Notes on Reading Van Vliet (2008)

By Anthony Hornof
Last updated April 3, 2018

These are Anthony Hornof's notes from reading Hans van Vliet (2008) *Software Engineering: Principles and Practice*, 3rd edition, John Wiley & Sons. Some of the notes might still use the page and figure numbers from the 2000 version (2nd edition) of the textbook, but I've tried to fix all of those.

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.

Preface

A software project can be like building or modifying a house.
Learning software engineering is like learning swimming.

Chapter 1 - Introduction

(Read 1/10/08)

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. There is a nice chart on p.3 that shows this.
Dramatic software failures can cause all sorts of calamities. A few specific examples are offered.
An 1982 book by Baber is cited in the context of some fictitious land of Ret Up Moc (Computer spelled backwards). I'm not familiar with this book.

"Quality and productivity are the two central themes in the field of software engineering."

<u>1.1 What is software engineering?</u>
The methodological process of building reliable, robust, efficient, accurate, useful computer programs.

Characteristics of the field:
- Concerned with "large" programs.
- Trying to master complex problems: people, processes, programs. Must be broken up and managed.
- Software evolves: Y2K, Euro, internet, new CPUs, etc.
- Building and maintaining s/w is very time-consuming. The last 10%...
- S/W development is a people problem.
- It is a UI problem. Must study people at work, understand context, provide documentation, training.

- Developers are not domain experts. They generally lack factual and cultural knowledge of the target domain.

Tacoma Narrows Bridge failure of 1940 was an example of designs and engineers extrapolating beyond the models and expertise.

Software does not wear out the same way as physical products.
"90% complete" syndrome - software "almost finished" for endless amount of time.

1.2 Phases in the Development of Software

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

But rarely a linear process.

Phases of S/W Development

Requirements engineering: Includes a feasibility study. Produces a requirements spec.
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.
Implementation: Start with a module's design spec. The first goal should be a well-documented program, not an efficient one.
Testing: Not just a phase that follows implementation.
Maintenance: Keep the system operational after delivery.
Project management: Deliver on time and within budget.

System Documentation: Project plan, quality plan, requirements spec., architecture description, design documentation, test plan.
Start 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.
Maintenance or evolution: Corrective, adaptive, perfective, and preventive.
Software life "cycle" because it is cyclic.

1.4 From the Trenches

Henri Petroski has a book on engineering successes and failures.

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.

1.5 Software Engineering Ethics

1.6 Quo Vadis - "Where are you going?"
Not yet a fully mature discipline.
Frederick Brooks "No Silver Bullet" article in 1987. Problems still persist.

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.

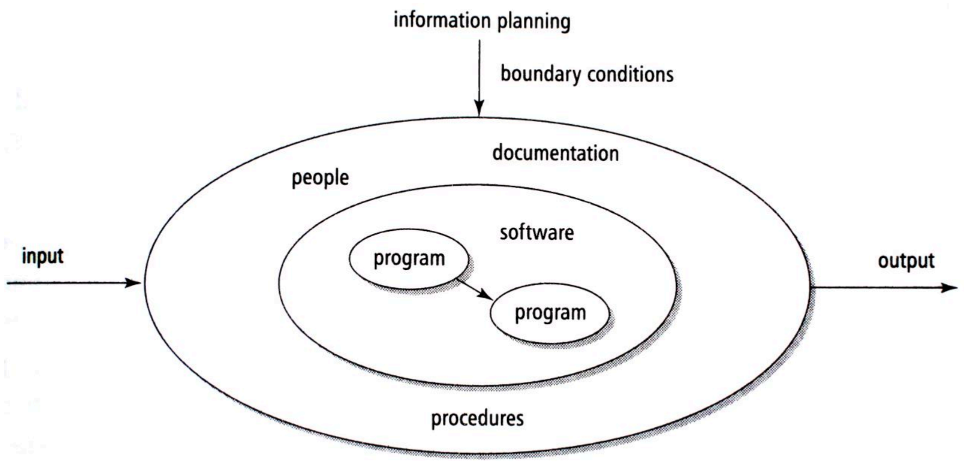


Figure 2.1 The systems view of a software development project

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

<u>2.2 Controlling a SWD project</u>
Control must be exerted along the following dimensions: time, info, organization, quality, money

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 accommodate maintenance, or going back to previous phases, very well.
Can have excessive maintenance costs. (World Tax Planner.)

In 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 (such as how Stuart Faulk suggests).

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, but with testing in every phase.
Verification & Validation in every phase to compare outcome to what is required.
    Verification: Building the system correctly.
    Validation: Building the right system.

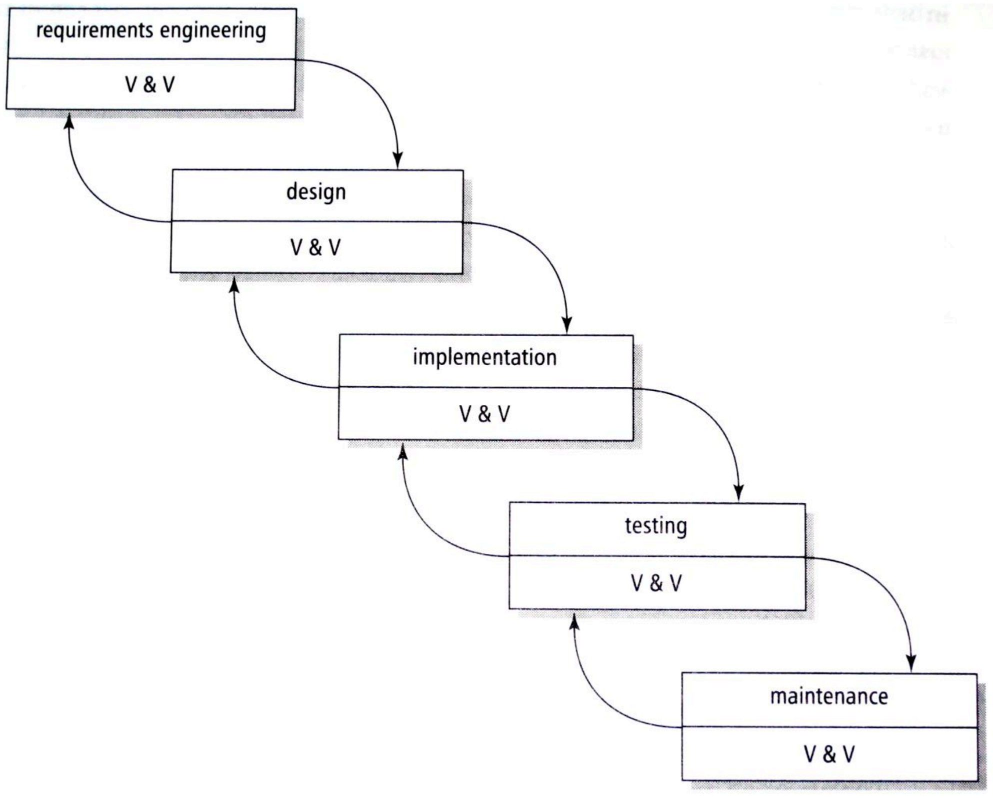


Figure 3.1 The waterfall model

Emphasis on getting the client to "sign off" on documents for 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 for the project.

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 half of the design in the "design" phase, a third 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 should be followed.)

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.

Very similar to (the former trendy approach of) RAD (rapid application 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

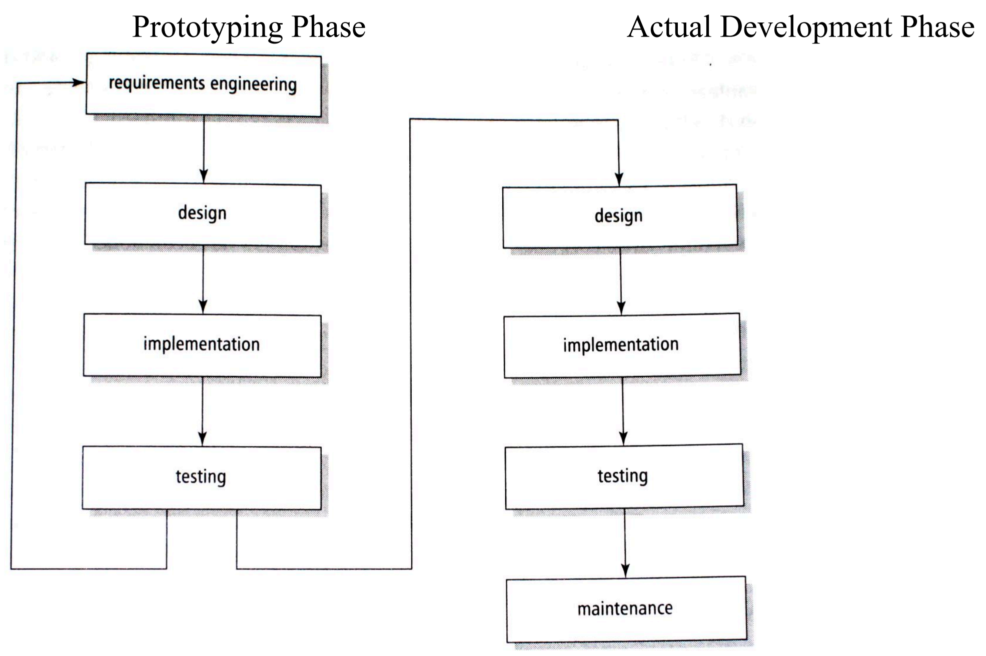


Figure 3.3 Prototyping as a tool for requirements engineering

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

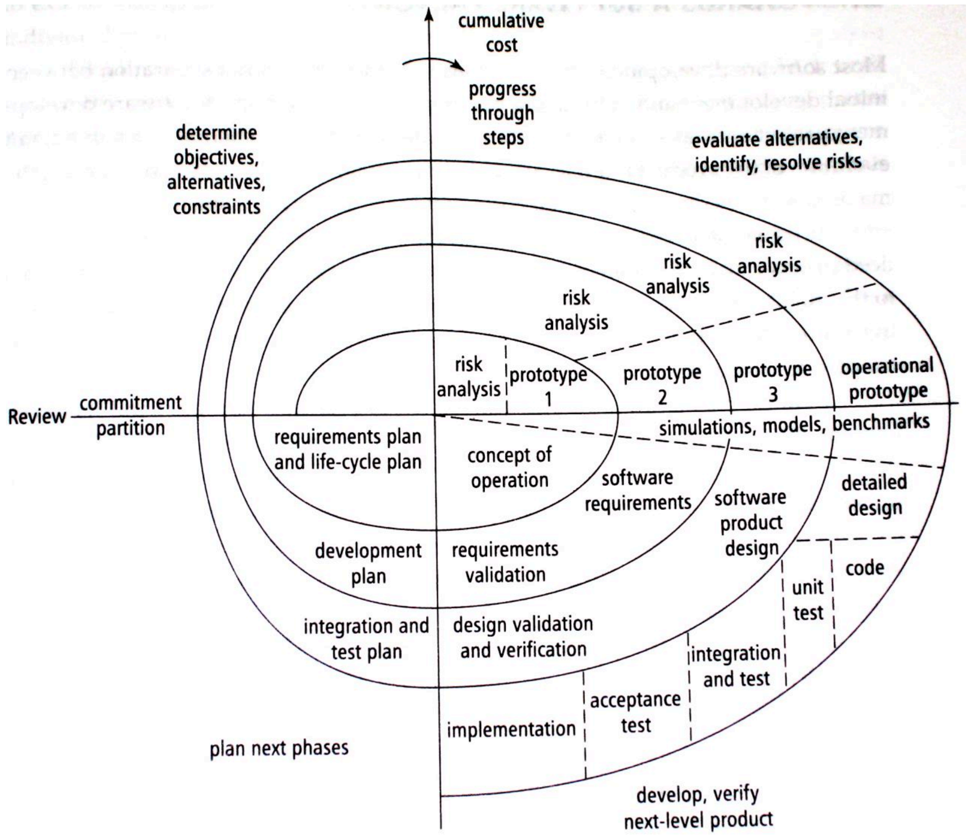
only grow at a steady, linear rate, because after the spurt you need to go back and "clean up the code" and update the documentation, etc. (Figure 3.8 on p.74)
4. ... invariant work rate: Adding more staff does not increase the speed of development. Large systems proceed at a saturated rate. (Windows 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)

"Windows Is So Slow, but Why.pdf" - S/W engineering in the news!!!

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.

Sections 3.7 and 3.8 - Software Factory and Process Modeling
These are specialized topics outside the scope of an introductory course.

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 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 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)
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...
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 on overheads.

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?

<u>Market-Driven versus Customer-Driven</u>

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

This relates to the problem we came up against the other day in thinking about how to develop an open source carpool software 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?

<u>Specification</u>

You need to organize the document. Pages 226-229 offer 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.)

<u>Requirements document should be</u>

* 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.

Section 9.4 - A Modeling Framework - Less important than other content in the chapter.

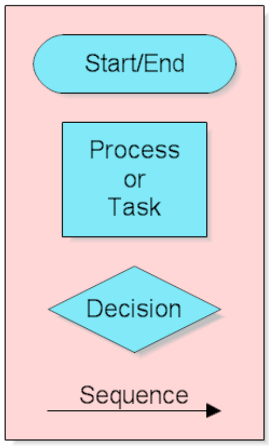
Chapter 10 - Modeling (3rd Edition)

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.
Find overheads of UML slides from Sommerville?

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 of the system, or aspects of the system.
Most have boxes and lines. It is important to be clear what each box and line represents, to build a shared understanding that is solid, clear, and will persist over time.

Diagrams (models) are generally static or dynamic.

Flowcharts show basic thread of control through an algorithm, dynamic models.
Models flow of control.



<http://www.rff.com/fcs_key.png>

The Unified Modeling Language
Diagramming techniques used in OOA and OOD (analysis and design).
Integrates and unifies the notations and methods of the three amigos: Booch, Jacobson, and Rumbaugh (object modeling technique, OMT). These are late 80s and early 90s.
There is also a UML process, but the language is still quite useful without the process.
(UML notes adapted from Sommerville, 2000, Software Engineering.)

But even before UML, or the three OO notations, roughly similar diagrams were used.

To Model	A Classic Approach	UML
Data Relationships (Static)	Entity Relationship Diagram/Model	Class Diagrams
System States (Modes) and Transitions (Dynamic)	Finite State Machines	State Machine Diagrams
How Info & Control Moves Through System (Dynamic)	Data Flow Diagrams	Sequence Diagrams

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

Major diagrams used in UML

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

State-transition diagrams: Describe the behavior of a system. Show all possible states that an object can get into as a result of events that reach that object.

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

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

Let's look at each model:

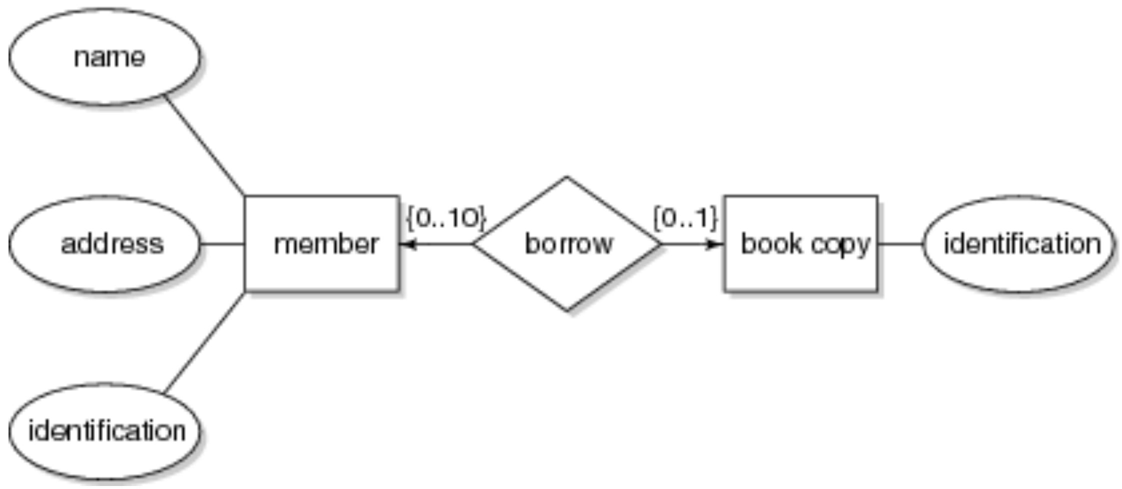
ERDs

Rectangles are entities (objects)

Ovals are attributes (entity features or characteristics)

Diamonds are associations (relationships) between entities.

Arrows connect associations to entities, and can be annotated with numbers to show cardinality (number of elements in a grouping).



Example: Figure 10.1 shows that a book can be borrowed by at most one member, and a member may borrow up to ten books.

UML Class diagrams

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

Key diagramming components: Name of class, attributes and operations, inheritance (or specialization).

Associations, similar to ERDs, such as is-a-member-of. Cardinalities.

Aggregations, in which objects can be part of more than one other object. Such as, a book can be on more than one reading list.

But *dynamic* models are critical to describe how a computer program functions because the program executes its commands over time. Similar to how a screenshot does not describe an interface. You also need to describe the dynamic aspects of the interface.

Finite State Machines (FSMs) (from Automata Theory? Discrete Math?)

Circles are states.

Arcs are transitions.

Useful for describing some key states that a system or component moves through.

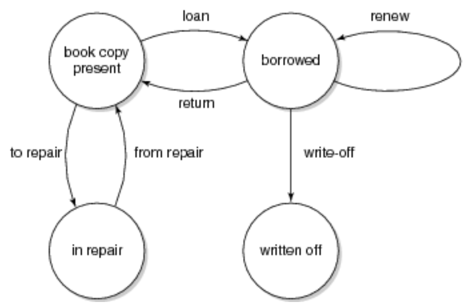
Such as to describe a UI problem.

click in text field ->

entering number with mouse entering with keyboard

<- click on a key on the screen

Try to combine these states. In general, try to avoid “modes” in UIs.



Example: Figure 10.2 shows the states that a book can be in.

UML State Diagrams

Very similar to FSMs, but add a start and end state.

Can have hierarchies of machines, introducing abstraction. Figure 10.11 shows an example.

A *dynamic* model that can illustrate an object’s lifecycle.

Stakeholders need to share an understanding of the dynamic aspects of the system.

Include two or three *dynamic* models in your final SDS. Another dynamic model....

Data Flow Diagrams (classic approach)

Rectangles show external entities.
Circles are processes.
Boxes w/o sides are data stores.
Arrows show data flow.

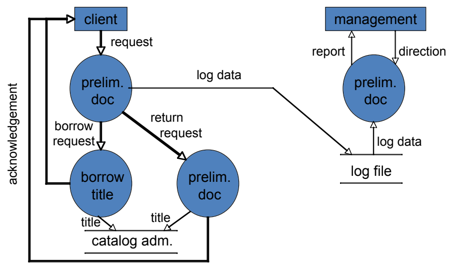


Figure 10.3 shows an example.

UML Sequence Diagrams (or Interaction Diagram, or message sequencing charts MSC)
The horizontal dimension shows objects in rectangles, with vertical dashed "lifelines".
Narrow rectangles on the lifelines show when the object is active. Time moves downward.
Arrows show messages between objects. Dashed lines indicate a "return" of some sort.
Also commonly known in the software engineering literature as MSC.

The difference between a State Diagram and an Interaction Diagram
A statechart says "All allowable sequences *must* conform to this state machine" whereas an interaction diagram says "Here is one possible sequence of actions." (MY, 11-9-10)

Conclusion: UML evolved from earlier OOA and OOD methods, which evolved from earlier non-OO diagraming and design techniques. When you think about a piece of code that you are going to write, you are already thinking about a range of 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, but also to communicate, record, and evaluate ideas with other team members and stakeholders. This is an important aspect of software engineering, the study of the full lifecycle of building things that run on computers.

Chapter 11 - Software Architecture (3rd Edition)

Architecture is typically thought of as the study and practice of constructing buildings.
A friend of mine (Lars) who is a that kind of an architect went to a computer conference and told a computer person that he is an architect, and the computer person said "hardware or software." So much of working across disciplines is learning the language.
"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.

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.
"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.

Software architecture: The large-scale structure of software systems or, more thoroughly, the 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)

<u>Software architectures serve three purposes (from van Vliet):</u>

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.* (Faulk's mentioned this in the context of a valued business entity.)

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.

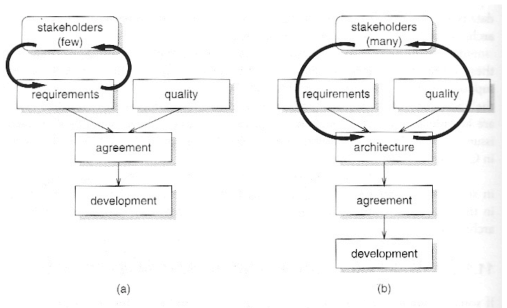


Figure 11.1 Software life cycle (a) without and (b) with explicit attention to software architecture

<u>Architectural Views</u>
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 building architecture are overplayed a bit in this edition. He picked a really stupid building design pattern as his example: "Tall buildings make people crazy." It is just irresponsible to reprint such trash, but this is just what happens when people reach across into other disciplines. A little knowledge can be a dangerous thing.

But anyway, he tries to present 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 much 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.

<u>KWIC-Index Example</u>
A classic example from Parnass (1972) though not thought of as an example of "software architecture" until 1996. (I got this 2nd detail from the footnote at the bottom of p.259)

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.

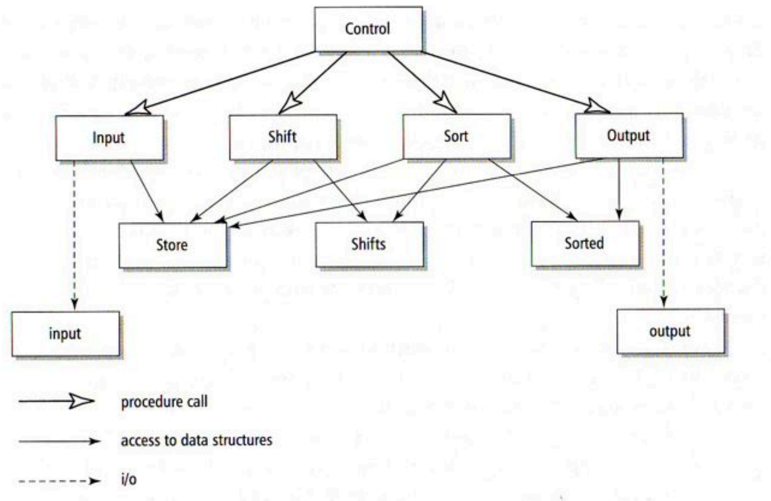


Figure 10.1 Main-program-with-subroutines solution of the KWIC-index program

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.

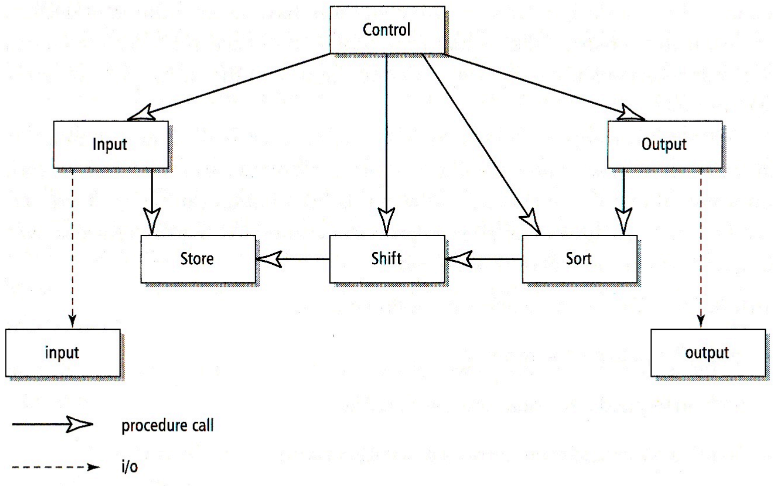


Figure 10.2 Abstract-data-type solution of the KWIC-index program

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.

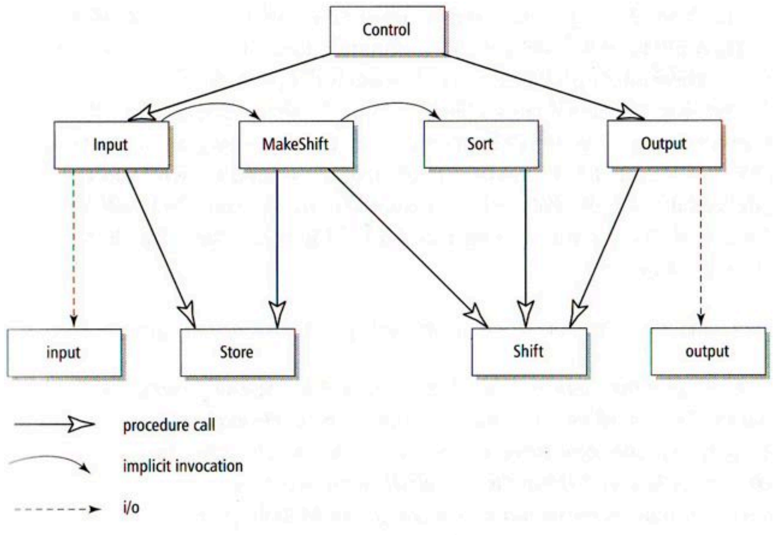


Figure 10.3 Implicit-invocation solution of the KWIC-index program

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

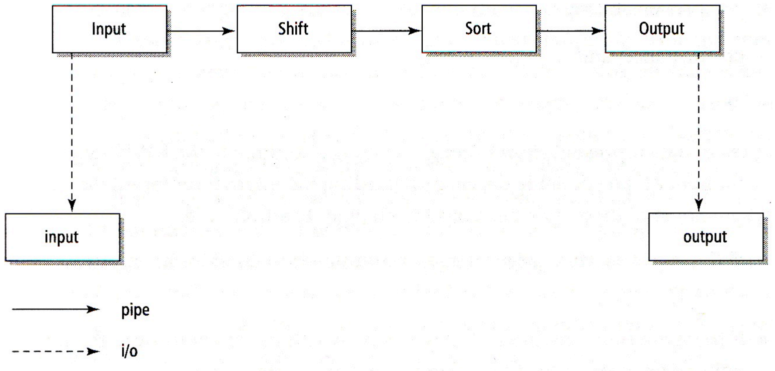


Figure 10.4 Pipes-and-filters solution of the KWIC-index program

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 11 - Software Design (2nd edition)

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. Info hiding
4. Complexity (size based, structure based)
5. System structure

Abstraction
Concentrate on the essential features and ignore—abstract from—those irrelevant to te current level. (For example, the sorting module sorts. You don’t really care how.)
Procedural abstraction - subproblems decomposed into subproblems.
Data abstraction - (OO Design)
Finds a hierarchy in the program’s data.
Primitive structures - booleans, ints chars, strings.
Provides some info hiding.

Modularity
Parnass states the benefits of modular design.
... continued in paper notes.

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:
- V&V
- 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.

Show Figure 13.1: 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.)
Remember that even the classic 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.

<u>Testing in the Requirements Phase is mostly Validation:</u>
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 Multimodal 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.

<u>Testing in the Design Phase is both Validation *and* Verification:</u>
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.

<u>Testing in the Implementation Phase</u>
This is where we typically think of the testing being done.

Unit testing of components, done in conjunction with coding. Usually individually.
Integration testing of whole system. Done when modules are put together. Usually the team.

<u>(Van Vliet organizes around) three approaches to testing:</u>
- **Coverage-based:** Focuses on making sure that enough of the system gets tested. Such as, every function call is examined with a set of test cases of legal and illegal inputs.
- **Fault-based:** Focus on finding problems. Set a goal for how many to find.
- **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:

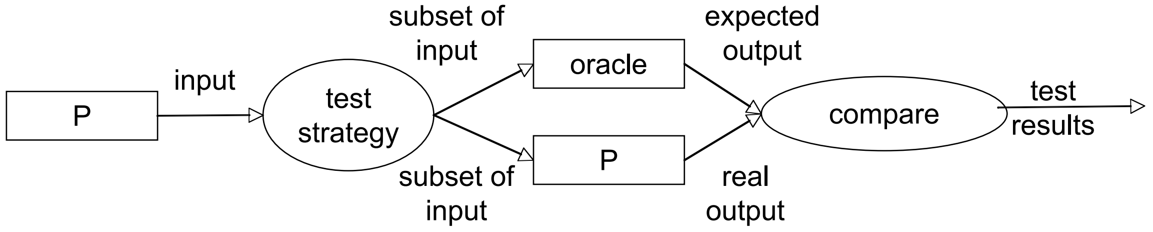
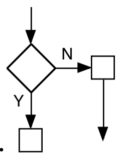


Figure 13.2 Global view of the test process.

Interview question at Microsoft:

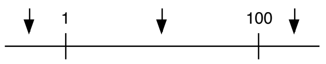
How would you test a function that returns the intersection of two rectangles. What are all the inputs that you would provide to the test function?

Coverage-Based Techniques

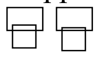


Path-testing or control-flow coverage.
Branch coverage.
Data-flow coverage - how variables are treated down various paths.

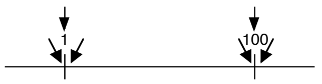
Equivalence partitioning: Break the input into domains and assume that all inputs in a given range are equivalent. (You can do the same for ranges of output.)
For example, your function expects a number between 1 and 100, inclusive.

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: 

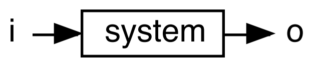
Error-Based Techniques

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.

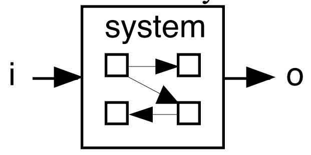
Another way to organize testing approaches:

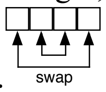
• **Black-box testing** (functional or specification-based). Test cases derived from specifications with little consideration of implementation details.



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. (Cool jargon. Also a great book by Steve McConnell. Good to read and mention at interviews.)

System 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 bug is fixed, you make sure new bugs were not introduced, that the code did not regress (go backwards). "Code churn causes bugs." (That's cool.) 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 16 - User Interface Design

Topics:
• What is the UI?
• What is the user's task?
• Lining up the task with the UI.
• User observation studies.
• UI design across the lifecycle. (Draw it and check off the boxes.)
  Requirements: What is the user's task. Formative testing.
  Design: How will the system support it.
  Testing: Summative user testing.
  Maintenance: Fixing what you got wrong.

<u>What is the UI?</u>
Is this a good interface? Anywhere that the user meets the system. What the user encounters and how the system responds to the user's commands.

<u>What is the user's task?</u>
How to determine.
  Task analysis.
  Context of use: users, tasks, equipment, social environment, physical environment.
    (This is part of your SRS requirements.)
How to notate.

<u>Lining up the task with the UI.</u>
Not easy.
"Mental models" is a somewhat overused term. It means different things to different people, and so you should either avoid using the term, or define it when you use it.
"Mental models" is used in this chapter to mean "how the user expects the system to function and behave." It is contrasted in this chapter with the term "conceptual model", which means how the designers intend for the system to function and behave.
The point is that how the user thinks a system will works is often different from how a designer thinks a system will work. And then there is the true reality of how the human actually attempts to use a system, and how the system responds to those attempts.

<u>User Observation Studies</u>
Formative.
Summative.
The chapter uses the term "User Virtual Machine" (UVM). This term is rarely or never used in the field of user-interface design. I believe that the term is used here to mean everything about the user interface and system, and how to use them, that the user would need to know in order to have complete expertise in using the system. When reading the chapter, I think you can just replace the term with either "user interface" or "system".

From the book:

Heuristic evaluation is based on some definition of usability. Usability is a complex of aspects such as ease of use, ease of learning, ease of recall after a period of not using a system (known as re-learnability), affection, help, likeability, etc. Approaches toward heuristic evaluation provide checklists of characteristics of the user interface ... that describe the different usability aspects. Such a checklist may also be applied as a guideline for making design decisions. Checklists exist in different forms, and often include a technique for calculating usability indexes. Each item may be checked when applicable and, additionally, each item that is diagnosed as non-optimal may give rise to design changes. Example items from such a checklist are (Shneiderman and Plaisant,2004; Nielsen, 1993):

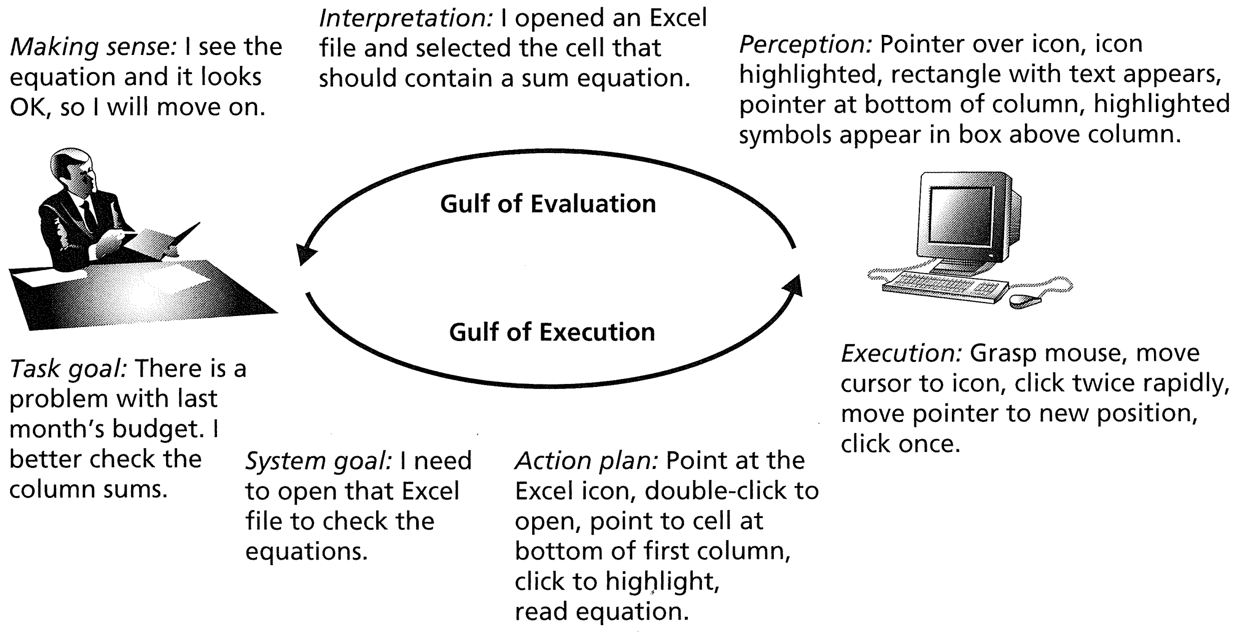
- **Use a simple and natural dialog.** Humans can only pay attention to a few things at a time. Therefore, dialogs should not contain information that is irrelevant or rarely needed. Information should appear in a logical and natural order to ease comprehension.
- **Speak the user's language.** A dialog is easier to understand if it uses concepts and phrases that are familiar to the user. The dialog should not be expressed in computer-oriented terms, but in terms from the task domain.
- **Minimize memory load.** The user should not have to remember information from one part of the dialog to another. Long command sequences, menus with a large number of entries, and uncertainty about 'where we are' hamper interaction. There must be easy ways to ftnd out what to do next, how to retrace steps, and get instructions for use.
- **Be consistent.** Users should not have to wonder whether different words or actions mean the same thing. Metaphors should be chosen carefully, so as not to confuse the user.
- **Provide feedback.** The system should keep the user informed of what is going on. If certain processes take a while, the user should not be left in the dark. A moving widget informs the user that the system is doing something; tl percentage-done indicator informs him about the progress towards the goal.
- **Provide clearly marked exits.** Users make mistakes, activate the wrong function, follow the wrong thread. There must be an easy way to leave such an unwanted state.
- **Provide shortcuts.** Novice users may be presented with an extensive ques- tion-answer dialog. It gives them a safe feeling and helps them to learn the system. Experts are hindered by a tedious step-by-step dialog and the system should provide shortcuts to accommodate them.
- **Give good error messages.** Error messages should be explained in plain language. They should not refer to the internals of the system. Error messages should precisely state the problem and, if possible, suggest a solution.

Chapter 5 - 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



Making sense: I see the equation and it looks OK, so I will move on.

Interpretation: I opened an Excel file and selected the cell that should contain a sum equation.

Perception: Pointer over icon, icon highlighted, rectangle with text appears, pointer at bottom of column, highlighted symbols appear in box above column.

Task goal: There is a problem with last month's budget. I better check the column sums.

System goal: I need to open that Excel file to check the equations.

Action plan: Point at the Excel icon, double-click to open, point to cell at bottom of first column, click to highlight, read equation.

Execution: Grasp mouse, move cursor to icon, click twice rapidly, move pointer to new position, click once.

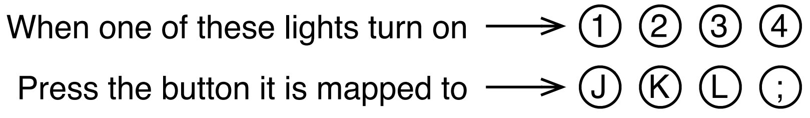
Figure 5.1 Stages of action in a budget problem: choosing, planning, and executing an action, and then perceiving, interpreting, and making sense of the computer's response.

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
J K L ;

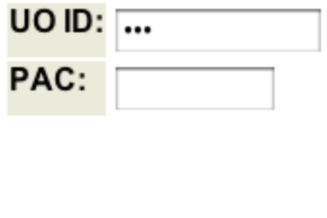
Stimulus-response compatible mappings: 1J 2K 3L 4;
Stimulus-response *incompatible* mappings: 1L 2J 3; 4K

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.)



UO ID: ...
PAC:

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.

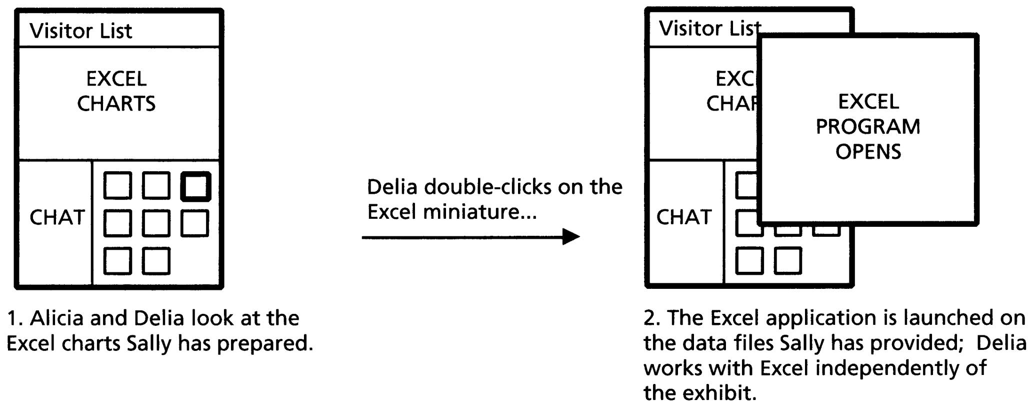
Modes should, in general, be avoided in UI design. Modes are restricted interaction states in which only certain actions are possible. Such as a "modal" dialog box that requires a response before you can do anything else with your computer; some reminders software work this way, such as to alert you 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.

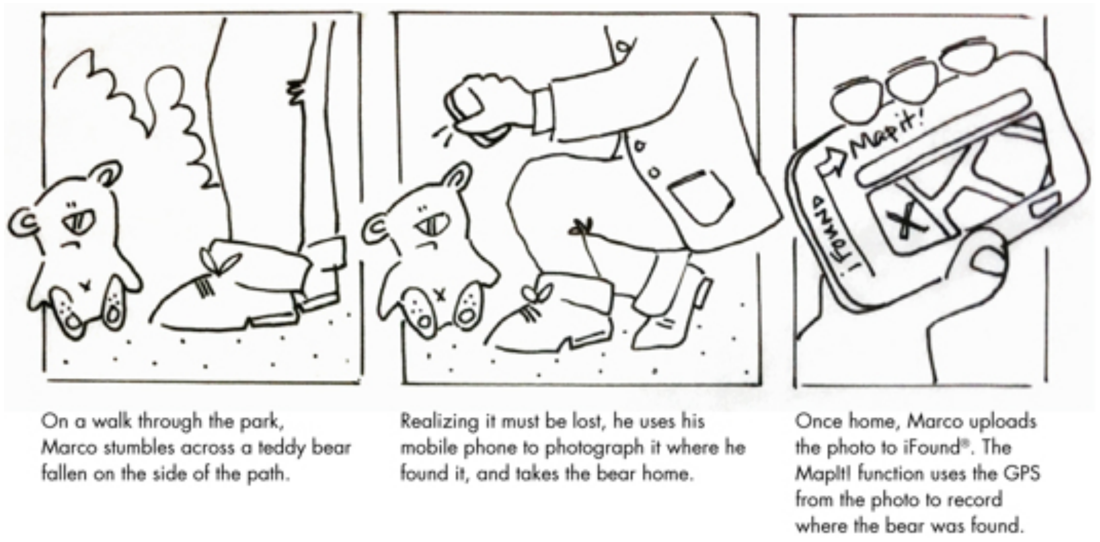
Interpreting System Feedback
Give the user feedback with regards to how they are progressing towards their goals, at multiple time scales, including responding to any input within 100 ms, just to show that the system received your command, but also on the time scale of seconds, showing progress towards 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


Visitor List
EXCEL CHARTS
CHAT
1. Alicia and Delia look at the Excel charts Sally has prepared.
Delia double-clicks on the Excel miniature...
Visitor List
EXC
CHAR
EXCEL PROGRAM OPENS
CHAT
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


MapIt!
iFound
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 - Usability Evaluation

...

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.

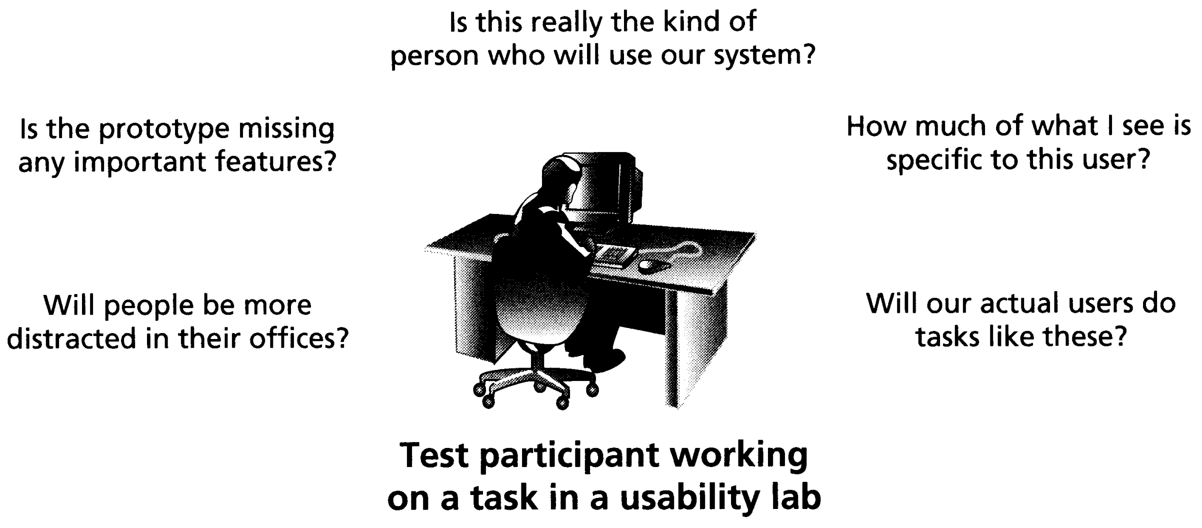

Is this really the kind of person who will use our system?
Is the prototype missing any important features?
How much of what I see is specific to this user?
Will people be more distracted in their offices?
Will our actual users do tasks like these?
Test participant working on a task in a usability lab

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.

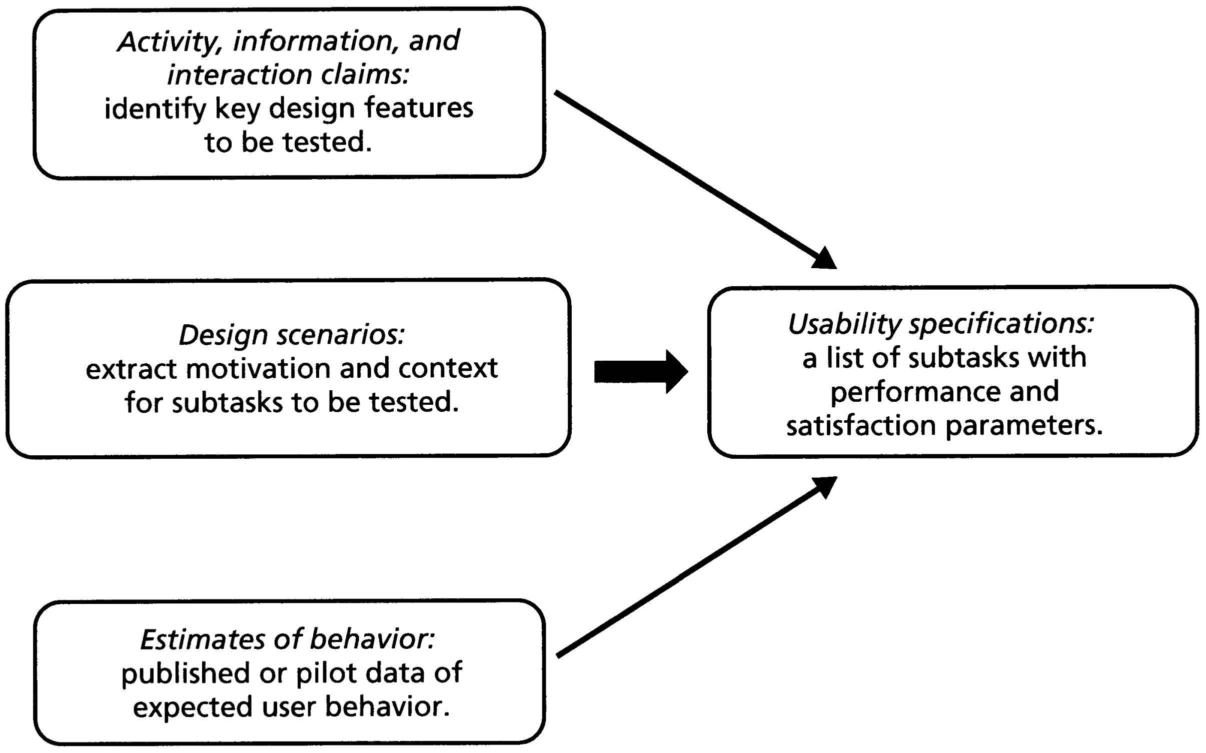

Activity, information, and interaction claims: identify key design features to be tested.
Design scenarios: extract motivation and context for subtasks to be tested.
Usability specifications: a list of subtasks with performance and satisfaction parameters.
Estimates of behavior: published or pilot data of expected user behavior.

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.

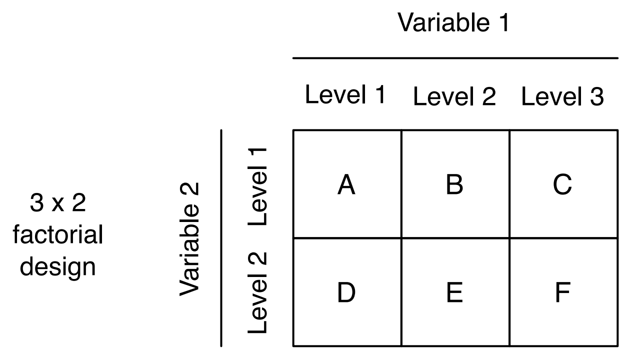
Controlled experiments versus field studies.

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.
Classic human performance measures are speed and accuracy. There is a tradeoff between them.
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.

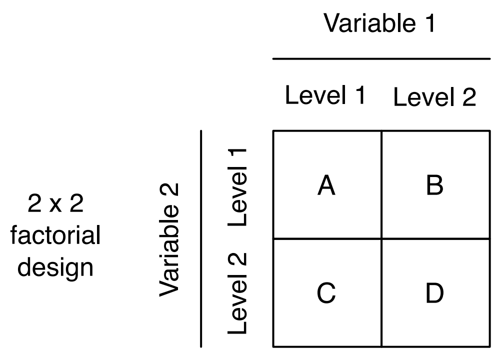
The VSF examples are very good. The assistance policy, for example.

Defining the Independent Variables of your Experiment

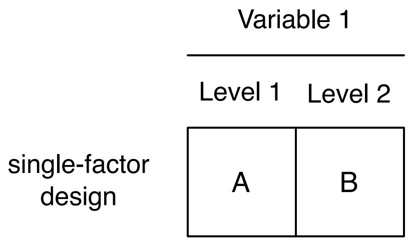
Experiments have **independent variables**, which are also known as **factors**. Each variable has *n* distinct possible settings, or **levels**. A **fully-crossed** experiment collects data on all possible settings of all variables. For example, if you have two variables, one with three levels and the other with two levels, you have a "3 × 2" factorial design, and you can illustrate it like this:



If you have two variables, both with two levels, you have a "2 × 2" factorial design, and you can illustrate it like this:



If you have just one variable, with two levels, you have a single-factor design, like this:



The letters in each box above, A through F, are the **conditions**. The factorial designs above have six, four, and two conditions. For a **balanced** experiment, it must be **fully-crossed**, meaning that you collect data in every **cell**, or every box in the diagram above. Or, in other words, you must collect data for every condition. In general, you should try to collect data from the same number of participants for each condition.