Notes on Reading Van Vliet (2000)

By Anthony Hornof

These are Anthony Hornof's notes from reading Hans van Vliet (2000) *Software Engineering: Principles and Practice*, 2nd edition, John Wiley & Sons. They 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 text that 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."

1.1 What is software engineering?

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, 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.

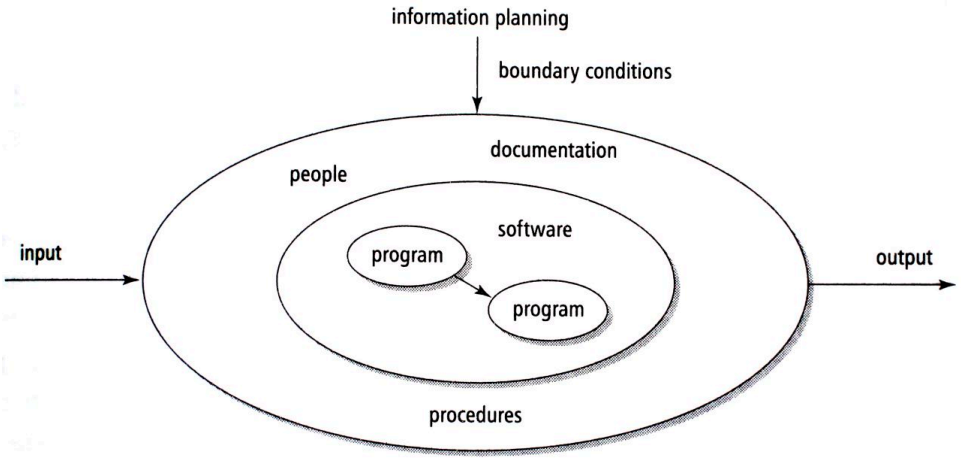


Figure 2.1 The systems view of a software development project

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

2.2 Controlling a SWD project

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

__ continue here

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.

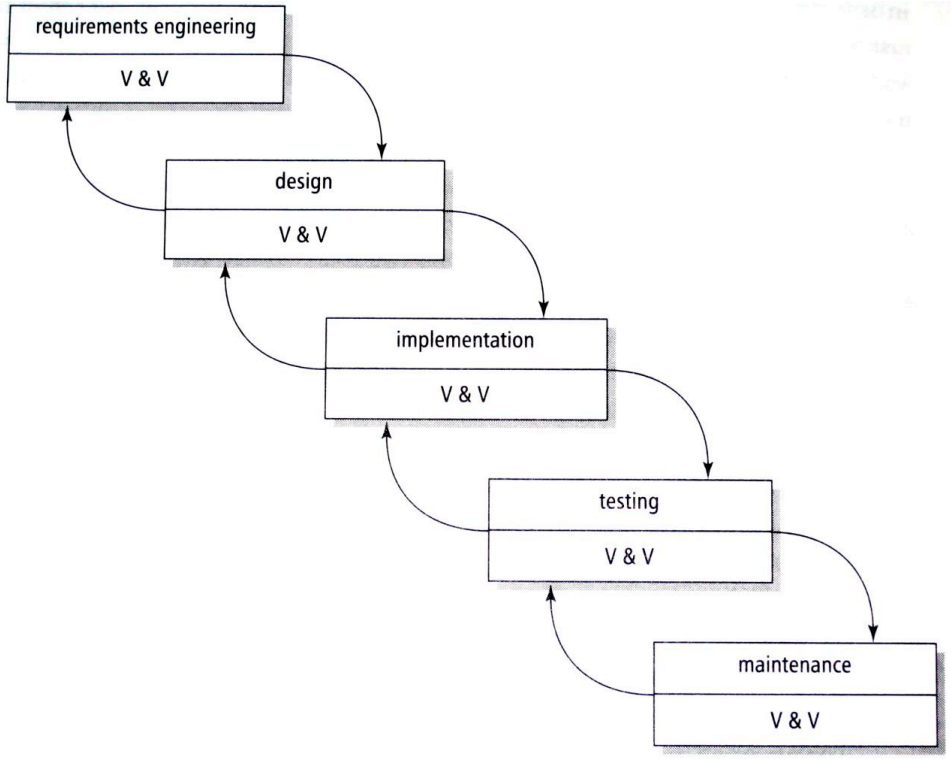


Figure 3.1 The waterfall model

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.)

Prototyping

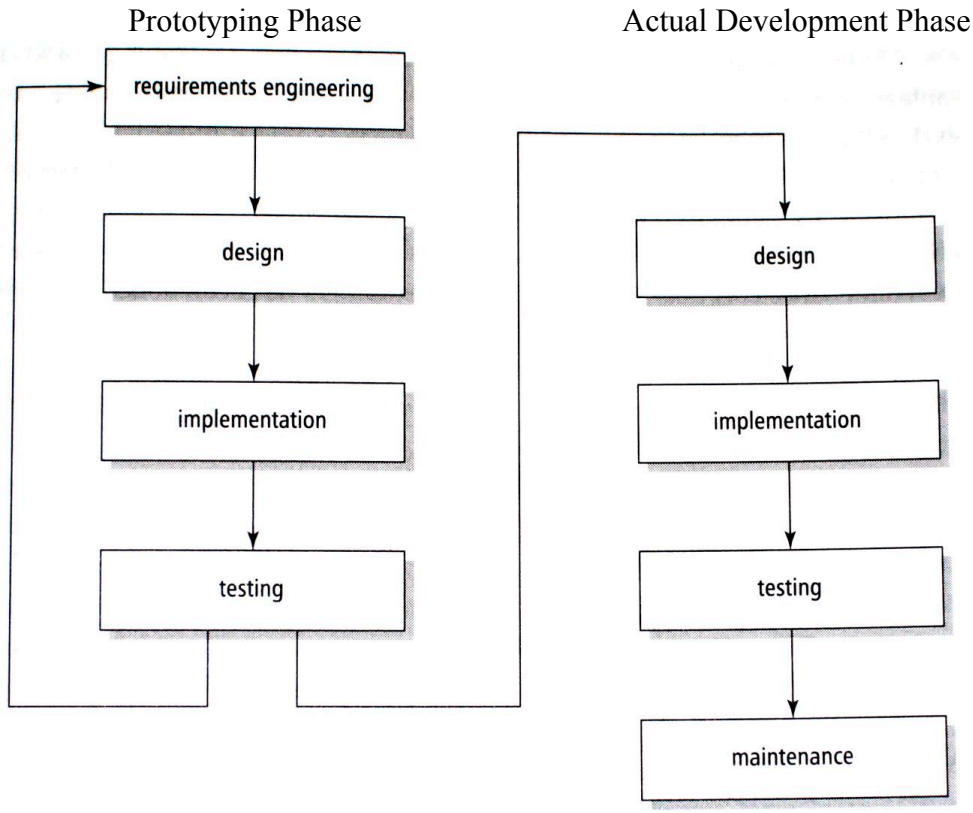


Figure 3.3 Prototyping as a tool for requirements engineering

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 extemely 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 protyping in Figure 3.4 (p.54)

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.

__ read about it

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

---

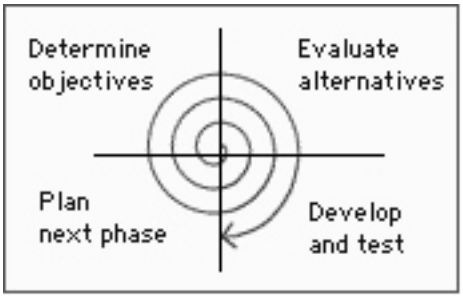
3.5 Maintenance or Evolution?

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 3.5 on p.61)
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.

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

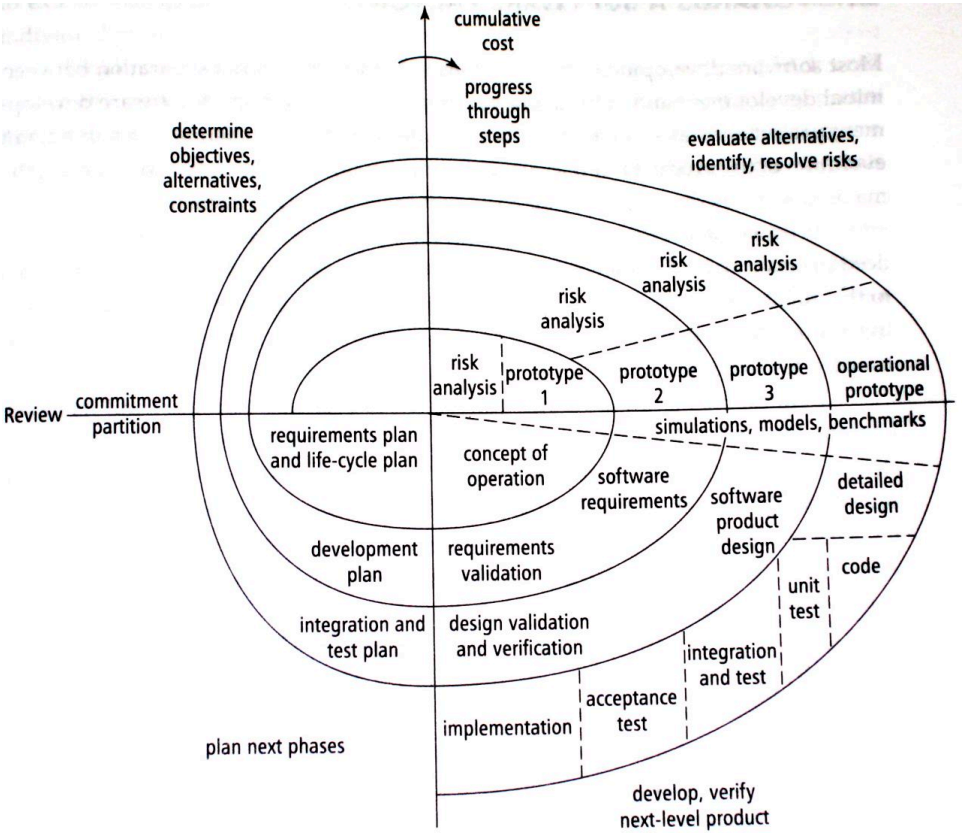
3.6 The Spiral Model



(From course web page)

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 RAD 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

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 beaurocrat 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 reimplement 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?

Market-Driven versus Customer-Driven

"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?

Specification

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.)

Requirements document should be

* Correct. Solving the right problem in the right way.
* Unambiguous. At some level, to all takeholders. 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.

CIS 443/543 Quiz - 2/21/08 **Name** _____
Chapter 9

1. "Most requirements engineering techniques offer little support for market-driven software development." - Do you agree or disagree with this statement? Explain briefly.

2. "The system should be user-friendly and have a fast response time." Is this a good requirement? Explain.

Chapter 10 - Software Architecture

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 global description of a software system 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.

<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 fo 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.)

KWIC-Index Example

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.

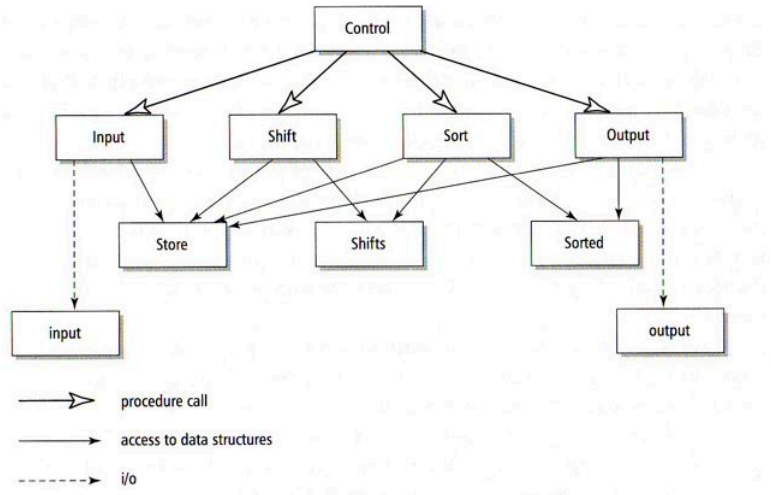


Figure 10.1 Main-program-with-subroutines solution of the KWIC-index program

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.

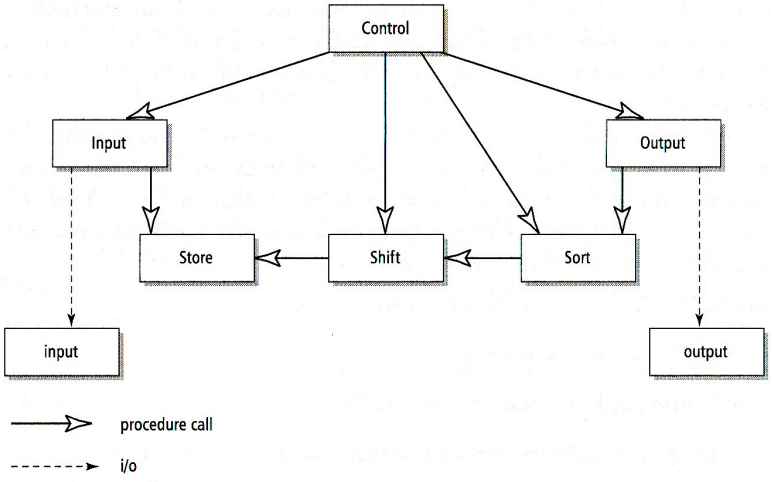


Figure 10.2 Abstract-data-type solution of the KWIC-index program

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.

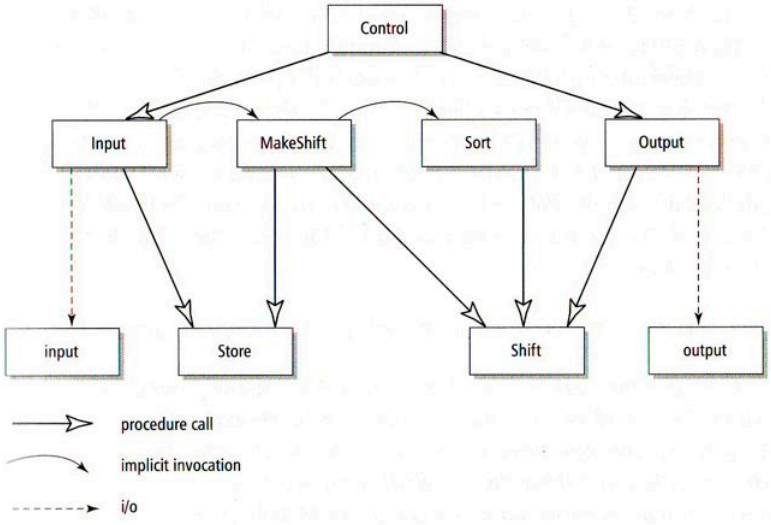


Figure 10.3 Implicit-invocation solution of the KWIC-index program

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

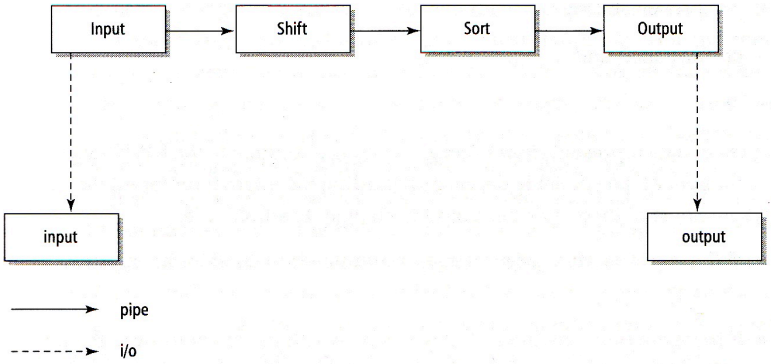


Figure 10.4 Pipes-and-filters solution of the KWIC-index program

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.

There is more discussion of this in Section 10.1.5

Chapter 11 - 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. 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.