Software Design

Goals of Design

- Decompose system into components
  - i.e. identify the software architecture
- Describe component functionality
  - informally or formally
- Determine relationships between components
  - identify component dependencies
  - determine inter-component communication mechanisms
- Specify component interfaces
  - Interfaces should be well defined
    - facilitates component testing and team communication

Decomposition

WHY ?
Handle complexity by splitting large problems into smaller problems, i.e. "*divide and conquer*" methodology

1. Select a piece of the problem
   - initially, the whole problem
2. Determine the components in this piece using a design paradigm
   - e.g. functional, structured, object-oriented, generic, etc.
3. Describe the components interactions
4. Repeat steps 1 through 3 until some termination criteria is met
   - e.g., customer is satisfied, run out of money, etc. ;-)

A Component Is ...

- ... a software entity encapsulating the representation of an abstraction
- ... a vehicle for hiding at least one design decision
- ... a "work" assignment
  - for a programmer or group of programmers
- ... a unit of code that
  - has one (or more) name(s)
  - has identifiable boundaries
  - can be (re-)used by other components
  - encapsulates data
  - hides unnecessary details
  - can be separately compiled

What is Good Design?

- The temptation of "correct design"
  - insurance against "design catastrophes"
  - design methods that guarantee the "correct design"

> *A good design is one that balances trade-offs to minimize the total cost of the system over its entire lifetime*
> *[…]*
> *a matter of avoiding those characteristics that lead to bad consequences.*
> **Coad & Yourdon**

There is no correct design! You must decide!

- Need of criteria for evaluating a design
- Need of principles and rules for creating good designs

Modularity

- A modular system is one that's structured into identifiable abstractions called components
  - Components should possess well-specified abstract interfaces
  - Components should have high cohesion and low coupling

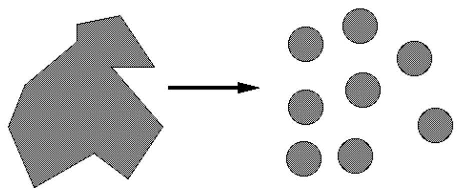
> *A software construction method is modular*
> *if it helps designers produce software systems*
> *made of autonomous elements*
> *connected by a coherent, simple structure.*
> **B. Meyer**

Meyer's Five Criteria for Evaluating Modularity

- **Decomposability**
  - Are larger components decomposed into smaller components?
- **Composability**
  - Are larger components composed from smaller components?
- **Understandability**
  - Are components separately understandable?
- **Continuity**
  - Do small changes to the specification affect a localized and limited number of components?
- **Protection**
  - Are the effects of run-time abnormalities confined to a small number of related components?

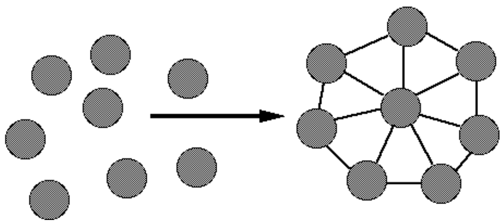
1. Decomposability

- Decompose problem into smaller sub-problems that can be solved separately
  - **Goal**: Division of Labor
    - keep dependencies **explicit** and **minimal**
  - Example: Top-Down Design
  - Counter-example: Initialization Module
    - initialize everything for everybody



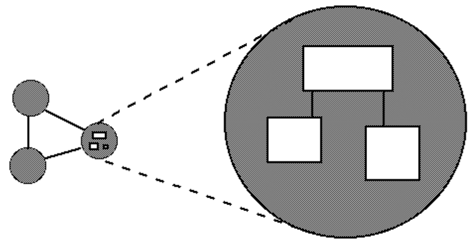
2. Composability

- Freely combine modules to produce new systems
  - **Reusability** in different environments → components
  - Example: Math libraries; UNIX command & pipes



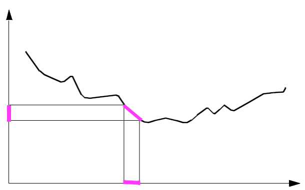
3. Understandability

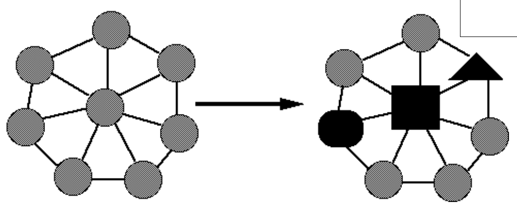
- Individual modules understandable by human reader
  - Counter-example: Sequential Dependencies (A | B | C)
    - contextual significance of modules



4. Continuity

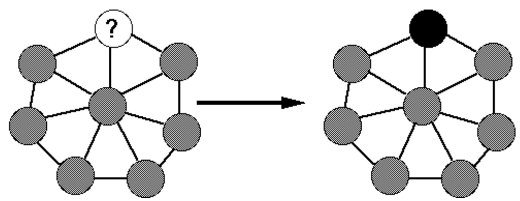
- Small change in requirements results in:
  - changes in only a few modules does not affect the architecture
  - Example: Symbolic Constants
  - Counter-Example: non-uniform access





5. Protection

- Effects of an abnormal run-time condition is confined to a few modules
  - Example: Validating input at source
  - Counter-example: Undisciplined exceptions



Meyer's Five Rules of Modularity

- Direct Mapping
  - consistent relation between problem model and solution structure
- Few Interfaces
  - Every component should communicate with as few others as possible
- Small Interfaces
  - If any two components communicate at all, they should exchange as little information as possible
- Explicit Interfaces
  - Whenever two components A and B communicate, this must be obvious from the text of A or B or both
- Information Hiding

1. Direct Mapping

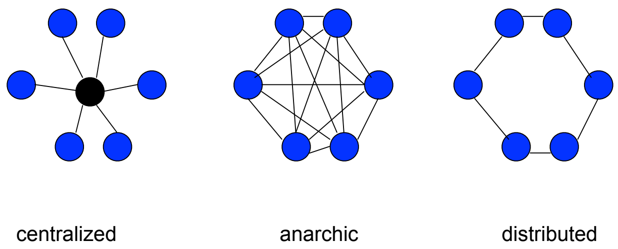
- Keep the structure of the solution compatible with the structure of the modeled problem domain
  - clear mapping (correspondence) between the two

Impact on:

- Continuity
  - easier to assess and limit the impact of change
- Decomposability
  - decomposition in the problem domain model as a good starting point for the decomposition of the software

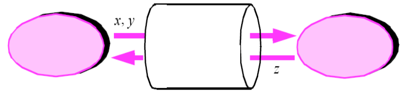
2. Few Interfaces

- Every module should communicate with as few others as possible
  - rather n-1 than n(n-1)/ 2
  - Continuity, Protection, Understandability, Composability



3. Small Interfaces

- If two modules communicate, they should exchange as little information as possible
  - limited "bandwidth" of communication
  - Continuity and Protection

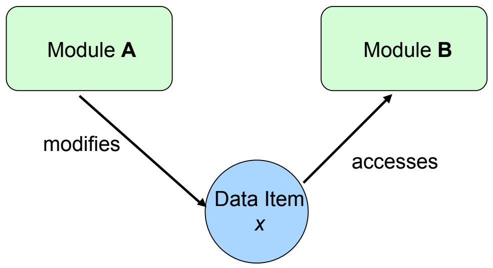


4. Explicit Interfaces

- Whenever two modules A and B communicate, this must be obvious from the text of A or B or both.
  - Decomposability and Composability
  - Continuity, Understandability

4. Explicit Interfaces (2)

- The issue of indirect coupling
  - data sharing



Rule 2 + Rule 3 + Rule 4 Rephrased

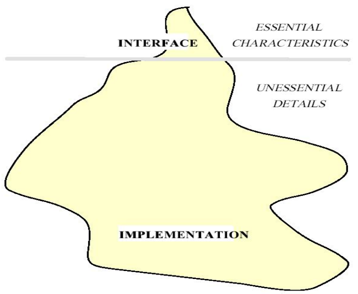
- Few Interfaces: *"Don't talk to many!"*
- Small Interfaces: *"Don't talk a lot!"*
- Explicit Interfaces: *"Talk loud and in public! Don't whisper!"*

5. Information Hiding

Motivation: design decisions that are subject to change should be hidden behind abstract interfaces, i.e. components
- Components should communicate only through well-defined interfaces
- Each component is specified by as little information as possible

- **Continuity:** If internal details change, client components should be minimally affected
  - not even recompiling or linking

Abstraction vs. Information Hiding



Information hiding is one means to enhance abstraction!

Software Architecture

Fowler's Ironic Definition of Architecture...

I define architecture as a word we use
when we want to talk about design
but want to puff it up to make it sound important

M.Fowler – "Who Needs an Architect", 2003

Definition of Software Architecture

"Software Architecture involves the description of

- **elements** from which systems are built,
- **interactions** among those elements,
- **patterns** that guide their composition and
- **constraints** on these patterns"

Shaw, Garlan – "Software Architecture", 1996

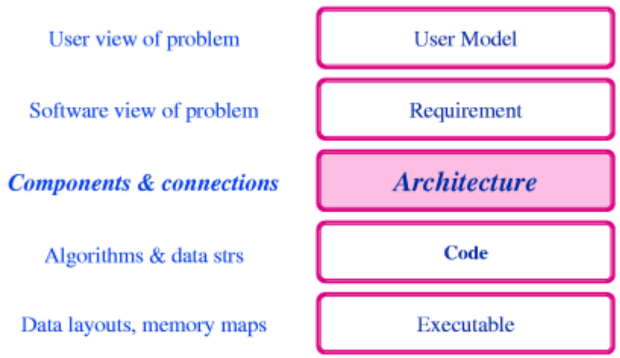
Definition of Software Architecture (2)

"The software architecture of a program or computing system is the **structure** (or structures) **of the system**, which comprise

- **software components**,
- the externally **visible properties** of those components and
- the **relationships** among them"

Bass,Clements, Kazman – "Software Architecture in Practice", 1998

The Place of Software Architecture



More Definitions...

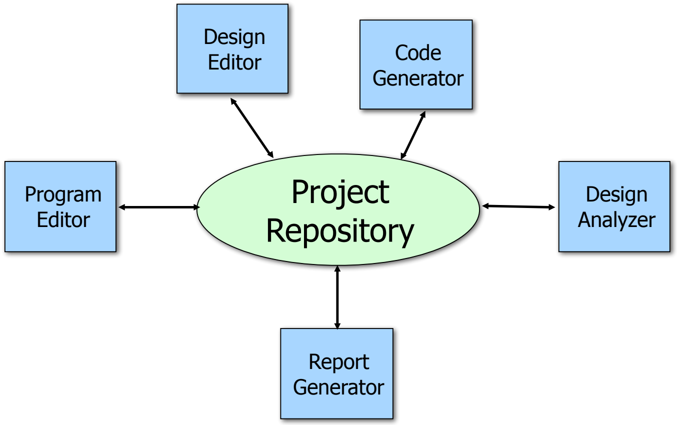
- In most successful software projects, the expert developers working on that project have a shared understanding of the system design. This shared understanding is called 'architecture.' This understanding includes how the system is divided into components and how the components interact through interfaces. These components are usually composed of smaller components, but the architecture only includes the components and interfaces that are understood by all the developers. **[R.Johnson, 2003]**
- Architecture is about the important stuff. Whatever that is. **[R.Johnson, 2003]**
- Architecture is the decisions that you wish you could get right early in a project, but that you are not necessarily more likely to get them right than any other. **[R.Johnson, 2003]**
- ...things that people perceive as hard to change **[M.Fowler, 2003]**

Non-Functional Requirements

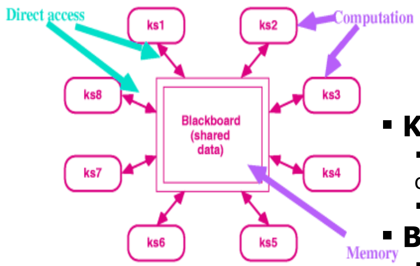
- **Performance**
  - Small number of subsystems ➔ large-grained components
  - Reduce communication
- **Security**
  - Layered structure
  - Most critical layer in the inside
  - High-level of security validation
- **Maintainability**
  - Easy to change ➔ Fine-grained, self-contained components
  - Producers of data separated from consumers
  - Avoid shared data structures

Some Architectural Styles...

Repositories. An Example



Repositories – Blackboard



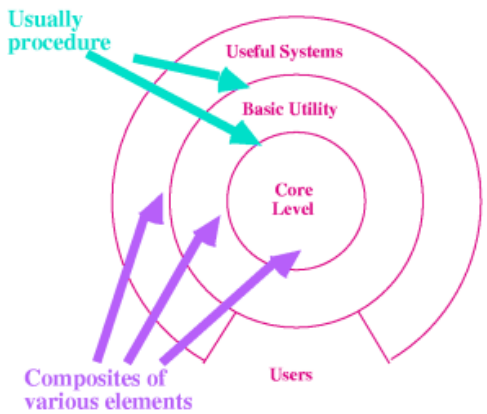
- **Knowledge Sources** (ks)
  - knowledge partitioned in independent computations
  - respond to changes in blackboard
- **Blackboard**
  - entire state of problem solution
  - means of interaction among "ks" to find the solution
- **Control**
  - in model → "ks" self-activated by changes in the blackboard

Repositories – Summary

- ✓ Efficient way to share large amounts of data
- ✓ Data producers and consumers are totally independent
- ✓ Subsystems don't have to care about auxiliary responsibilities
  - e.g. backup, security, recovery from error

- ✖ The common data-model is a compromise among subsystems
- ✖ Expensive translations to a new model
- ✖ Repository forces a centralized policy on all subsystems
  - ✖ e.g. backup, security, access control

Layered Architecture (a.k.a. Abstract Machine)



Layered Architecture – Summary

✓ Changeable and portable architecture

✓ Suitable for incremental development

- Provide services as soon as a layer is implemented

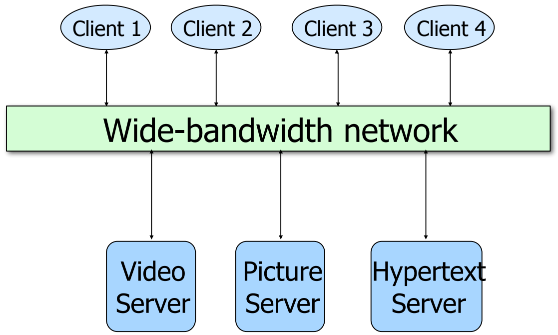
✓ Support reuse

✖ Hard to achieve such a rigorous structuring

- ✖ Hard to find the proper levels of abstraction

✖ Reduces performance by increased communication

Client-Server. An Example



Client-Server – Summary

✓ Architecture is distributed

- ✓ Easy to add and integrate new servers
- ✓ System can be reconfigured dynamically

✓ Servers are not aware of clients

- ✓ Neither identity nor number

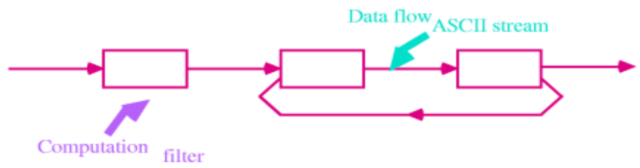
✓ Each server can organize its own data-model

- ✓ better then the centralized data-model in *Repository*

✖ Performance problems if large amounts of data are exchanged

✖ Hard to anticipate problems with integrating data from a new server

Pipes and Filters



- **Filter**
  - incrementally transform some flow of data at inputs to data at outputs
  - share no state with other filters
  - don't depend on the upstream and downstream filters
- **Pipe**
  - Move data from a filter output to a filter input
  - form graphs of data transmission

Pipes and Filters – Summary

- ✓ Understand behavior as a composition of the behavior of individual filters
- ✓ Support for filter reuse
- ✓ Systems are easy to maintain and enhance
  - ✓ By changing or adding new filters

- ✖ Not good for interactive applications
- ✖ Filters need a common format for data transfer
- ✖ Each filter must parse and un-parse the data stream → overhead