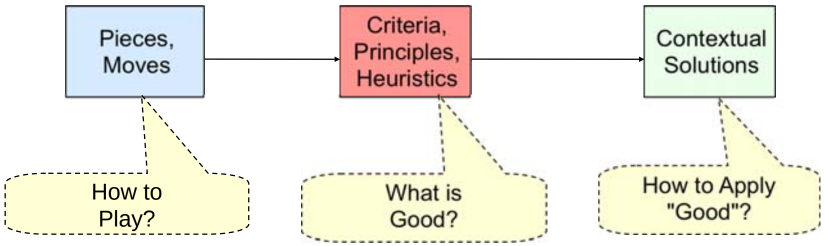
Criteria and Principles of Good Design

From Journeyman to Master



Stages of Learning

- Learn the Rules!
  - algorithms, data structures and languages of software
  - write programs, although not always good ones
- Learn the Principles!
  - software design, programming paradigms with pros and cons
  - importance of cohesion, coupling, information hiding, dependency management
- Learn the Patterns!
  - study the "design of masters"
  - Understand! Memorize! Apply!

Citing Robert Martin ...

"... But to truly master software design, one must study the designs of other masters.

Deep within those designs are patterns that can be used in other designs.

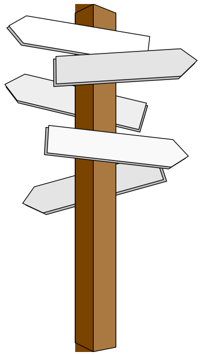
Those patterns must be understood, memorized, and applied ***repeatedly*** *until they become second nature."*

Where Do We Stand ?

- We know the Rules
  - 1-2 OO programming language (Java, C++)
  - some experience in writing programs (< 10 KLOC)
- We heard about Principles
  - "Open-Closed"; "Liskov Substitution Principle" etc.
  - randomly applied some of them
- We dream of becoming "design masters" but...
- …we believe that writing good software is somehow "magic"
  - exclusively tailored for geniuses, "artists", gurus ;-)

A Roadmap

- What is Good Design?
  - Goals of Design
  - Key Concepts and Principles
  - Criteria for Good Design
  - Principles and Rules of Good Design
- What is Good Object-Oriented Design?
  - Guidelines, Rules, Heuristics
- How to Apply Good Design?
  - Design Patterns
  - Architectural Patterns (Styles)



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

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 & Jourdon**

There is no correct design! You must decide!

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

Key Design Issues

Main purpose - **Manage software system complexity** by ...

... improving software quality factors

... facilitate systematic reuse

1. **Decomposition/Composition**
   - Why and How ?
   - What is a component?

2. **Modularity**
   - Definition. Benefits
   - Criteria
   - Principles

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 SW 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

Component Interface

A component interface consists of several elements:

- Exports
  - services provided to other components
- Imports
  - services required from other components
- Access Control
  - e.g. protected/private/public

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**

Benefits of Modularity

Modularity facilitates software quality factors, e.g.:
- Extensibility
  - well-defined, abstract interfaces
- Reusability
  - low-coupling, high-cohesion
- Portability
  - hide machine dependencies

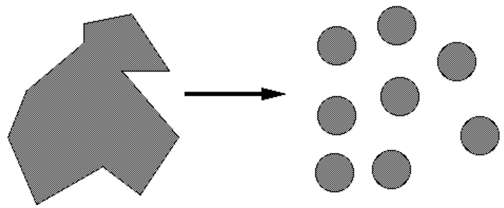
Modularity is important for good design since it
- Enhances for separation of concerns
- Enables developers to reduce overall system complexity via decentralized software architectures
- Increases scalability by supporting independent and concurrent development by multiple personnel

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?

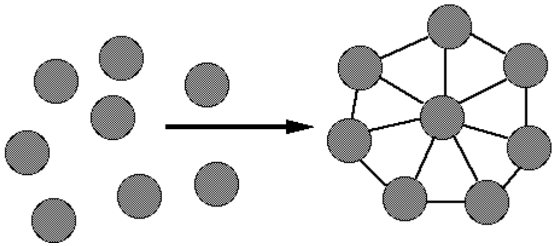
1. Decomposability

- 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
  - Counter-example: use of pre-pocessors



Decomposability and Composability

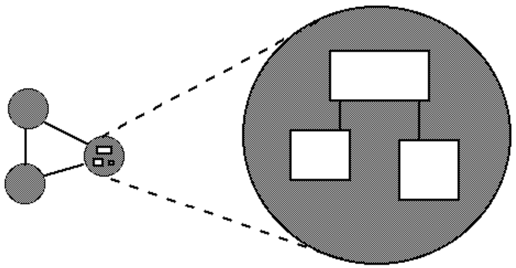
The second [precept I devised for myself] was to divide each of the difficulties which I would examine into as many parcels as it would be possible and required to solve it better.

The third was to drive my thoughts in due order, beginning with these objects most simple and easiest to know, and climbing little by little, so to speak by degrees, up to the knowledge of the most composite ones; and assuming some order even between those which do not naturally precede one another.

Rene Decartes

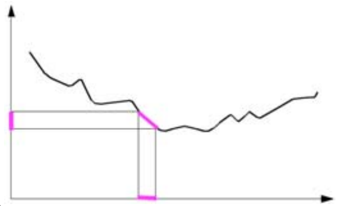
3. Understandability

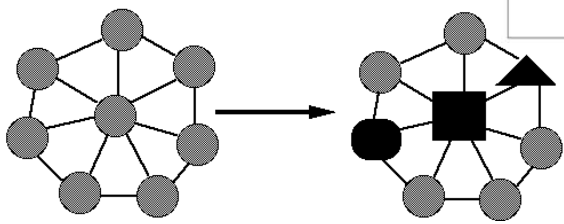
- Individual modules understandable by human reader
  - Counter-example: Sequential Dependencies (A | B | C)
    - contextual significance of modules



4. Continuity

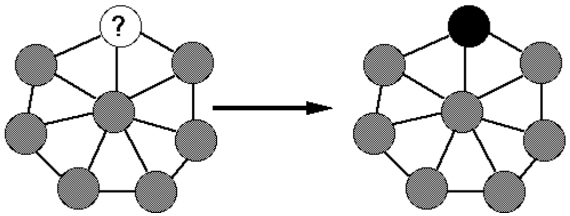
- Small change in requirements results in:
  - changes in only a few modules does not affect the architecture
  - Example: Symbolic Constants
  - Counter-Example: non-uniform access





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

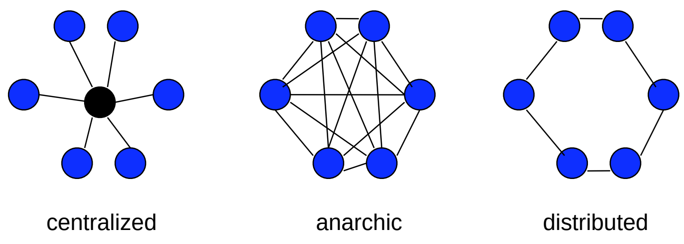
- Keep the structure of the solution compatible with the structure of the modeled problem domain
  - clear mapping (correspondence) between the two

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

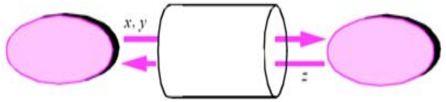
2. Few Interfaces

- 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

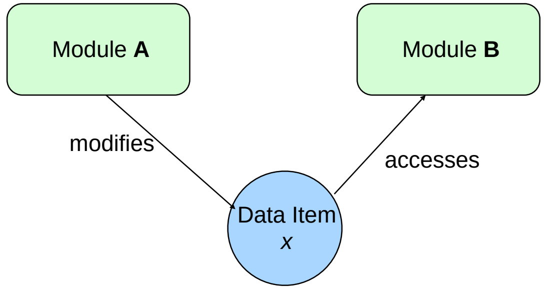


4. Explicit Interfaces

- 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

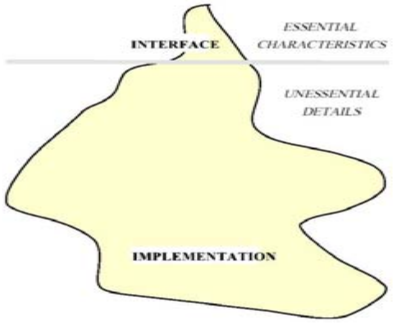
- Few Interfaces: *"Don't talk to many!"*
- Small Interfaces: *"Don't talk a lot!"*
- Explicit Interfaces: *"Talk loud and in public! Don't whisper!"*

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!