Principles of Software Construction: Objects, Design and Concurrency

Design Goals

Christian Kästner  Charlie Garrod
Learning Goals

• Discuss alternative designs in terms of design goals
  ▪ Design for division of labor
  ▪ Design for understandability and maintenance
  ▪ Design for change
  ▪ Design for reuse
  ▪ Design for robustness

• Characterize modularity and its benefits

• Apply design strategies to achieve design goals
  ▪ Explicit interfaces (clear boundaries)
  ▪ Information hiding (hide likely changes)
  ▪ Low coupling (reduce dependencies)
  ▪ High cohesion (one purpose per class)
  ▪ Low repr. gap (align requirements and impl.)

• Understand how strategies support goals

• Explain tradeoffs in designs with design goals and strategies
Goal of Software Design

• For each desired program behavior there are infinitely many programs that have this behavior
  ▪ What are the differences between the variants?
  ▪ Which variant should we choose?

• Since we usually have to synthesize rather than choose the solution...
  ▪ How can we design a variant that has the desired properties?
void sort(int[] list, String order) {
    ... 
    boolean mustswap;
    if (order.equals("up")) {
        mustswap = list[i] < list[j];
    } else if (order.equals("down")) {
        mustswap = list[i] > list[j];
    }
    ... 
}

void sort(int[] list, Comparator cmp) {
    ... 
    boolean mustswap;
    mustswap = cmp.compare(list[i], list[j]);
    ... 
}

interface Comparator {
    boolean compare(int i, int j);
}

class UpComparator implements Comparator {
    boolean compare(int I, int j) { return i<j; }}

class DownComparator implements Comparator {
    boolean compare(int I, int j) { return i>j; }}

Tradeoffs
it depends
(see context)
depends on what?
what are scenarios?
what are tradeoffs?
The design process

1. Object-Oriented Analysis
   - Understand the problem
   - Identify the key concepts and their relationships
   - Build a (visual) vocabulary
   - Create a domain model (aka conceptual model)

2. Object-Oriented Design
   - Identify software classes and their relationships with class diagrams
   - Assign responsibilities (attributes, methods)
   - Explore behavior with interaction diagrams
   - Explore design alternatives
   - Create an object model (aka design model and design class diagram) and interaction models

3. Implementation
   - Map designs to code, implementing classes and methods
Object-Oriented Design

• “After identifying your requirements and creating a domain model, then add methods to the software classes, and define the messaging between the objects to fulfill the requirements.”

• But how?
  ▪ How should concepts be implemented by classes?
  ▪ What method belongs where?
  ▪ How should the objects interact?
  ▪ This is a critical, important, and non-trivial task
Software Quality

- **Sufficiency / Functional Correctness**
  - Fails to implement the specifications ... Satisfies all of the specifications

- **Robustness**
  - Will crash on any anomalous even ... Recovers from all anomalous events

- **Flexibility**
  - Will have to be replaced entirely if specification changes ... Easily adaptable to reasonable changes

- **Reusability**
  - Cannot be used in another application ... Usable in all reasonably related applications without modification

- **Efficiency**
  - Fails to satisfy speed or data storage requirement ... satisfies speed or data storage requirement with reasonable margin

- **Scalability**
  - Cannot be used as the basis of a larger version ... is an outstanding basis...

- **Security**
  - Security not accounted for at all ... No manner of breaching security is known

Source: Braude, Bernstein, Software Engineering. Wiley 2011
Design for Change
Software Change

• ...accept the fact of change as a way of life, rather than an untoward and annoying exception. —Brooks, 1974

• Software that does not change becomes useless over time. —Belady and Lehman

• For successful software projects, most of the cost is spent evolving the system, not in initial development
  ▪ Therefore, reducing the cost of change is one of the most important principles of software design
Design for Division of Labor
Building Complex Systems

- Division of Labor
- Division of Knowledge and Design Effort
- Reuse of Existing Implementations
Design for Change
Design for Division of Labor
Design for Understandability
Design for Reuse
Design for Robustness
Design for Enabling Innovation
Design for Security
...

Modularity
Before System/360

- 7070
  - 7074
- 7090
- 1401
  - 1410
- 1620
- 7080
- 7030

- 8 different processors
- 6 different data formats and instruction sets
- each designed from scratch
- each with their own hardware connections
  - memory
  - I/O
  - disks
Modular System/360

- Family of similar processors
- Standardized communication with I/O devices (except for data rate)
- A single memory-CPU coupling
- A single I/O control system
- Compatible binary representations
  - compiled on one machine, executed on another

=> Interchangeable hardware
=> Separate design of hardware
System/360 in 1970

<table>
<thead>
<tr>
<th>Components</th>
<th>Memory</th>
<th>Control</th>
<th>ALU</th>
<th>Storage</th>
<th>Input/Output</th>
<th>Communications</th>
<th>System Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMS Circuits</td>
<td>Ferrite Core Memory (various sizes)</td>
<td>ROS 25</td>
<td>Model 20</td>
<td>Selector(1)</td>
<td>Multiplexor(1)</td>
<td>Basic Programming Support (BPS)</td>
<td></td>
</tr>
<tr>
<td>SLT Circuits</td>
<td>Thin Film Memory 1968</td>
<td>ROS 30</td>
<td>Model 25</td>
<td>Disk Drive</td>
<td>Telephone Line Adapter</td>
<td>Basic Operating System (BOS)</td>
<td></td>
</tr>
<tr>
<td>Integrated Circuits</td>
<td></td>
<td>ROS 40</td>
<td>Model 30</td>
<td>Removable Disk Drive</td>
<td>High Speed Telephone Line Adapter</td>
<td>Tape Operating System (TOS)</td>
<td></td>
</tr>
<tr>
<td>Cards</td>
<td></td>
<td>Model 40</td>
<td>Model 44</td>
<td>High Speed Multiple Disk Drive</td>
<td>Light pen (input)</td>
<td>Full Duplex System (FTS)</td>
<td></td>
</tr>
<tr>
<td>Boards</td>
<td></td>
<td>Model 65</td>
<td>Model 67</td>
<td>Magnetic Drum</td>
<td>Terminal</td>
<td>Disk Operating System (DOS)</td>
<td></td>
</tr>
<tr>
<td>Automated Assembly</td>
<td></td>
<td>Model 75</td>
<td>Model 85</td>
<td>Magnetic Cards</td>
<td>Card Readers (various)</td>
<td>OS/360 PCP Primary Control</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cache Memory 1968</td>
<td>Model 91</td>
<td>Model 191</td>
<td>Tape Drive</td>
<td>Card Punches (various)</td>
<td>OS/360 MFT Memory Management</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Block Transfer Processor</td>
<td>High Speed Tape Drive</td>
<td>Line Printer</td>
<td>Compilers (various)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Addressable Tape Drive</td>
<td>Optical Scanners (various)</td>
<td>Paper Tape Reader</td>
<td>PL/1 Programming Language</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Vector Array Processor</td>
<td>Bank Check Reader</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[Baldwin & Clark 2000]
## Design Structure Matrix

<table>
<thead>
<tr>
<th>Design Parameter</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Material</td>
<td>*</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Tolerance</td>
<td>X</td>
<td>*</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Mfr. Process</td>
<td>X</td>
<td>X</td>
<td>*</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Height</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Vessel Diameter</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Width of Walls</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Type of Walls</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Weight</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Handle Material</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Handle Shape</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

[Baldwin&Clark 2000]
<table>
<thead>
<tr>
<th>Drive System</th>
<th>Main Board</th>
<th>LCD Screen</th>
<th>Packaging</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>. X X X X</td>
<td>. X X</td>
<td>. X X X X</td>
<td>. X X X</td>
</tr>
<tr>
<td>X X . X X</td>
<td>X X X X X</td>
<td>X X X .</td>
<td>X X X X</td>
</tr>
<tr>
<td>X X X . . X</td>
<td>X X X X X</td>
<td>X X X . X</td>
<td>X X X X</td>
</tr>
<tr>
<td>X X X X X</td>
<td>X X X X X</td>
<td>X X X X X</td>
<td>X X X X</td>
</tr>
</tbody>
</table>

[Baldwin & Clark 2000]
Modularity

- Interdependence within and independence across modules

“A module is a unit whose structural elements are powerfully connected among themselves and relatively weakly connected to elements in other units. Clearly there are degrees of connection, thus there are gradations of modularity.”
Modularity

- Abstraction, Information Hiding, and Interface

- “A complex system can be managed by dividing it up into smaller pieces and looking at each one separately. When the complexity of one of the elements crosses a certain threshold, that complexity can be isolated by defining a separate abstraction that has a simple interface. The abstraction hides the complexity of the element; the interface indicates how the element interacts with the larger system.”
<table>
<thead>
<tr>
<th>Drive System</th>
<th>Main Board</th>
<th>LCD Screen</th>
<th>Packaging</th>
</tr>
</thead>
</table>

- Graphics controller on Main Board or not?
  - If yes, screen specifications change;
  - If no, CPU must process more; adopt different interrupt protocols
Interface (or Design Rule)

- Establishing a stable contract/interface

**Benefit:**
- Efficiency; independent decisions

**Costs:**
- Eliminates a choice
- Interface may not change (evolution)
- Interface may prevent superior design (opportunity costs)
Understanding the Domain (OO Analysis + Design)

- **Poorly Understood**
  - Unforeseen Interdependencies
  - High Risk; Integration and Testing Problems
  - High Opportunity Costs

- **Well Understood**
  - Clear Design Rules
  - Standards
  - Testing and Integration can be done by Users
Consequences

- Modularity increases the range of “manageable” complexity
- Modularity allows concurrent (design) work
- Modularity accommodates uncertainty
  - Isolate risks
  - Isolate parts that will likely change
- Modularity allows decentralized improvements
- Modularity as an investment (~stock option)
Design Goals and Design Strategies
Overview

• 5 design goals
  ▪ Design for division of labor
  ▪ Design for understandability
  ▪ Design for change
  ▪ Design for reuse
  ▪ Design for robustness

• 5 design strategies (for now)
  ▪ Explicit interfaces (clear boundaries)
  ▪ Information hiding (hide likely changes)
  ▪ Low coupling (reduce dependencies)
  ▪ High cohesion (one purpose per class)
  ▪ Low repr. gap (align requirements and impl.)
• Modular Decomposition
  - decomposing both design and implementation!

• Limit interactions in design process

```
«interface» Item
+getImage()
+getName()

«interface» World
+draw()
GUI

«interface» Item
+getImage()
+getName()

«interface» World
+draw()
GUI

draw:
  if (item instanceof Rabbit)
    img = rabbitImg
  if (item instanceof Fox)
    img = foxImg
  ...
```
In the first design GUI and Item are independent
- New items can be added without changing the GUI
- A different GUI can be implemented without changing items

Items have an interface describing how they can be drawn
- Only this data available to GUI

In the second design the GUI needs to know the possible items
- GUI designer and Item designer need to communicate
Design Strategy: Explicit Interfaces

- Whenever two modules communicate, it should be obvious from their interfaces
- Make interfaces between modules explicit
  - Public methods
  - Documentation, contracts
  - Import of classes and libraries
- Avoid global state (e.g., static fields)
- Avoid hidden interactions through side channels (data, files, network, ...)
- Keep interfaces small and exchange as little information as possible
Design for Reuse

- Modular Composability, compose modules from different sources

- Good modules contain well-defined tasks reusable in many contexts
  - (Module may be a class, a package, a subsystem, ...)
  - **Explicit interfaces** how others can use a module
  - Limited dependencies on other modules (**low coupling**)
  - Useful **cohesive** functionality that’s worth reusing

- Examples
  - GUI reuses Swing GUI elements
  - World reuses collection libraries
  - JUnit reusable in many projects
  - Rabbits potentially reusable in different virtual world
  - GUI reusable for arbitrary other simulations with items
  - HW 1 reuses stack implementation from HW 0
Design Strategies: Coupling and Cohesion

Coupling

- Modules should depend on as few other modules as possible
  - Easier to understand (little context to understand)
  - Independent modules easier to modify without rippling effects (Design for Change)
  - Easier to reuse in different context

Cohesion

- All responsibilities of a module should be related and well defined – one purpose per class
  - A module with lots of unrelated functionality is unlikely to be reusable as it (or unnecessarily large)
  - A cohesive module is easier to understand

Chose design alternatives with lower coupling and higher cohesion
Coupling and Cohesion

**Coupling**

- Modules should depend on as few other modules as possible
  - Easier to understand (little context to understand)
  - Independent modules easier to modify without rippling effects (Design for Change)
  - Easier to reuse in different context

**Cohesion**

- All responsibilities of a module should be related and well defined – one purpose per class
  - A module with lots of unrelated functionality is unlikely to be reusable as it (or unnecessarily large)
  - A cohesive module is easier to understand

Chose design alternatives with lower coupling and higher cohesion
Example

- Create a Payment and associate it with the Sale.
Example:

\[
\text{makePayment() \rightarrow : Register \rightarrow 1: create() \rightarrow p : Payment} \\
\text{2: addPayment(p) \rightarrow : Sale}
\]
Example

\begin{verbatim}
makePayment()

: Register
1: create()

: Payment

2: addPayment(p)

: Sale

\end{verbatim}
Second solution has less coupling
Register does not know about Payment class
Topologies with different coupling

(A)  (B)  (C)
Why High Coupling is undesirable

- **Element with low coupling** depends on only few other elements (classes, subsystems, ...)
  - “few” is context-dependent

- **A class with high coupling** relies on many other classes
  - Changes in related classes force local changes; changes in local class forces changes in related classes (brittle, rippling effects)
  - Harder to understand in isolation.
  - Harder to reuse because requires additional presence of other dependent classes
  - Difficult to extend – changes in many places
Common Forms of Coupling in OO Languages

• TypeX has an attribute (data member or instance variable) that refers to a TypeY instance, or TypeY itself.

• TypeX has a method which references an instance of TypeY, or TypeY itself, by any means.
  - Typically include a parameter or local variable of type TypeY, or the object returned from a message being an instance of TypeY.

• TypeX is a direct or indirect subclass of TypeY.

• TypeY is an interface, and TypeX implements that interface.
Low Coupling: Discussion

• Low Coupling is a principle to keep in mind during all design decisions

• It is an underlying goal to continually consider.

• It is an evaluative principle that a designer applies while evaluating all design decisions.

• Low Coupling supports design of more independent classes; reduces the impact of change.

• Context-dependent; should be considered together with cohesion and other principles and patterns

• Prefer coupling to interfaces over coupling to implementations
Low Coupling: Discussion

• Subclassing produces a particularly problematic form of high coupling
  ▪ Dependence on implementation details of superclass
  ▪ -> Prefer composition over inheritance

• Extremely low coupling may lead to a poor design
  ▪ Few incohesive, bloated classes do all the work; all other classes are just data containers

• High coupling to very stable elements is usually not problematic
Coupling to “non-standards”

- Libraries or platforms may include non-standard features or extensions
- Example: JavaScript support across Browsers

```html
<div id="e1">old content</div>
```

In JavaScript...

**MSIE:** `e1.innerText = "new content"`

**Firefox:** `e1.textContent = "new content"`

_W3C-compliant DOM standard_
Coupling and Cohesion

Coupling

• Modules should depend on as few other modules as possible
  ▪ Easier to understand (little context to understand)
  ▪ Independent modules easier to modify without rippling effects (Design for Change)
  ▪ Easier to reuse in different context

Cohesion

• All responsibilities of a module should be related and well defined – one purpose per class
  ▪ A module with lots of unrelated functionality is unlikely to be reusable as it (or unnecessarily large)
  ▪ A cohesive module is easier to understand

Choose design alternatives with lower coupling and higher cohesion
• Classes are easier to maintain
• Easier to understand
• Often support low coupling
• Supports reuse because of fine grained responsibility
(except for coupling), looks OK if `makePayment` considered in isolation, but adding more system operations, `Register` would take on more and more responsibilities and become less cohesive.
Example

```
makePayment()  
Register

1: create()  
p: Payment

2: addPayment(p)  
Sale

makePayment()  
Register

1: makePayment()  
Sale

1.1. create()  
Payment
```

Example:
```
makePayment()  
Register

1: create()  
p: Payment

2: addPayment(p)  
Sale

makePayment()  
Register

1: makePayment()  
Sale

1.1. create()  
Payment
```
class Graph {
    Node[] nodes;
    boolean[] isVisited;
}
class Algorithm {
    int shortestPath(Graph g, Node n, Node m) {
        for (int i; ...)
            if (!g.isVisited[i]) {
                ...
                g.isVisited[i] = true;
            }
        return v;
    }
}
Cohesion: Discussion

- **Very Low Cohesion:** A Class is solely responsible for many things in very different functional areas.

- **Low Cohesion:** A class has sole responsibility for a complex task in one functional area.

- **High Cohesion:** A class has moderate responsibilities in one functional area and collaborates with classes to fulfil tasks.

- **Advantages of high cohesion**
  - Classes are easier to maintain
  - Easier to understand
  - Often support low coupling
  - Supports reuse because of fine grained responsibility

- **Rule of thumb:** a class with high cohesion has relatively few methods of highly related functionality; does not do too much work.
Monopoly Example

class Player {
    Board board;
    Square getSquare(String name) {
        for (Square s: board.getSquares())
            if (s.getName().equals(name))
                return s;
    return null;
}
}

class Player {
    Board board;
    Square getSquare(String n) { board.getSquare(n); }
}

class Board{
    List<Square> squares;
    Square getSquare(String name) {
        for (Square s: squares)
            if (s.getName().equals(name))
                return s;
    return null;
}
Aside: Law of Demeter (LoD)

- LoD (or Principle of Least Knowledge): Each module should have only limited knowledge about other units: only units "closely" related to the current unit
- In particular: Don’t talk to strangers!
- For instance, no a.getB().getC().foo()
Design for Understandability

- Understand (maintain, debug, test) modules locally, ideally in isolation

- Good modules are self-contained and understandable with little context
  - **Explicit interfaces**, well-defined and well-documented
  - **Low coupling, high cohesion**
  - Self-documenting implementations

- Examples:
  - GUI can be understood without knowing how Rabbits load their icons
  - Graphs can be understood without knowing how they are used; graph algorithms can be understood without knowing how graphs are implemented
Design for Change

- Anticipate change where possible
  - Identify risks
  - Identify instable requirements
  - Identify opportunities for future innovation
  - Identify stable parts
  - Anticipate future trends, customers, ...

- Design modules such that:
  - Interfaces correspond to stable parts
  - Internal implementations hide unstable parts
  - Allows changing module implementation locally without affecting remaining system

- Example:
  - Initialization of virtual world encapsulated in single class
  - Common behavior of rabbits, foxes codified in interfaces, AI implementation hidden for change
Design for Change: Continuity

• Continuity: Small changes in the requirements should require only small changes in the implementation

• Design for change
  ▪ **Low coupling** and **high cohesion** prevent ripple effects
  ▪ Anticipate likely changes, extensions, and risk; hide behind an interface (**information hiding**, polymorphism)

• Heuristic **Low Representational Gap**: Align object model with domain model

• Examples:
  ▪ New language editor in Eclipse should not require modification of Eclipse’s platform
  ▪ Changing what Travis CI executes: change travis.yml file
  ▪ Separate GUI from core implementation
  ▪ Avoid replicating code (one change instead of many)
Continuity (details)

- Poor continuity
  - Major requirements change = minor change
  - Minor requirements change = months of work

- Good continuity
  - Major requirements change = major overhaul
  - Minor requirements change = change a config file

- Some crosscutting usually not avoidable

- Client is less likely to change purpose of the application
  - ...but more likely to change how a dialog appears

- Design with an expectation of change
  - ...but make it less costly to make changes

- Clients will pay for what they see
  - ...but no one will pay $1000 to make text bold
OO Design Strategy: Low Representational Gap

• Align object model with domain model
  ▪ Map problem-space abstractions to solution-space abstractions
  ▪ Model solution-space relationships after problem-space relationships
  ▪ May even start with one class per concept
  ▪ Name classes corresponding to real-world concepts

• Supports design for change: if problem structure and solution structure similar, problem changes should correspond to solution changes

• Supports design for division of labor: knowing the decomposition of the problem, may help decomposing the solution

• Supports design for understandability and reuse: ...
Low Representational Gap

Noteworthy concepts in the domain.

A Payment in the Domain Model is a concept, but a Payment in the Object Model is a software class. They are not the same thing, but the former inspired the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.

The object-oriented developer has taken inspiration from the real world domain in creating software classes.

Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.
Design Strategy: Information Hiding

- A module exposes the interface, but hides how it implements it
  - Interface must remain stable (or else rippling changes)
  - Implementation may change freely

- Each interface represents a design decision and modules hide remaining ones
  - Remember System/360

- Identify stable and unstable parts of the problem
  - E.g., if database will change, abstract behind database interface

- If change predicted correctly: Only one module to change
Design for Robustness

• Modular Protection: Errors and bugs unavoidable, but exceptions should not leak across modules

• Good modules handle exceptional conditions locally
  - Local input validation and local exception handling where possible
  - **Explicit interfaces** with clear pre/post conditions
  - Explicitly documented and checked exceptions where exceptional conditions may propagate between modules
  - **Information hiding**/encapsulation of critical code (likely bugs, likely exceptions)

• Example
  - Printer crash should not corrupt entire system
  - Exception/infinite loop in rabbit AI should not freeze GUI
  - Exception in shortest-path algorithm should not corrupt graph
Summary

• 5 design goals
  ▪ Design for division of labor
  ▪ Design for understandability
  ▪ Design for change
  ▪ Design for reuse
  ▪ Design for robustness

• 5 design strategies (so far)
  ▪ Explicit interfaces (clear boundaries)
  ▪ Information hiding (hide likely changes)
  ▪ Low coupling (reduce dependencies)
  ▪ High cohesion (one purpose per class)
  ▪ Low repr. gap (align requirements and impl.)