Design

CSE 403 - Spring 1997

Outline

- General stuff
- Decomposition & Abstraction
  - Coupling & Cohesion
- Information hiding & layering

Design

- Determining a realization that will satisfy a given specification.
  - Not only the functional specifications.
  - But also achieving other requirements on qualities:
    - performance, safety, ease of evolution, etc.
    - others?
- How to select an effective design from the (generally unbounded) space of possible designs?

It is HARD!

- Effective software design is a difficult activity.
- Some claim it is a matter of:
  - engineering, art, craft.
- The textbook discusses how design should simultaneously be a disciplined and a creative activity.
  - This requires significant experience, which can only be gained slowly.
- Some basic principles and guidelines to be put into practice.
  - Mileage will vary.
  - But, don't be discouraged.

Decomposition

- A design must satisfy the specification.
  - Including functional requirements, performance, time-to-market, etc.
- Managing complexity.
  - Manage the complexity of a (even modest) system by decomposing the specification into a larger collection of smaller pieces.
    - Complexity per piece decreases.
- These smaller pieces are called modules and are the central entities of a design.
  - We are also interested in the interactions among these modules as well.

Benefits of Modularization

- Modules can be worked on independently.
  - Separate developers, testers, etc.
- Modules can make the system more understandable.
  - Externally and internally.
- Modules can be specified separately from the implementation.
  - They provide interfaces that act as contracts, which help in:
    - later maintenance. How?
    - simplifies the design process by allowing to focus on the interfaces.
- How to decompose into modules?

Criteria for Decomposition

- What criteria should we use to guide ourselves to decompose systems into what pieces.
  - Ease of change/evolution, ease of reasoning, understandability, safety, time-to-recompile/relink, managerial concerns, portability, traceability to specification, etc.
  - Others?
- The textbook [GJM] claims ease of change is the most important criterion.
  - True, it is important.
  - But, it probably can not be claimed to be the most important, without considering the nature of a system being built.

Abstraction

- What is it and what role does it play in software design?
  - Functional abstraction
  - Data abstraction
  - Control abstraction

Functional Abstraction

- Functional abstraction uses a name (and possibly parameters) to represent a useful computational activity.
  - Procedures, functions and libraries, etc.
  - Every reasonable language supports functional abstraction.
- Note that the original motivation for procedures was as a mechanism to save memory, not to abstract.
  - Possible to use procedures and functions, but not to represent an abstraction.

Data Abstraction

- In data abstraction, a name, a collection of (hidden) data, and a set of operations are bundled together.
  - Abstract Data Types (ADT's)
- Most languages do not directly support data abstraction.
  - But Ada and CLU are examples languages that do.
  - Also, object-oriented languages.

Control Abstraction

- Control abstraction gives a name to a construct that sequences computations.
- Most languages provide some basic control abstractions, e.g.,
  - if-then, if-then-else, while, repeat loops, etc.
- Few languages allow the definition of new control constructs.
  - Common Lisp and other functional languages.
  - How about object-oriented languages?

Abstraction & Language Support

- Obviously, it would be nice if the language we use supported the type of abstractions we need to use.
- But this is not true always in life.
- That should not prevent us from using abstraction.
  - Say, we wanted to follow the discipline of ADT in C.
  - What do we need to do?
  - What are the pitfalls we need to watch out for?

Separation of Concerns

- Deal with different aspects of a problem separately.
  - Aspects of a software product:
    - reliability, space and time efficiency, interactions with environment, UI, etc.
  - Aspects of software development process:
    - development environment, team organization and structure, scheduling, control procedures, etc.
- What concerns do we have in software design???

Separation of Concerns

- Sometimes, issues are so intertwined that they can not be separated.
  - Must account for the effects of decisions made for one issue have on others.
  - E.g., Atomic vs. non-atomic transaction and performance and correctness issue.
- Separation of concerns may lead to separation of responsibilities.
  - Utilization of different skill sets for better performance.

Generality

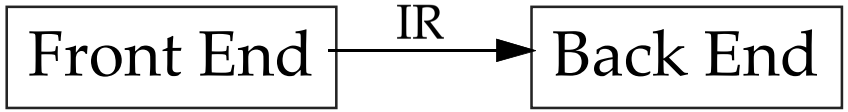
- Is writing a more general component (module) always a better thing in software engineering, since you end up with a part that need not be rewritten as often?

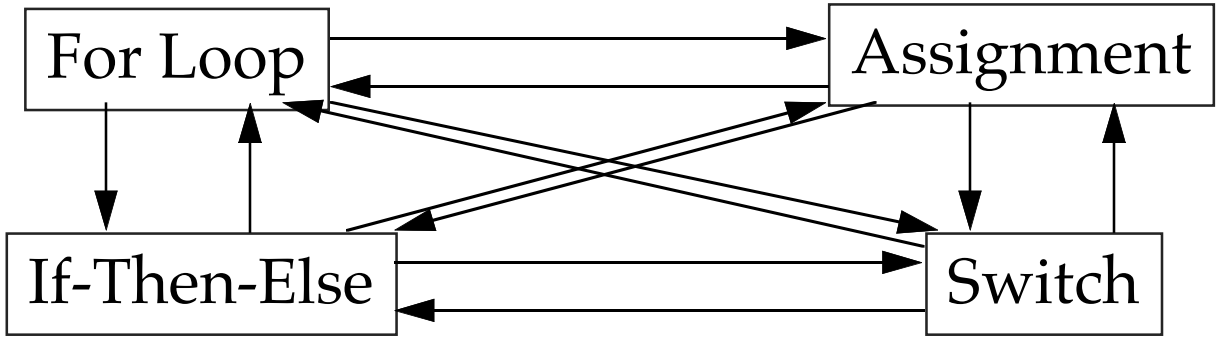
Coupling & Cohesion

- These criteria can be considered (in part) in terms of:
  - What kinds of pieces are defined by subsystems?
  - What kinds of relationships among the pieces.
- The "kinds of pieces" can be characterized by the notion of *cohesion*.
  - Cohesion represents why specific elements are bundled together.
  - Intra-module.
- The "relationships" can be characterized by the ways in which modules depend on one another---*coupling*.
  - Inter-module.

Coupling

- Coupling is of fundamental importance in designs.
  - Separating into subsystems that are actually tightly interrelated may not have the desired benefits.
  - Such as?
- Compiler example (at the top level):





Types of Coupling

- Different types of coupling between modules.
- Module A may modify the code of module B.
- Modules A and B may share global variables.
  - M modules that share N global variables have $O(NM^2)$ couplings.
- Module A may access the internal state of module B.
- Module A accesses module B.
- Module A accesses module B by name
  - Calling an exported procedure.
  - Accessing exported data elements or types.

Coupling Affects...

- The ease of understanding interfaces
- The ease of managing development
- The ease of change
- The ease of reuse

Cohesion

- Cohesion, in contrast to coupling, considers the relationships among units within a module.
  - Why do we place procedures, data and types together in a module?
- Coupling and cohesion are tightly related, since selecting what goes into module affects the relationships between modules.

Types of Cohesion

- *Coincidental*: elements have no meaningful relationships.
- *Logical*: elements perform similar functions.
- *Temporal*: elements used at about the same time.
- *Communicational*: elements access the same data.
- *Sequential*: elements are in data-flow relationships
- *Functional*: elements cooperate to execute a single function.

Balancing Cohesion and Coupling

- Low Coupling, High Cohesion.
  - "Tight" cohesion is generally considered to be desirable.
  - "Loose" coupling is generally considered to be desirable.
- In the extreme, this would mean that the ideal system decomposition would be to have a set of modules that are entirely decoupled.
  - Why is this unreasonable?

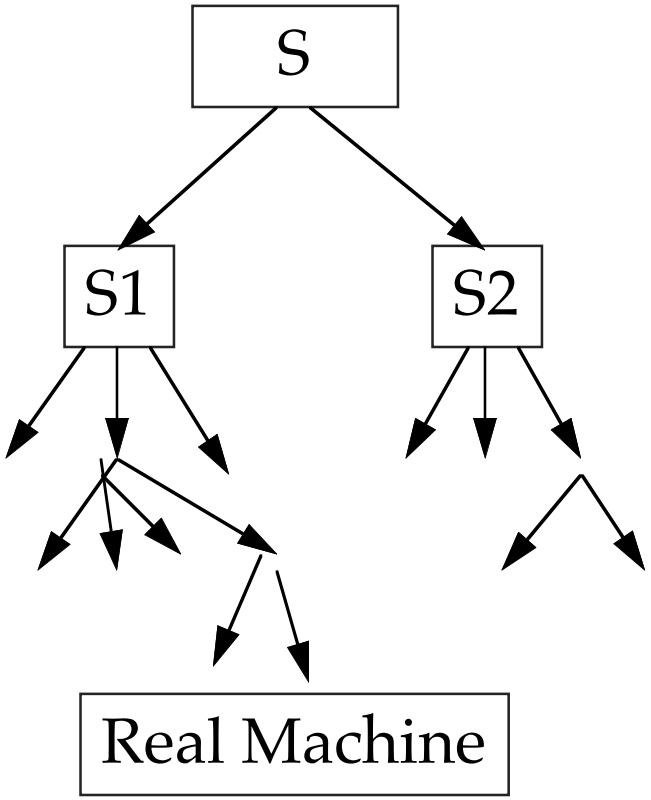
Top-Down Design

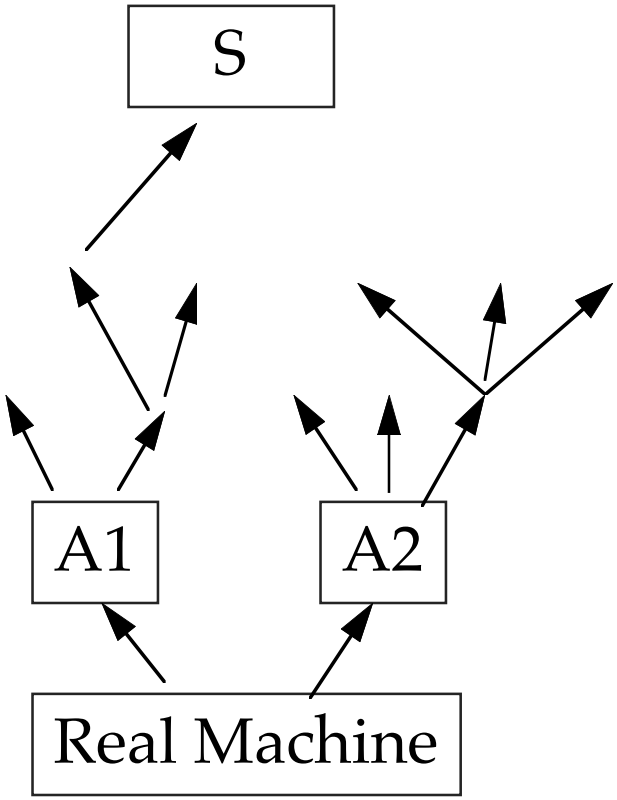
- Top-down design starts from the specification and successively refines into "more realizable" descriptions.
- The intent is that one of the successive refinements is easy to map to an implementation that is directly executable on a real machine.
- Dangers of top-down design:
  - It implies you are considering doing only one thing.
    - Reuse?
  - You may not get closer to the real machine during each refinement.

Bottom-Up Design

- Bottom-up design starts from the available, executable elements and refines additional abstractions.
- The intent is to define abstractions that are increasingly useful for realizing the specification.
- Dangers of bottom-up design.
  - You may create too much generality in your abstractions, causing decreased efficiency (and perhaps confusion.)
  - You may keep building abstractions that still don't permit you to realize your specification.

Pictorially...





Middle-Out

- In reality, it is rare that a design takes place entirely bottom-up or top-down.
  - Normally, there is a mixture of refinement and abstraction.
- Keeping track of the heuristics and reasons for the search is important.
  - To prevent later difficulties during enhancement.
  - "A Rational Design Process: How and Why to Fake it." [Parnas & Clements 1986]

Function- *vs.* Data-Oriented Design

- What abstractions should be stressed during design?
- Design approaches tend to have a primary focus on either functions or data.
  - Regardless of the primary focus, both data and function must be considered.

Function- *vs.* Data-Oriented Design

- Function orientation (functional decomposition)
  - Focus on the verbs: what is it that will be done to things.
  - Top-down: decompose these verbs into simpler verbs.
  - Bottom-up: compose the lower-level verbs into more complex verbs.
  - E.g., sort
- Data orientation (data decomposition)
  - Focus on the nouns: what is it that things will be done to?
  - Top-down: represent these nouns as compositions of simpler nouns.
  - Bottom-up: compose new nouns from simpler nouns.
  - E.g., customer
- Which is more stable?