Architectural Design

Overview: Basic principles and approaches

Notes:

Design tasks

- External design
  - as seen by end-user
- Architectural design
  - Overall architecture
    - selection of architectural style
    - may not be repeated in each project
  - Subsystem/module breakdown
    - closely related to schedule and team structure
- Module design
  - may be divided into preliminary and detailed design

1

Notes:

External design is closely tied to requirements analysis, but may already include some elements of architectural design and even module design. How much to include is a tricky question, since we have to balance the danger of premature design commitments against the danger of incomplete specification – the correct balance will depend on analysis of risks.

The architectural style is an overall pattern to guide breakdown into modules, and is typically based on experience and familiarity, as well as available tools and components. Examples of architectural styles include

> Transaction processing systems: One (or at most a few) shared information bases, with application functionality being organized as "transactions" that are independent and stateless.
>
> Client-server systems: Division into replicated client-side processing and shared server-side processing. Physical architecture (distribution) motivates a transaction-like processing architecture with a minimum of per-client state in the server.
>
> Discrete event simulation: typical division into a core event queue mechanism (model-independent) and a set of event handlers that respond to events, modify shared state, and generate scheduled events.
>
> Knowledge-based systems: Division into knowledge base and inference engine.

Subsystem breakdown governs (or is governed by) possible build order

Preliminary module design is interface definition, detailed is implementation sketch

Modular design

- Division into modules and subsystems with precisely defined interfaces
- A module is
  - Unit of understanding (fits entirely in one head)
  - Work unit (programmer/designer assignment)
  - Unit of replacement (and firewall for mistakes)
- Subsystems may be modules or collections of modules

2

(c) 1998 M Young CIS 422/522 5/11/98 3

Notes:

The end-product of architectural design should be a division of the system into modules and subsystems with precisely defined interfaces.

A module is:

> A unit of understanding. That is, a designer, developer, or maintainer should be able to understand a module in its entirety, all at once. If I sit down to read parts of a program (for example, to decide how to make some change), I should be able to read a whole module and understand how it fits together. This also implies that a module must be cohesive, not a random collection of details, and it must be reasonably independent of other modules.
>
> A work unit. A module is typically the smallest unit of development that we place on a schedule and assign to a developer. Typically a module should be no more than 1 week of work for 1 programmer, so it makes a good unit for scheduling. This also makes it the unit for configuration control, unit testing, etc.
>
> When we make changes, we generally prefer to replace a module rather than make small changes to many modules (although there are occasionally exceptions to this rule). If we make a design error, we hope that the error can be fixed by modifying a single module.

Subsystems may be modules or collections of modules. An example of a collection of modules is a math library. A math library is not a module, because the whole thing doesn't fit in one head, but in the architectural design of a system we would probably give only a single interface skeleton and then just list all the functions that must be implemented.

Goals of Modular Design

- Intellectual manageability
- Parallel development
  - Module interface specifications must be self-contained, small enough to be understood, and a sufficient basis for refinement, testing, etc. on a unit-by-unit basis
- Evolution & Maintenance
  - Modules tend to localize change; interfaces change less than implementations
- ... *each goal implies desirable properties*

Notes:

Goals —these are really just the consequences and elaboration of the characteristics on the previous slide.

Intellectual manageability — follows from the "unit of understanding" characterization of a module. It requires that a modular design have

High cohesion: Each module "hangs together" and makes sense.

Low coupling: Modules do not depend on each other in complex ways.

Parallel development — the units of work should permit us to schedule several developers to work at the same time without too much interaction.

Evolution and maintenance — If we do a good job in the architectural design, we will localize change both for this project and for others

For maintenance, we will make it easier to locate what must be changed, and to gain confidence that *only* these parts need to be changed.

For longer term evolution, across several projects, we will isolate the changing parts from the parts that can be reused without change, thus facilitating reuse.

The properties we look for in a good modular design are intended to achieve these goals. When trying to judge a design or a design decision, it is often useful to "look behind" properties like coupling and cohesion, or module size, and think explicitly about these goals.

Module hierarchies

- Modules should form a hierarchy (DAG) based on the "uses" relation
- Defn: A uses B if correct functioning of A depends on correct functioning of B.
  - Note subtlety: "correct" functioning defined by nature of the spec
- "Uses" may not be the same as "calls"
  - "Up-calls" are reversed from "uses" relation (example: mouse event handlers)
  - "Plug-in" interfaces (Netscape, Photoshop, etc.) are also reversed

Notes:

The uses (*usa*) relation: A uses B def correct functioning of A depends on correction functioning of B

> Or think of it in a pragmatic sense: I can't build and test A until I've built and tested B. (That's not *quite* the same, but close.) A cycle would mean that nothing can be finished until everything is.

Most of the "uses" relations, in most systems, correspond to either procedure (or method) calls or data structure references. But...

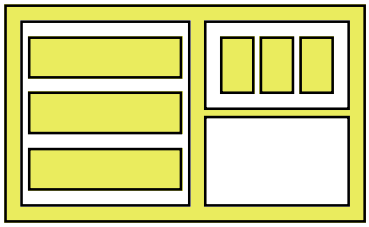
Subtlety in definition

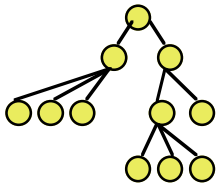
> We can change the "uses" relation without changing what modules do, by changing the specification. For example, we could break a "uses" dependency between a program and a print driver by making it "correct" if it calls the print routines in the right way.
>
> In large and complex systems, it is common to break "uses" dependencies in just this way (although often implicitly). One sign of this is "stubbed" interfaces, where a schedule dependency is broken.

Examples where "uses" is not "calls"

> An "upcall" is a procedure call "up" the abstraction hierarchy, rather than down. A common example is a mouse event handler — the window system calls a procedure in the user's code, but it is the user's code the depends on the window system.
>
> Plug-in interfaces, device drivers, etc. are also places where a use/call relation has been purposely reversed.

Alternate views of hierarchy





- Sometimes hierarchy is explicit, sometimes implicit
  - e.g., levels of detail in a data flow diagram vs. a tree
- A single design may have multiple hiearchies
  - e.g., “uses” vs. “calls”

(c) 1998 M Young CIS 422/522 5/11/98 6

Notes:

Just a note on hierarchical structure — sometimes it is obvious in the notation, and sometimes it is not. For example, a data flow diagram or an SADT chart may have many levels of detail, and those form a hierarchy even though there is no single diagram that shows the hierarchy.

It sometimes happens that a single design has multiple hierarchies, possibly sharing leaves or subtrees at a certain level. This is particularly the case in very large systems, in which we might have, for example:

A directory structure to organize the code.

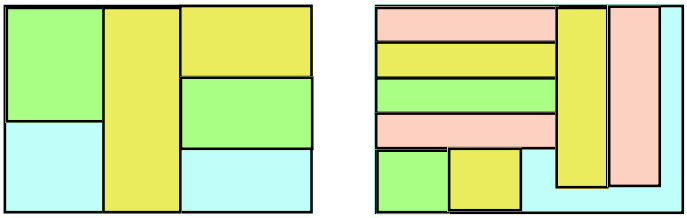
A tree of configurations, or a forest of system variants

The architectural design structure itself

These different hierarchies should be related, but the overlap may not be simply that one is a summarization of the other. For example, we could have two partially overlapping configurations of a single product, or two products sharing a subsystem.

What are the issues?

- The need for modularity is universally agreed, from far back
- The approaches and techniques for modularity are the challenge



(c) 1998 M Young CIS 422/522 5/11/98 7

Notes:

For any one system, there are many different ways it could be divided into modules and subsystems.

Everyone learns, from their very first computer science class, that they should break their programs down into modules. However, an arbitrary breakdown would not meet the goals of intellectual manageability, parallel development, and evolvability.

Choosing a good breakdown into modules is a challenging and creative design activity. There are no simple recipes to follow. A great architectural design looks simple, even obvious, when you see it, but it is not easy to achieve. The best we can do is study some good approaches, study examples, and practice practice practice.

Architectural style selection

Which way to cut?

- The top-level division could be by
  - Steps
    - pipes & filters, passes
  - Layers
    - hierarchy of virtual machines
  - Objects
    - data structures, devices, etc.
- Key considerations are separation of concerns (information hiding), simple and general interfaces, and ability to subset

Notes:

There should be some overall organizing principle to an architectural design. Sometimes that comes from adopting a well-known architecture for an application domain (e.g., the standard division of a compiler into a front-end and back-end), but sometimes we have to use more basic approaches.

We could divide by

> Steps in processing, i.e., “first do this, then do that”. Unix pipelines made up of a set of filters (transformations) are a good example of breaking a task into steps.
>
> Layers, or levels of abstraction (covered in more detail later in the lecture)
>
> Objects, such as data structures or devices (covered in more detail later in the lecture)

To determine what approach is best, we need to explicitly consider how well we can meet the goals of a modular design using a particular approach to division. How well do we isolate the things that are likely to change? Can we produce interfaces that are simple (for intellectual manageability) and general (to survive maintenance and promote reuse)? Will it fit well in the build-plan, allowing parallel development of parts, and incremental development and delivery of subsystems?

Modularity and Process

- We must have an *approach* to creating a modular design
  - May involve backtracking, but there must be a way to make progress
- "Structured analysis," "Object-oriented design," etc. are all guidelines for finding a good modular design

Notes:

In addition to saying what kind of design we want to end up with, we need a constructive approach to creating the design.

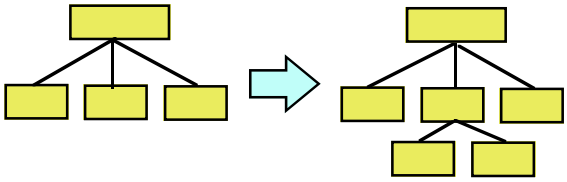
There are no simple, sure-fire recipes. We will make mistakes, and we will have to back up and change some of our decisions in the module breakdown. What we need is some good guidelines for making reasonable choices, getting started and making progress.

There are many "design methodologies" which prescribe particular approaches. Some are more detailed (and rigid) than others. None will guarantee a good design, but they do have the advantage of prescribing some steps which, if performed with skill and good judgement, can result in a good design.

I will not cover particular design methodologies in detail — many books and industrial courses are available for that, and many companies adopt or develop particular methodologies to be usein their projects. These change over time. Ten years ago, "structured" design methods prevailed. Now "object oriented" methods prevail, and in ten years I have no doubt that some other class of methods will be dominant.

Stepwise Refinement

- A (mostly) top-down approach to elaborating a modular structure
  - Refining "steps" in a computation
- Associated with "structured programming," "structured design," "structured analysis" (70's & 80's popular methods)



(c) 1998 M Young CIS 422/522 5/11/98 10

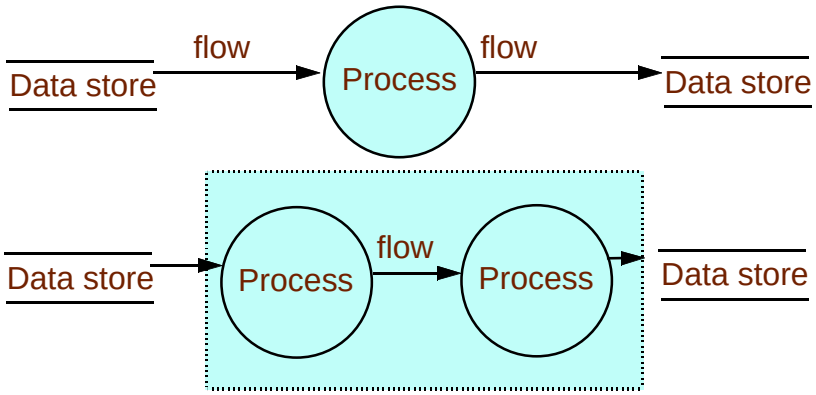
Notes:

The "structured" methods dominant the 70's and 80's were based on the idea of "stepwise refinement." This is a so-called "top-down" method, because it begins with an overall description of a system (or a program), and progressively divides it into smaller and smaller pieces, growing the design hierarchy "downward" from the root toward the leaves.

Usually stepwise refinement divides a step in computation into two or more smaller steps. (The term "stepwise" is because each such refinement is a step in program design, not because the nodes of the design tree are typically steps in processing.

A Step in "Structured Analysis"

- Design represented as data and transformations
- Refine design by elaborating a transformation



(c) 1998 M Young CIS 422/522 5/11/98 11

Notes:

Not only programs, but also systems can be broken down by a process of stepwise refinement. The "structured analysis" methods break down processes into smaller processes, which at some level are whole programs.

A typical example of stepwise refinement structured analysis is "exploding" a process in a data flow diagram, replacing it with a set of processes and flows.

The complexity of module interfaces

- Module interface is a contract
  - Everything a user of a module is permitted to assume
  - Everything a developer of a module is required to ensure
- Stepwise refinement problem: Interfaces are often not as simple as they look
  - Interface is complete input/output relation or pre/post condition, not just the calling convention

Notes:

Every module has an interface and an implementation. The interface is how other modules interact with it.

> A module is a "contract" in the sense that it describes how a module "promises to behave." If A uses B, then the interface of B describes what A is permitted to depend on, and what B promises to do for A.
>
> A module is likewise a contract between developers: Here is what I promise my module will do. Interfaces between modules are therefore also interfaces between people, and they determine how independently the developers can work.

The qualities we want in a modular design lead us to want simple interfaces.

Dividing a step in processing into smaller steps tends to create complex interfaces. That is, the interface between one step and the next may be a complete description of the condition that must have been established by the first.

Interfaces may look simple, but in fact be complex. If we divide up a process in a data flow diagram, the flow is an interface. At one level the flow could look simple, e.g., "customer records." However, a complete description of the dependencies between the records may be much more complex.

"Hidden" Interfaces

- Our *descriptions* of interfaces seldom fully describe interactions among modules
- Examples
  - Storage allocation and deallocation is interaction (through global variables)
  - In real-time systems, use of resources is interaction
  - Bugs propogate information in unintended ways

Notes:

One thing we must be very careful of in architectural design is seeing the real dependencies between modules. Examples:

> Storage allocation: Which modules are responsible for deallocating the storage? This is not apparent in the interface at the level of a programming language like C; it is typically expressed in comments, if at all. But it is part of the contract between modules, and therefore part of the "real" interface.
>
> Resource use: In real-time systems, tasks interact by using resources. (Note how the subtlety of the definition of "uses" crops up again: Using cpu time is not enough to establish a "uses" relation in non-real-time systems, because response time is not part of the specification in those systems.)
>
> In some cases we may want to explicitly consider what happens when part of the system does not work correctly, i.e., how failures or exceptions are handled. This can be a major design consideration in the architecture of some systems with stringent dependability requirements.

Sometimes we can organize a system to reduce the problem of "hidden" or "implicit" interfaces

> Garbage collection (as in Java, Lisp, etc.) drastically reduces the complexity of interfaces involving dynamically allocated data.
>
> Rate-monotonic scheduling changes the interface rule regarding resource scheduling for hard-real-time systems.

Ripple Effects

- Changes during development
  - No design is initially perfect; design changes may be expensive if they have wide effects
- Changes after deployment
  - Not only bug fixes, but evolution of a system
- A module interface is a firewall
  - The "goodness" of a modular design is largely determined the extent to which change is localized

Notes:

A "ripple" is a change that requires more changes elsewhere in the system. In other words, "ripple effects" mean that we have failed to localize changes within modules.

Complex interfaces are prone to ripple effects. It is hard to make a change to a module that does not cause other changes visible at the interface, and thereby "ripple" to other modules.

We can produce better designs by considering "design for change" explicitly, and considering how well a module "hides" design decisions from other modules.

Information Hiding

- Key concept: A module localizes and encapsulates a design decision
- Method: Identify anticipated changes
- Examples
  - Hardware encapsulation modules: Publish abstract properties of a device, hide details of hardware interface
  - Data structure modules: Publish properties of abstract data type, hide implementation details

Notes:

"Information Hiding" was proposed (by Parnas) as an alternative to stepwise refinement

> Paper: "On the criteria to be used in decomposing programs into modules," D.L. Parnas, Communications of the ACM, 1972. This paper seems quite dated now, and today the example program is not very convincing.

Principle: Associate a "secret" with a module, i.e., we start with the expected changes and then design the modules to accomodate and localize those changes.

Examples

> Hardware encapsulation — this is why the A7 design distinguishes between "essential properties" of data and "arbitrary details" of devices. The real meaning of "arbitrary" is "likely to change", whereas it is considered less likely that essential data characteristics (visible through the module interface) will change.
>
> Data structure modules — data structures (and algorithms to manipulate them) can be encapsulated in modules, permitting local changes. We can, for example, produce a first implementation using very simple data structures, and later replace a few critical data structures to improve the performance of a system. (Note the importance of minimizing "hidden" interfaces to make this possible.)

Some ways of factoring

- Data abstraction
- Layering virtual machines
- Mechanism vs. Policy
  - Mechanism: simple, application-independent layer of functionality
  - Policy: an application-specific use of mechanism

4

(c) 1998 M Young CIS 422/522 5/11/98 16

Notes:

If our goal is separation of concerns, then breaking down the design into modules means “factoring” the individual concerns (or design secrets) into different subsystems or modules.

Again, we need some constructive guidelines on how to accomplish the factoring. Here are a few that are widely useful.

Data abstraction (abstract data types, objects, etc.) is the most widely known and practiced method. You are probably already at least somewhat familiar with it. (Next slide ...)

Layered systems, where each “layer” is a so-called “virtual machine,” is a general structuring method useful particularly for subsystems larger than individual abstract data types. (More later ...)

Mechanism/Policy splits are a particular kind of layering decision. They are difficult to choose initially (often the correct split is only apparent after we’ve seen several changes to a system, or variations in a family of systems), but they are extremely effective ways to deal with very “volatile” or rapidly changing aspects of a system.

Data Abstraction

- Design method: Modularize based on key data structures
- Algebraic specifications: Concise statement of properties of a *set* of operations
  - "push" is not a module, but "stack" is
- Realization in programming languages
  - Clu, Modula-2, Ada83, ... module constructs with opaque user-defined types
  - Modula-3, C++, Ada95, Java ... objects

Notes:

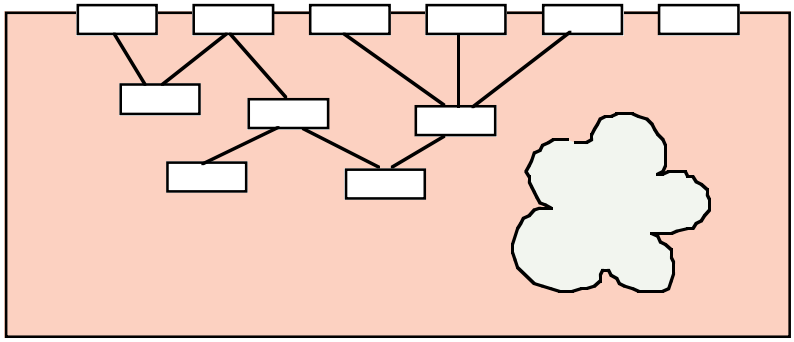
Design based on "objects" or "abstract data types" is now widely applied, thanks partly to the popularity of object-oriented languages which make a mapping from ADT-based designs to implementation classes fairly straightforward.

An advantage of ADT modules is that we have well-developed ways of describing their interfaces.

> Signatures of methods or procedures are NOT full descriptions of the interfaces; if that's all we have, then we have a serious problem of hidden interfaces. Algebraic specifications provide a concise and precise way of specifying interface semantics.
>
> Note that if we considered individual operations like "push" as modules, they would have very complex interfaces. Grouping all the accesses to an abstract data type permits much simpler interfaces.

Virtual Machine Abstraction

- Virtual machine presents an abstract interface to state, which can be modified (only) through operations



(c) 1998 M Young CIS 422/522 5/11/98 19

Notes:

A “virtual machine” encapsulates design decisions, much like an abstract data type, but may not encapsulate a single abstract data type

It is typically larger than an ADT, often a subsystem rather than an individual module. It may be composed of many related abstract data types or object classes.

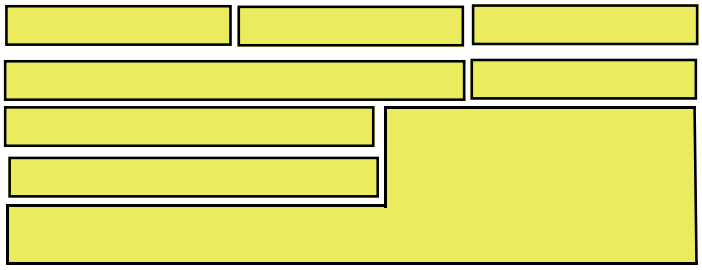
Sometimes a virtual machine encapsulates some state; in this way it can be more like an individual object than a class. For example, PostScript defines a single virtual machine abstraction of a printer.

The interface of a virtual machine may be relatively complex, compared to an ADT. However, it should be a coherent whole, simpler than the collection of its parts.

Virtual machines sometimes provide procedural abstractions rather than (or in addition to) data abstractions. For example, “curses” provides a virtual machine interface to terminals, and “quickdraw” provides a virtual machine interface to the Macintosh screen, and X provides a virtual machine interface to graphical terminals.

Layered Systems

- A modular organization principle
  - developed primarily in OS & networking
  - "virtual machine" abstractions



Notes:

Virtual machines are often "layered"

The meaning of a layers diagram is:

> Vertical adjency implies that the higher module can use services of the lower module. No other uses are possible.

The meaning of a layers diagram is very close to a "uses" hierarchy, but may not be quite the same. A virtual machine in a layer may have a "downward" interface as well as an "upward" interface. The "downward" interface describes dependence on a service, but not necessarily on an implementation of that service.

> For example, we would draw the device-dependent part of X below the device-independent part. X depends on a device-dependent part, but not on a particular implementation of that part.

"Little languages"

A way of packaging virtual machines

- PostScript
  - Uniform interface to different printing mechanisms
- Spreadsheets
- Sed, ed, awk ... package pattern matching and string processing library
- Embedded interpreters in Emacs, Autocad, ...
- TCL/TK shell languages

9

(c) 1998 M Young CIS 422/522 5/11/98 21

Notes:

Because virtual machines may have (relatively) complex interfaces, we may need a better way of presenting those interfaces than a large library of procedure calls.

> Some virtual machines are packaged as procedure libraries, e.g., the Posix interface to Unix; sometimes the same virtual machine is packaged both as library and in another form, such as a little language.

Special purpose programming languages, or "little languages" for some special task, are a common way of presenting virtual machine interfaces.

> PostScript is a hardware encapsulation "virtual machine." It's machine state includes the graphics state, the current path, the dictionary state, etc. It includes several abstract data types such as "path", "dictionary", "color", etc. Although fairly complex, it presents a uniform high-level view of many different printers, effectively isolating printer characteristics from application programs.
>
> Sed, ed, and to some extent even awk can be thought of as ways of presenting an interface to the regular expression libraries of Unix.

Little languages are also a good example of a "mechanism/policy" split

> We make a mechanism/policy split to protect against some particularly volitile aspect of design.
>
> Typically we have a fixed mechanism, and the split is to give us a very flexible way of handling new policies. A table-driven program, or a language interpreter, is mechanism while table contents or little language "program" are policy.

Knowledge based systems

Mechanism vs. Policy

- Mechanism
  - "Inference engine" is a domain-independent mechanism for selecting and executing rules
- Policy
  - A particular knowledge-based system combines a highly application-specific "knowledge base" (collection of rules and facts) with the inference engine.
- In this case, mechanism is a "virtual machine"

6

(c) 1998 M Young CIS 422/522 5/11/98 22

Notes:

Mechanism/policy splits are often the result of experience in a domain, recognizing that some core functionality can be implemented once and used in many different ways. “Expert systems” or “knowledge based systems” are a typical example. The inference engine (say, backward chaining rule interpreter) is mechanism, and the rule set (say, Mycin rule base for disease diagnosis) is policy.

Mechanism vs. Policy: Paging

- Mechanism: Page in, page out
- Policy: Page replacement policy
- Page replacement policy can be changed without altering paging mechanism
- Paging mechanism can be changed (e.g., page caching) without altering replacement policy

5

(c) 1998 M Young CIS 422/522 5/11/98 23

Notes:

The "policy/mechanism" split (or at least the name) originated in operating systems, which are typically structured as hierarchies of virtual machines.

Virtual memory paging is a classic example of policy/mechanism.

Mechanism: Low-level service for moving pages to and from disk

Policy: When to move pages, and which pages to move

In the case of paging, either policy or mechanism can be changed independently (in some operating systems in which the "downward" interface from policy to mechanism has been carefully designed.)

Page replacement policy can be changed without changing the paging mechanism (on almost any operating system).

Paging mechanism can be changed without modifying the policy part (sometimes).

Micro-kernel operating systems are carefully designed with "downward" interfaces, and limit dependencies by moving mechanism part out of the kernel of the operating system and into service modules. As a result, they can be configured with different kinds of paging mechanisms, such as slow RAM or network paging in place of disks.

Design Factoring with Shared Data Structures

- Key data structures may define system interfaces
  - In place of the familiar procedure call interface
- Modularity principles are realized through data design
  - Separation of concerns, layering, etc.

Notes:

(Note: This slide and notes were drafted in 1996 when the course project at DEI, Universidad Degli Studi di Padova was electronic exchange of "business card" information (essentially the same domain addressed in the recent "VCARD" standard.) I thus wanted to explicitly explore the idea of interfaces and modularity when almost all the core design decisions were in the design of a textual format for information exchange.)

Interface complexity may be hidden in complex data structures, especially text.

Textbooks talk about designing program modules; I have yet to see a chapter on designing textual data for modular systems. Can we apply the same principles? I think so, but I am only beginning to sort this area out in these terms.

> Separation of concerns: Does the shared data permit a program that "knows about" some design decisions and not others? We may need to look explicitly at the program structures to properly consider this, and we may need to consider variant programs. For example, thinking only about browsers is not enough to see where html succeeds and fails in supporting separation of concerns.
>
> Layering: Data, particularly textual data, may be viewed at different levels of abstraction. For example "string of text", "properly parenthesized list", "Lisp S-expression" may all describe the same piece of data. We should be able to describe the data structure at levels corresponding to abstractions in processing.

Data as Interface: Examples

- RFC822 (electronic mail)
  - A success: replaced many 1-1 interfaces with an N-N interface standard
  - A failure: did not separat body-content encoding from head/body parsing
    - the notorious >from problem
- MIME (electronic mail w/ encapsulation)
  - separation of issues: transport encoding, content type, recognition of parts
  - reverse a calling relation from uses (parts handlers)

Notes:

Examples of the relation between textual structure design and program are not too difficult to find. Since many application-level internet protocols exchange textual data, the internet standards documents are rich sources of both successful design and mistakes.

RFC822 defines the standard format of email messages, and is one of the older internet standards. It was a success from the standpoint of establishing a single interface among many different e-mail programs, but as one would expect in a “first try”, there were a few mistakes. The best known of these is the “>from” problem, which results because the functionality of separating fields in an email message (the body is just another field) depends on an irreversible modification to the contents.

> This suggests a technique (or principle?) we can use in designing textual structures: Transparency.

MIME is essentially an extension to RFC822, providing ways to enclose many kinds of content besides text. It is much more recent, and reflects a lot of accumulated experience about defining textual structures (also it is a really excellent example of design, and the MIME RFC is worth reading as an example of a first-rate specification document.) MIME specifically identifies concerns that should be separated, and makes sure that they are independent.

MIME also provides for reversing a calls/uses relation, by identifying a caller. Second technique: Use metadata to break dependencies.

Text as Interface: Abstraction levels

- Lexical level (e.g., ascii text or token streams)
  - Lowest level; should not depend on syntax or semantics
- Syntactic level (e.g., html tag tree)
  - May separate abstract syntax from concrete syntax
  - May separate "skinny" syntax from "correct" syntax
    - ex., html content model
- Semantic level(s)
  - May separate multiple issues

Notes:

One kind of textual data we have a lot of experience with is programming languages, and we can use our experience in that domain to suggest some good abstraction levels for defining textual data.

The standard lexical/syntactic/semantic levels distinction is a good one, and we should use it. Just as in programming languages, these should be kept distinct, with dependencies only "downward".

Syntactic structure can further be divided into "skinny" and "fat" structure, depending on how much semantic information we need. This is actually fairly rare in programming languages, but it is very useful where it exists. For example, a Lisp program can be viewed as S-expressions (the skinny syntax) without any knowledge of Lisp semantics, and this is one reason powerful Lisp programming tools predated similar tools for more syntactically complex languages.

HTML is an imperfect example of skinny vs. fat semantics: There is a natural tree form that you can *almost* derive without knowing the particular tags and content model. Unfortunately, the distinction between container and non-container tags (the latter including <hr>,
, and <img>) means that a tool must know the content model to manipulate the structure, even if the content model is not important to it. Thus separation of concerns is not complete.

We may have separation of concerns at semantic levels, as well, and again the question to ask is: Can I perform one kind of semantic processing without knowledge of another?