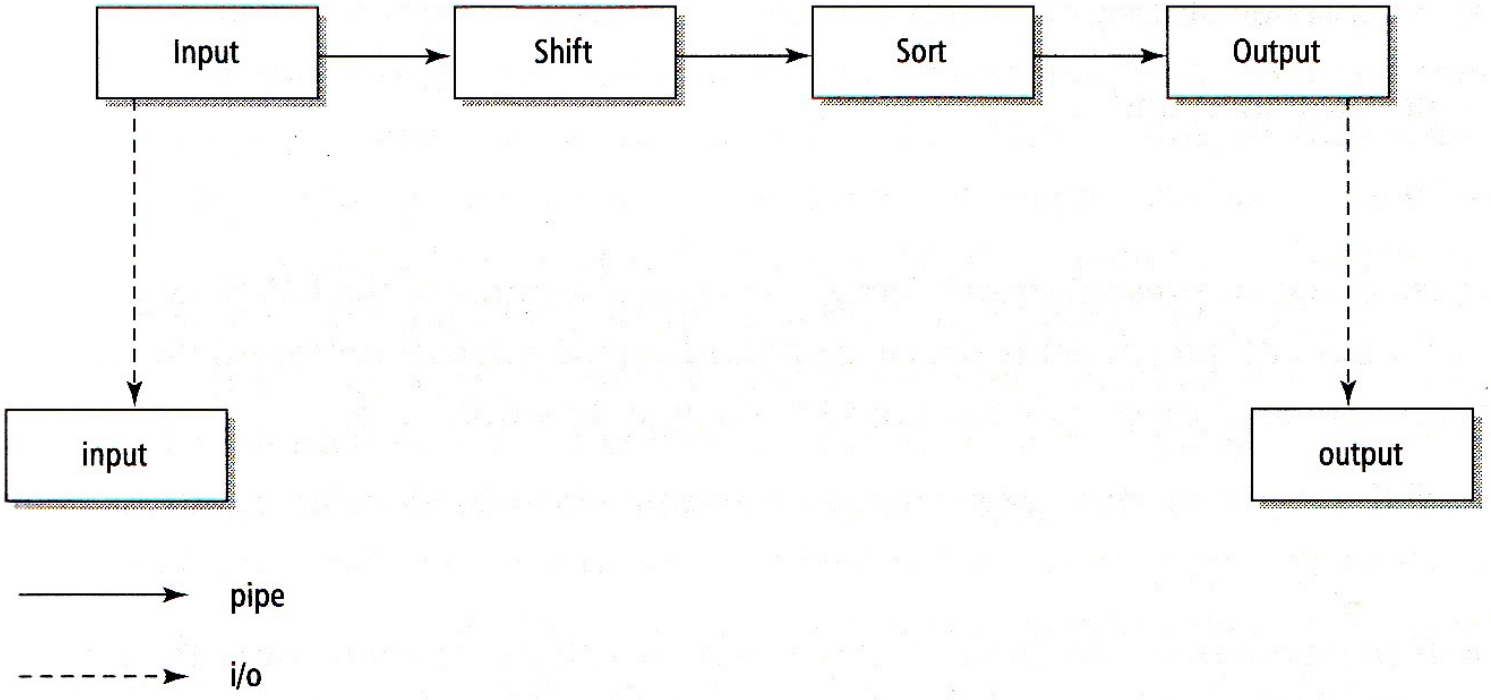


Figure 10.4 Pipes-and-filters solution of the KWIC-index program

Easy to plug in another filter. Can't use data from any module but the previous. Does not handle errors well. Errors must be passed through successive filters.

These designs all have strengths and weaknesses, and software qualities of the ultimate system start to appear, *at the architectural level.*

How good is each architecture for...	**#1. Main program and subroutines with shared data**	**#2. Abstract data types**	**#3. Implicit invocation**	**#4. Pipes and filters**
Changes in functionality, such as skipping lines starting with "the"	Neutral, though might require excessive tinkering with existing code.	Hard because the processing algorithm tends to be spread across components.	Particularly good. Functional changes can generally just be added on to the existing chain of modules.	
Decomposibility for independent development	Hard - all developers need to know all data structures	Good. Just need to agree on the way functions are called.		Good. Just need to communicate with one upstream and one downstream component.
Performance	Good. There is very little redundant or extraneous processing. Modules quickly and directly manipulate the data.		Bad—overhead in the scheduling of events.	Bad—requires parsing and unparsing at every stage.

Chapter 12 - Software Design

You consult a map before starting a trip. It outweighs the misery of time lost by going down the wrong road. (This is a pre-GPS statement.)

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, the sorting module sorts. You don't really 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 hidden 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 a little like the hints or mnemonics you use to remember someone's name. Don't tell them!)

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.

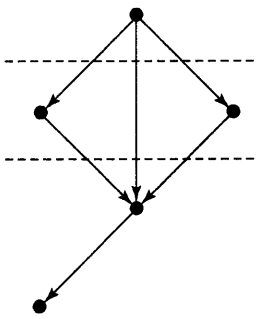
<u>Relations can include:</u>

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.

Good in-class exercises:

(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.

Chapter 13 - Software Testing

(Some of the ideas in the lecture come from Greg Foltz, a software tester from Microsoft who guest lectured in this class on 11-7-04.)

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

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.)

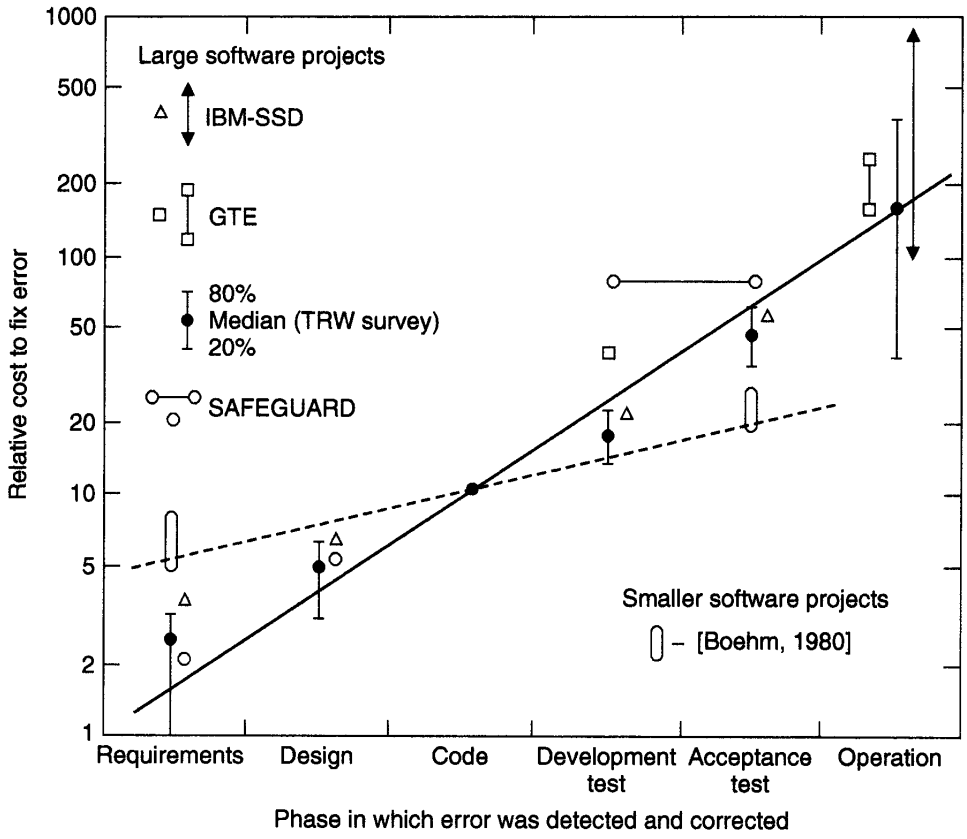


Figure 13.1 Relative cost of error correction (*Source*: Barry B. Boehm, *Software Engineering Economics*, Figure 4.2, p. 40, ©1981, Reprinted by permission of Prentice Hall, Inc. Englewood Cliffs, NJ.)

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.
MS 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 - Mostly Validation

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

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 needed to know what to do.
Note how the SRS for VizFix is less precise, and closer to what you have been producing. I thought through the problem after developing one similar system, and by myself thought through a better system, and just wrote down

my ideas. But they are less feasible, testable, consistent, and complete. Use the Multimodal Experiment software as an example, not the VizFix.

Verification: Testing in the requirements phase also includes verification, but 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 - Validation *and* Verification:

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 the difficulty in both recording and listening to Skype audio, or whatever that was) 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 of individual components, done in conjunction with coding. Usually individually. Done during implementation.
Integration testing of two or more modules when connected. Involves multiple team members. Done during the "testing" phase.

(Van Vliet organizes around) three approaches to testing:

- **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.
- **Fault-based:** Generate a large number of mutated (errored) variants of a program, and see if your testing process finds all of the errors.
- **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:

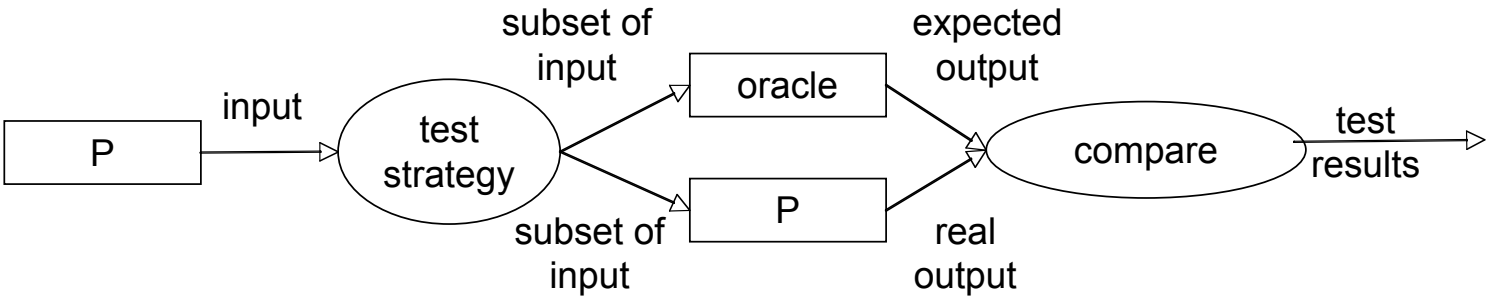


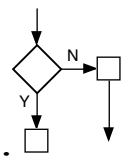
Figure 13.2 Global view of the test process.

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

Path-testing or control-flow coverage. 

Branch coverage.

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

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: 

Fault-Based Techniques

Fault-based techniques do not directly test the code but instead test your testing procedure. The idea is, if your testing procedure is thorough and adequate, it should catch all possible errors in the code.

Mutation testing is most common. Automatically mutate the code by

- replacing one constant or variable with another
- change "`if n<0`" to "`if n<1`"

(See Table 13.5 for more.)

See if your testing procedure finds all the errors.

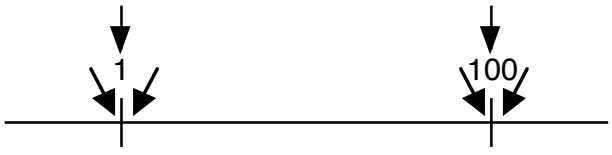
(A problem: Some planted bugs do not change the nature of the program.)

The nerdy jargon is "What percent of the mutants did you kill?"

This is a real thing. (Prof. Young, 2019)

Error-Based Techniques

Complementary to coverage-based.

Identify where errors are likely to occur. Such as on the boundaries, "fencepost errors" and other "off by one" errors. Test right on, and around each boundary: 

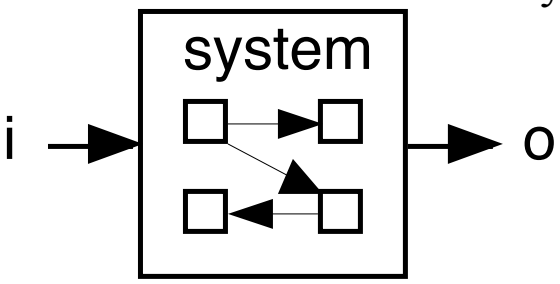
Faults are likely to occur when two modules developed by different teams interact, so focus testing on the interaction between the these modules.

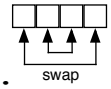
Another way to organize testing approaches:

- **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 MS, the developers or testers would try to do a real project with the system.

Regression testing: 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 MS 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 5 (R&C 2002) - Interaction Design

Notes from Rosson & Carroll (2002) by A. Hornof in 2012, 2015.

Information design focused on figuring out what task objects and actions to show, and how to represent them. The goal of *interaction* design is to specify the mechanisms for accessing and manipulating task information.

(Don Norman's example of a wall of doors with identical handles.)

Interaction design tries to make sure that people can do the right things at the right time.

The interaction design that you build into a system will determine the activities that your users can engage in.

The human-computer interaction cycle: Establish a human goal, translate it into a system goal, develop an action plan, execute the plan, perceive the results of the execution, interpret the results, and decide whether the goal has been accomplished.

The plan-execute-perceive cycle of human-computer interaction

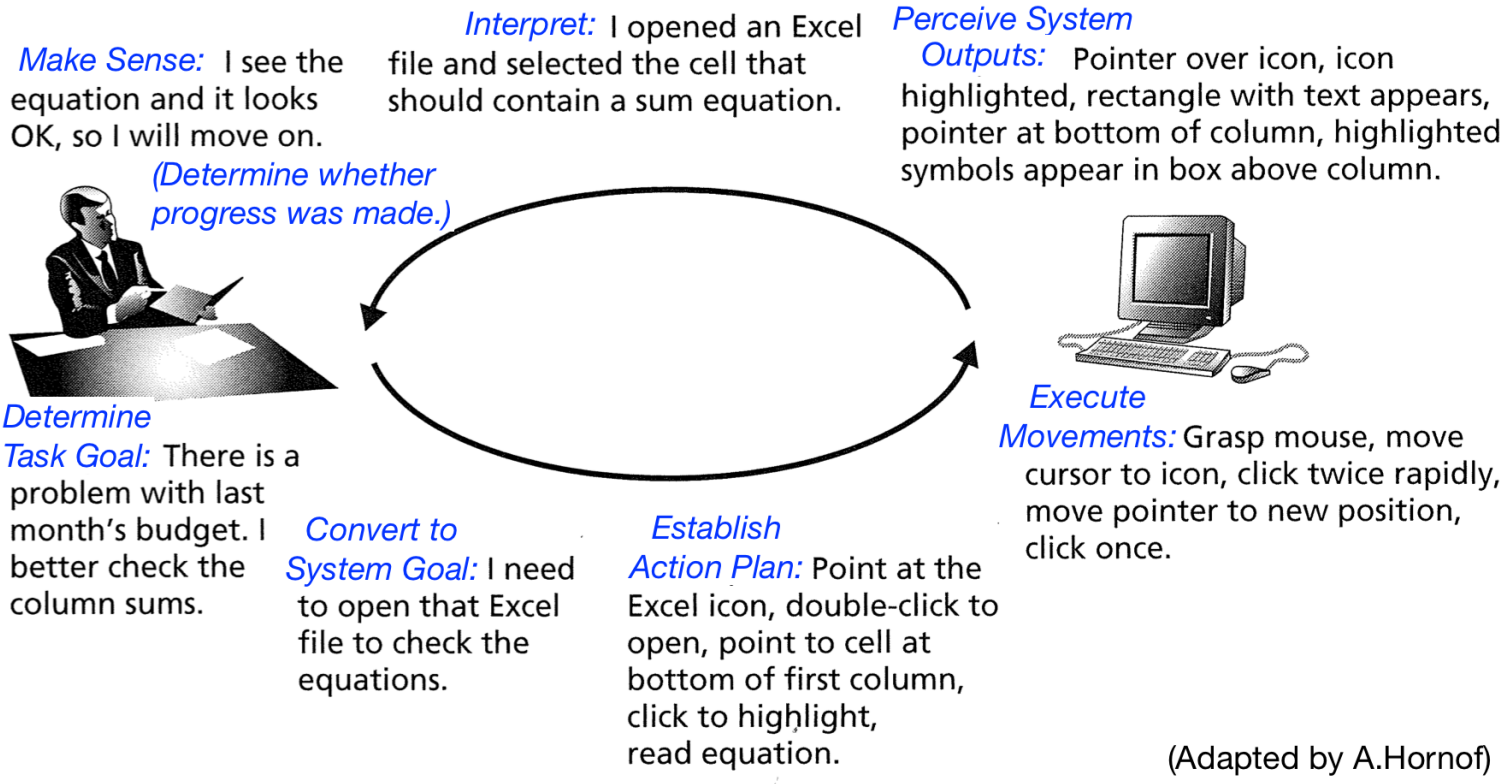


Figure 5.1 Stages of action in a budget problem: choosing, planning, and executing an action, and then perceiving, interpreting, and making sense of the computer's response.

Interaction design relates to how easy it is for a user to (a) translate his or her goals into the procedures for using a system to accomplish those goals, (b) carry out those procedures, and (c) determine that he or she is making progress towards his or her goals.
(Example: Installing Keynote on iPad.)

5.1 Selecting a System Goal
People approach a computer with a human goal. They translate it into a **system goal** and determine and execute an appropriate **task strategy** to accomplish the human goal using the system.

An **affordance** refers to perceivable characteristics of an object that helps a person to know (not "that makes it obvious" as the book defines) what the object can do, and how it can be manipulated. It relates to "stimulus-response compatibility", which is a measure of the speed and accuracy with which a person can learn, execute, and retain knowledge of the mappings between stimuli and responses. Such as, the mappings between four lights and four buttons.

When one of these lights turn on ⟶ ① ② ③ ④

Press the button it is mapped to ⟶ Ⓙ Ⓚ Ⓛ ;

Stimulus-response compatible mappings: 1J 2K 3L 4;
Stimulus-response *incompatible* mappings: 1L 2J 3; 4K

The **gulf of execution** refers to the difficulty that people have in determining the physical actions needed to accomplish a task with an interface. The **gulf of evaluation** refers to the difficulty that people have in determining whether they are making progress towards those goals after executing an action. (Neither is the "psychological distance" of anything as the book states because there is no such measure.) These two "gulf of" terms are not actually used very often, but they are terms from an 1980s book popularizing human factors (Norman's *POET* book). And the fundamental concepts are *very* important in UI design and analysis (but I prefer plain words).

Direct manipulation is thought to make computers easy to use by introducing graphical user interfaces (GUIs) rather than command-line interfaces because GUIs perhaps reduce the gulf of execution by making screen objects look and sort-of behave like things in the world. And because it makes it difficult for programmers to get away with assigning radically different functions to the same actions. Though they sometimes do, such as how dragging a file or a folder to the trash deletes it, but dragging a floppy disk to the trash ejects it.

Direct manipulation started with WIMP interfaces: Windows, Icons, Menus, Pointers. Touchscreen displays, such as with tablets and smartphones, take direct manipulation to a greater extreme. But all kinds of inconsistencies are introduced. For example, what is "clickable" still needs to be made very clear, and often is not. Direct manipulation is not a magical way to make interfaces easier to use. For example, on a touchscreen, there is no "right click" to see a number of potential commands for an object. And you cannot rest your finger on a button while deciding whether to press it, or touch type. And it introduces many, many modes.

5.2 Planning an action sequence: People develop and execute task strategies. When interacting with computers, these typically include perceptual and motor. They can also be purely cognitive. They can be planned ahead, prepared. So consistency matters a lot, because they permit a user to plan a few steps in advance based on how they expect the functionality to be accessed, and how the computer will behave. (Such as, when you encounter a couple fields that say "username" and "password," to be able to type your username, tab, your password, and enter. This was not the case on DuckWeb a few years ago.)

Action sequences, or cognitive strategies, are planned and executed on the micro level (tasks that last a few seconds, such as above) as well as the macro level (tasks that last minutes, such as connecting to a network and sending a print job to a printer).

The UI designer's challenge is to support the user at every step in their action plan, and to make it clear to them what functionality is available so that the users can map that functionality to their tasks and goals. Such as, if a user wants to print double-sided, make it clear whether that functionality is available, and if it is how to access it.

Consistency is important. People can **chunk** interaction sequences such as typing in a username and password, copying and pasting, opening applications. To "chunk" is to join several interrelated pieces of data into a single piece of data. Such as how encoding LBT WCP ULO may require more than just three chunks, but other arrangements should take just three chunks. You can also chunk procedural knowledge, such as how scrolling down in a document should be consistent across all applications, and the same actions should always accomplish it (whether it be two fingers up—or down—on a trackpad, moving your hand to and rolling the scroll wheel on the mouse in a manner that can be prepared before your hand arrives.

A expert-user command sequence for....

Opening a program: Command-Space and the first few chars of the application name.

Googling something: Command-Space, "Fire", Enter, Command-L Tab.

Turning off unwanted "help" in PyCharm: See "+Notes on Using PyCharm IDE.pages"

Action sequences—or task strategies—should be consistent across applications, and should not conflict. This permits the user to plan and prepare the execution of the strategy before initiating the task. When the system fails to support the prepared and executed action sequence, not only does the user have to diagnose, troubleshoot, and experiment to figure out how to do it; but all of the preparation for the initial execution is also wasted. And the interaction with the device becomes the primary task, not the human-centered goal that initiated the interaction. For example: You go to print or scan a document, and it doesn't work.

Mistakes: An inappropriate intention is established and pursued. More common among novice users. Buying a copy of "Garage Band" because you want to start a band in your garage.
Slip: The correct goal is attempted, but a problem arises along the way. More common among experts. Example: The goal is to get cash from an ATM; you do it but you leave your ATM card. Can often be avoided by improving the interaction design, such as by giving back the card before the cash.
More examples on page 169, with design approaches to avoid the problems.

Modes should, in general, be avoided in UI design. Modes are restricted interaction states in which only certain actions are possible. Such as a "modal" dialog box that requires a response before you can do anything else with your computer; some reminders software work this way, such as to alert you of a scheduled event. A pop-up window on a web page asking you to take a survey is a modal dialog box within the context of that web page. Smartphones use modes extensively; it contribute to their reconfigurable flexibility, but it also requires lots and lots of extra button presses and swipes to switch from one mode to another.

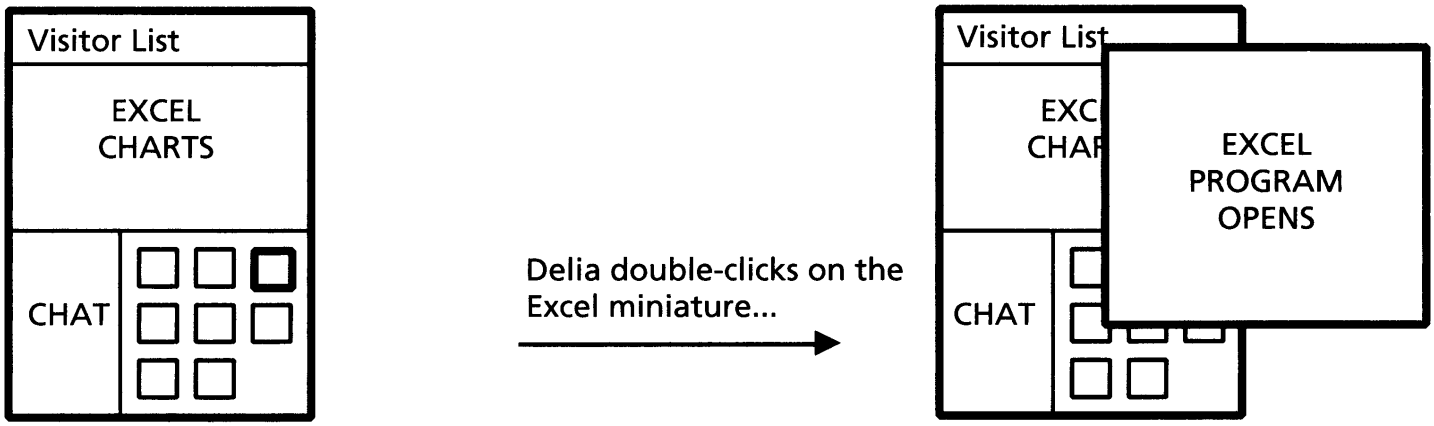
Articulatory directness—how directly a device maps to its input requirements—is interesting to think about in terms of touch-displays. Spreading two fingers is surely like stretching something, to zoom, but a four-finger versus a three-finger swipe does not seem to have articulatory directness with anything in particular.

Interpreting System Feedback
Give the user feedback with regards to how they are progressing towards their goals, at multiple time scales, including responding to any input within 100 ms, just to show that the system received your command, but also on the time scale of seconds, showing progress towards the goal. (Unix does not give great feedback. Many direct manipulation interfaces do.)

Storyboards
A **storyboard** is an event-by-event description of a sequence of interactions between a user and a device. They are named after the comic-book-like sequences that are used to plan movie shots.

A storyboard of the start of the bank robbery in *Batman* - *The Dark Knight* (2008)

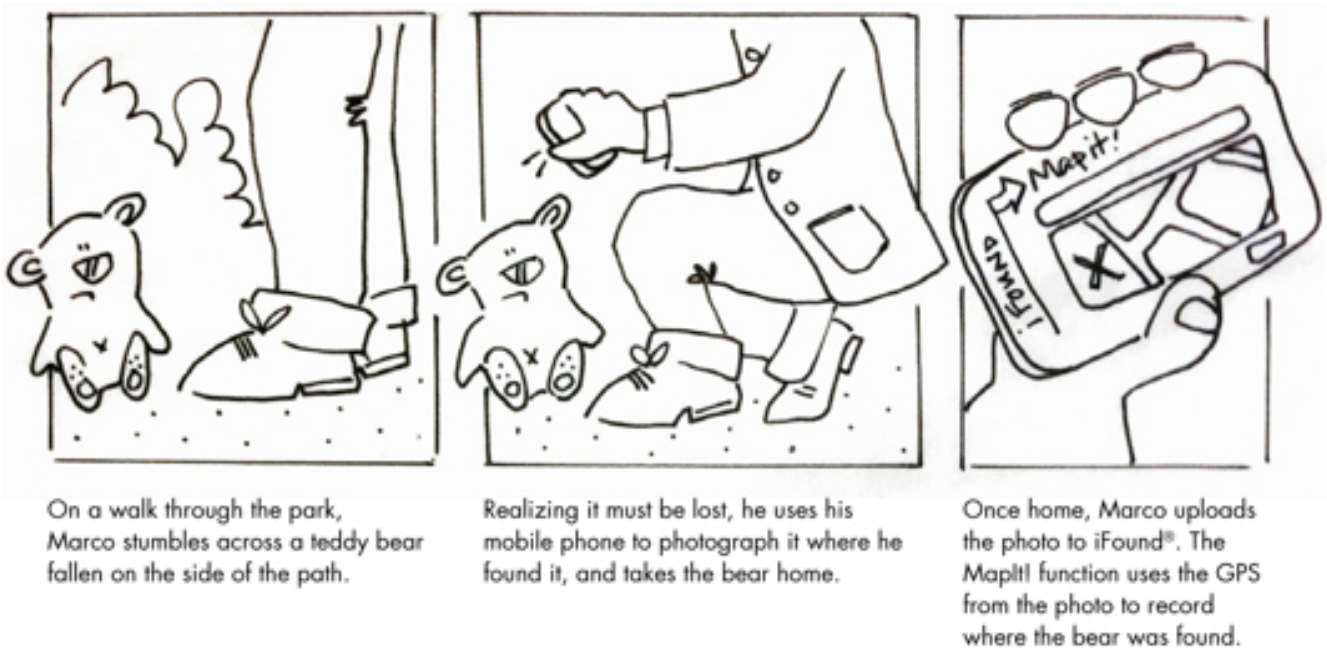
http://s3images.coroflot.com/user_files/individual_files/152129_WOEkopMM6ezXqt52G9vrtE758.jpg



1. Alicia and Delia look at the Excel charts Sally has prepared.

2. The Excel application is launched on the data files Sally has provided; Delia works with Excel independently of the exhibit.

From Figure 5.7 in Rosson & Carroll



On a walk through the park, Marco stumbles across a teddy bear fallen on the side of the path.

Realizing it must be lost, he uses his mobile phone to photograph it where he found it, and takes the bear home.

Once home, Marco uploads the photo to iFound®. The MapIt! function uses the GPS from the photo to record where the bear was found.

Part of a storyboard for an "iFound" app, which would interact with "iLost".

http://web.mit.edu/2.744/www/Project/Assignments/userExperienceDesign/ifound.jpg

Storyboards are not interfaces but they capture, in a static representation, the time-based element of the interface, which makes it easier to consider alternative designs side-by-side.
(Perhaps show the EyeMusic storyboards, annotating sound file, and the NIME promo video.)

How can you represent interaction sequences? Remember, a screenshot is not an interface. You must show how an interface evolves over time, such as with a storyboard. “Here is what the user sees. If they click here, then they see this....” The challenge is to represent a dynamic artifact.

Action sequences can be studied, and improved, at different time scales, including the fractions of a second needed to move the mouse to click on a target, or press keystrokes.

Fitts’ law predicts pointing time as a function of distance (d) and width (w). There is a logarithmic relationship between d/w and pointing time. MT = a + b log (d/w) + 1. The main point is that tiny targets are very slow and difficult to click on, and the edges of the screen have certain advantages. But overall pointing-and-clicking is quite slow for time-pressured practiced tasks. You should learn keyboard shortcuts, even for responding to dialog boxes. (It is sort of foolish not to.) A good interface design should support keyboard shortcuts. One of the big differences between software for the masses like iPhoto and software for the pros such as Lightroom is that the pro versions support *lots* of keyboard shortcuts, such as to rate a photo *and* advance to the next photo with a single keystroke. (My friend Mark in NYC took my advice.)

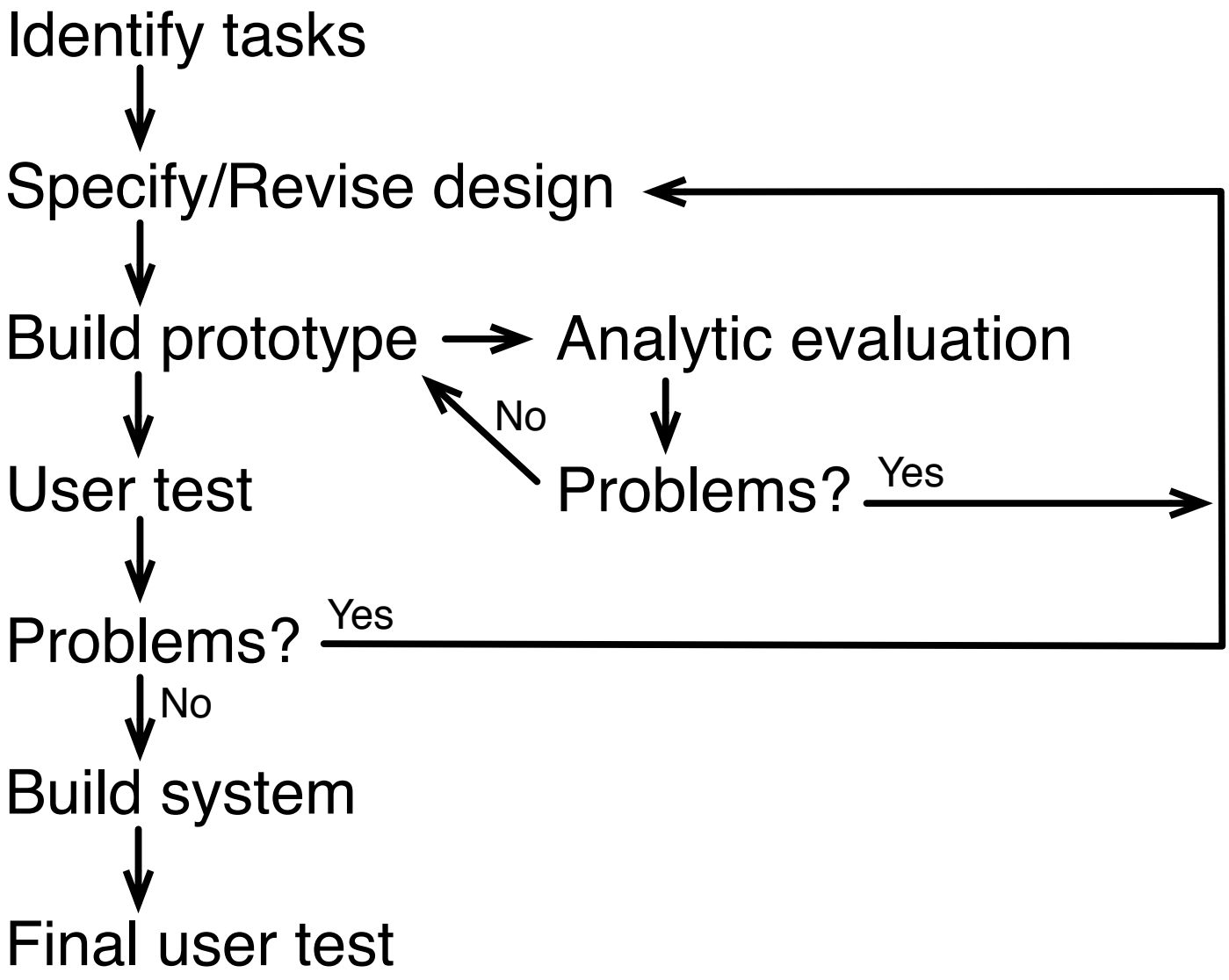
Chapter 7 (R&C 2002) - Usability Evaluation

7.1 - Usability Specification for Evaluation

A **usability evaluation** is a study to determine the ease of use and ease of learning of a system. **Ease of use** is a measure of how well a system supports users accomplishing tasks.

Formative Evaluation	vs.	**Summative Evaluation**
takes place during the design process—how are we doing?		takes place after the design process—how did we do?

How usability evaluation fits into a software development process model:



7.3 - Analytical Methods

Analytic Evaluation	vs.	**Empirical Evaluation**
Studying or modeling the interface without users.		Observe real users doing real tasks.
Cheaper, faster, sometimes can help to show *what* is wrong.		Slow, expensive, does not always reveal *why* better or worse.
		Follows the pattern of a psychological experiment.

Two examples of analytic evaluation techniques:

The keystroke level model (KLM) is an analytic usability evaluation technique in which you basically:

1. Count the number of keystrokes and mouse moves and clicks (or touchscreen presses and swipes) necessary to do a task with a particular UI or UI design.
2. Assign appropriate timings to each keystroke (0.28s) and mouse move and click (1.3s).
3. Add up the time required to do all of the actions.
4. Use that time as a basis for comparison to benchmarks, or comparison to alternative designs.

(See Card, Moran, and Newell, 1983, for more information on KLM.)

Heuristic Evaluation (Nielsen, from 1994 *Usability Inspection Methods*) is an analytic usability evaluation technique in which you make passes through the interface, inspecting for problems based on these heuristics, or guidelines:

- *Visibility of system status*: The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.
- *Match between system and the real world*: The system should speak the users' language, with words, phrases, and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.
- *User control and freedom*: Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.
- *Consistency and standards*: Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.
- *Error prevention*: Even better than good error messages is a careful design which prevents a problem from occurring in the first place.
- *Recognition rather than recall*: Make objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.
- *Flexibility and efficiency of use*: Accelerators—unseen by the novice user—may often speed up the interaction for the expert user to such an extent that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.
- *Aesthetic and minimalist design*: Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.
- *Help users recognize, diagnose, and recover from errors*: Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.
- *Help and documentation*: Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

7.3 - Empirical Methods

This section in the textbook provides a very accurate and relevant discussion of how to conduct a usability study. This is perhaps the most important section in the textbook for you to learn. All of the terms that are in bold in this section are very important terms.

(The Appendix on "Inferential Statistics" is also very good, on p.363.)

"The gold standard for usability evaluation is empirical data."

Empirical: Based on observation (not theory or conjecture).

You are looking to establish a cause-and-effect relationship between characteristics of the system and ease of use. You want to claim that your interface causes a task to be easy to perform for a population.

Validity

You want your experiment to have good "validity".

Validity refers to the best available approximation to the truth of propositions.

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

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.

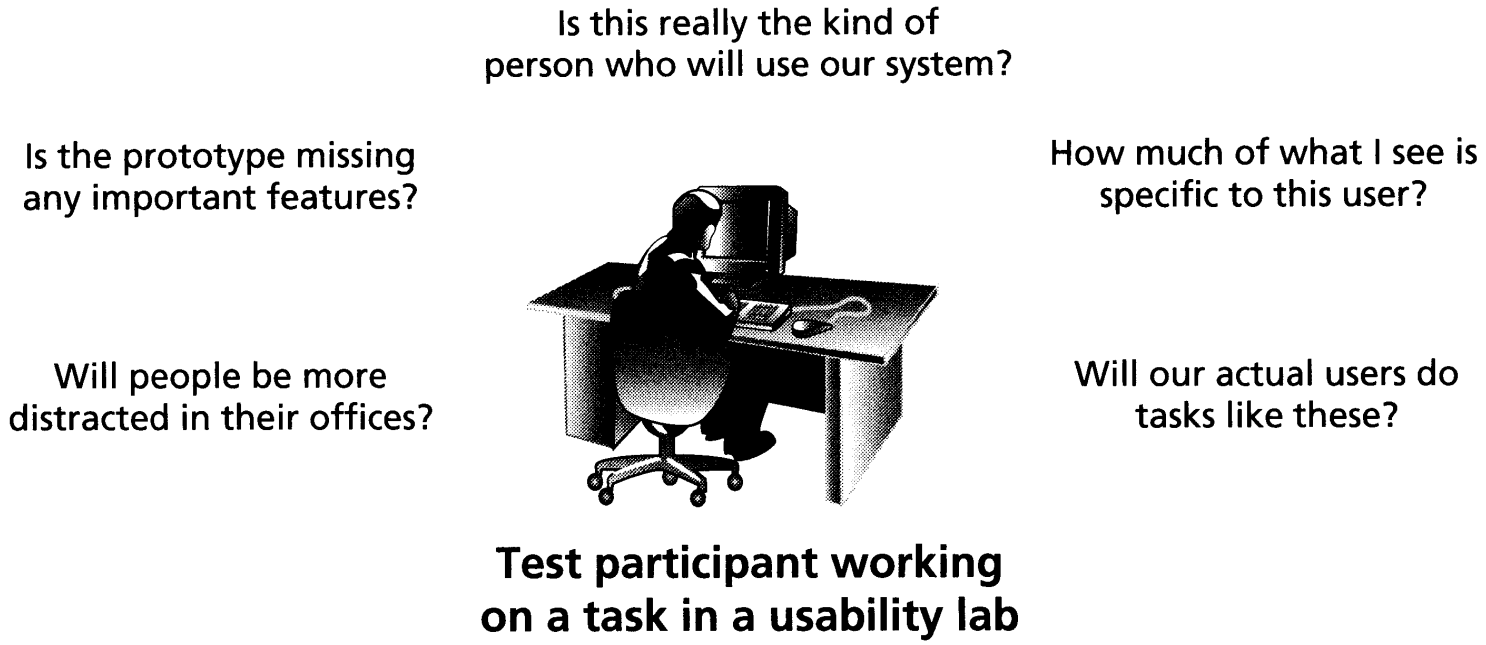


Figure 7.3 Validity concerns that arise in usability testing done in a laboratory.

Example script for a usability study

Roles: Test monitor, technicians, users or "participants".
Recruitment criteria (for this example):
1. Users who have never used an iPad, iPhone, or iPod touch.
2. Users who have used the iPhone (or iPod touch) calendar (at least once a day for at least a year? month? And who find it relatively easy to use). And who love their iPhone?

Purpose of observation: I am trying to learn how people might use the iPod Touch (or iPhone) to enter an appointment in a calendar.
This study should take about five to ten minutes. Feel free to quit any time.
I would like to ask you to think-aloud while you do the task. By this I mean to say what comes to your mind as you are working. To help you do this, I am going to ask the two of you to work together and to agree on every action that you take, and to make sure that both of you understand what is happening all the time. (The think-aloud protocol can be facilitated by two users doing "co-discovery.")
Your first task is to create an appointment this Saturday from noon to 4PM to grade papers.
Your second task is create an appointment on June 13 to attend commencement.

<u>Debriefing questions:</u>
1. What did you think?
2. How did you do the tasks?
3. How did you figure out how the calendar worked?
4. Did the system respond as you expected? Always?

5. Was there anything about the task that seemed particularly easy or difficult?
6. What were some of the feelings that you had as you did the task?
7. Do you think the calendar is easy or difficult to use?
8. Is there anything else that you would like to share about this?
9. Those are all of my questions. The study was designed for the hypothesis below. Do you have any questions for me?

My hypothesis is that the iPhone calendar interface causes difficulty in recording appointments. (I have told you before that this is an unnecessarily difficult task.) I will operational my hypotheses to be:

H1: In order to enter an appointment, a novice user will require at least twenty screen touches (in which a swipe will count as two screen touches) in addition to the appointment's text string.

H2: In order to enter an appointment, even an expert user will require at least twenty screen touches (in which a swipe will count as two screen touches) in addition to the appointment's text string. And at least one error will occur for every appointment entered, in which an "error" is any undesired system response that requires the user to make an extra movement.

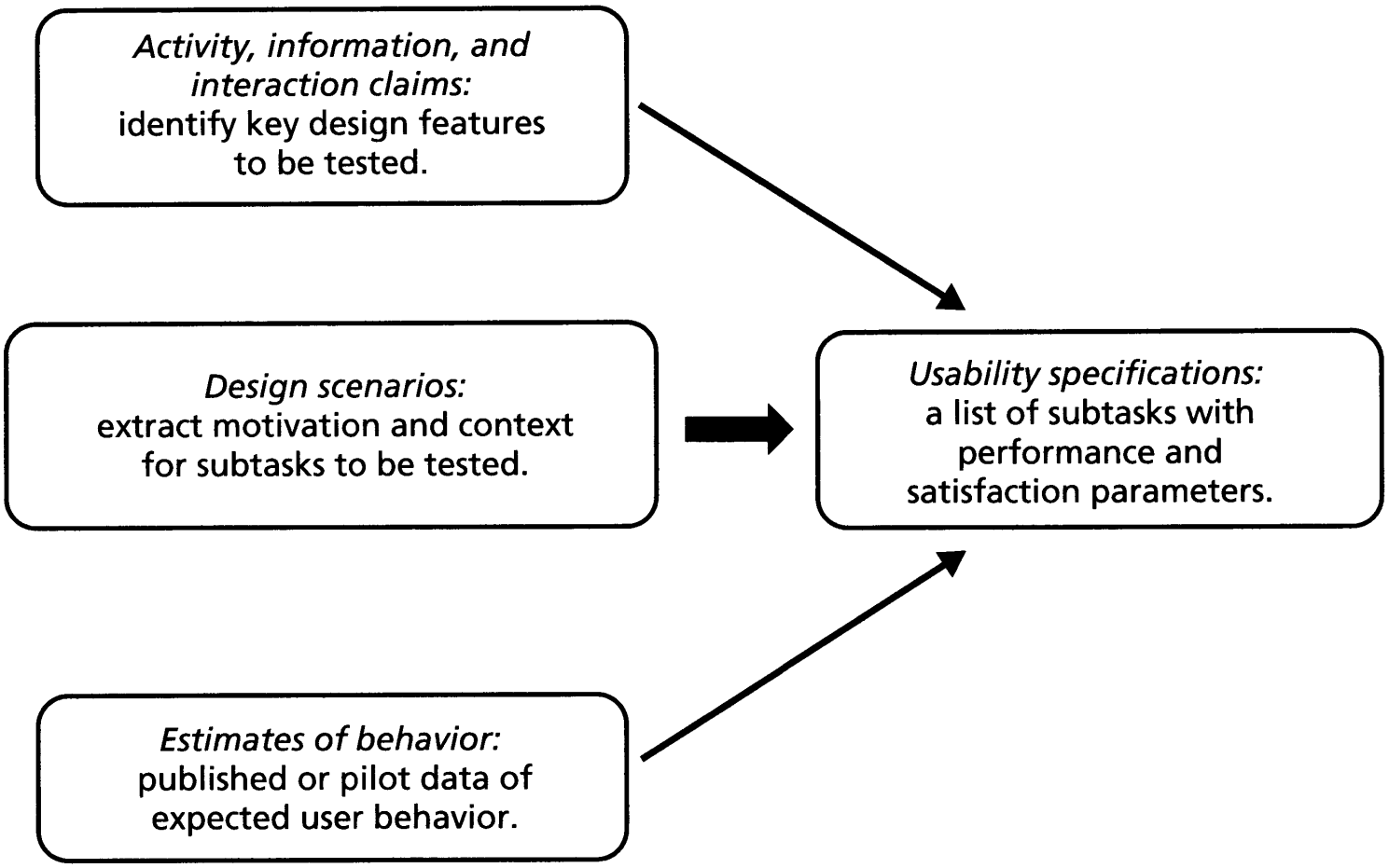


Figure 7.4 Developing usability specifications for formative evaluation.

Important Topics in Empirical Methods

Think-aloud protocol - prompting a user to verbalize what they are doing as they proceed.
Co-discovery - having two users work together and agree on each step aloud.

Controlled experiments versus field studies.

Independent variable - characteristic that is manipulated to create different experimental conditions.
Dependent variable - an experimental outcome.

Hypotheses - predictions of causal relationships between dependent and independent variables.
Experimental design - the details of how a cause-and-effect relationship is explored between independent and dependent variables.
Within-subject - all participants see all conditions.
Between-subject - different groups see different conditions.
Random assignment to remove order effects.
A major goal in experimental design is remove alternative explanations as to why the dependent variables changed when you changed the independent variables.
Informed consent - confidentiality, can quit any time without penalty. This is to protect participants.

The VSF examples are very good. The assistance policy, for example.