Object Oriented Design Principles

Lubomír Bulej
KDSS MFF UK
Software design

Finding solutions to complex problems

- Many sources of complexity
  - Functionality, cost constraints, performance, security, backwards compatibility, reliability, scalability, ...

Software design is difficult!

- Design concerns structure and trade-offs.
  - Design is not code, but art and experience.
- Design has to fit the problem.
  - Often we need to solve the problem to understand it.
- Design has to fit the requirements.
  - Requirements tend to change.
  - We often don’t know what they really are.
How hard can it be?

- Top Gear
What could possibly go wrong?

We could end up with poor design!

- **Rigidity**
  - System is difficult to change. Changes force further changes to other parts of the system.

- **Fragility**
  - Changes tend to break the system in places that have no conceptual relationship to the changed part.

- **Immobility**
  - Difficult to disentangle reusable components.

- **Viscosity**
  - Changes that preserve design are harder than hacks.
What could possibly go wrong?

We could end up with poor design!

- **Needless complexity**
  - System contains infrastructure that adds no direct benefit.

- **Needless repetition**
  - System contains repeated structures.

- **Opacity**
  - Code is difficult to read and understand.
How to arrive at good design?

What is good software design?

- Flexible, reusable, maintainable...
- Naturally fits the requirements.
- Robust in the face of changes.

How to arrive at good design?

- Start with a simple solutions.
- Stimulate changes to force design changes.
- Apply OO design principles when necessary.
What... principles?

SOLID object oriented design principles

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

... and a few more general principles

- Abstraction
- Encapsulation
- High cohesion
- Low coupling
Single Responsibility Principle

There should never be more than one reason for a class to change.
Single Responsibility Principle

Each responsibility is an *axis of change*

- Changes in requirements manifest through changes in responsibility among classes
  - Class with more responsibilities will have more than one reason (i.e., along more axes) to change

Violation: more responsibilities in a single class

- Responsibilities become coupled
  - Changes to one responsibility may impair the ability of a class to meet other responsibilities
- Leads to fragile design that can break in unexpected ways when changed
Example: SRP violation

Rectangle has 2 responsibilities

- Mathematical model
- Graphical rendering
Solution: separated responsibilities

Responsibilities in two different classes

- Changes related to rendering of rectangles cannot affect the CG applications
Example: possible SRP violation

Interface looks reasonable, but...

... **Modem has two responsibilities!**

- Connection management
- Data communication

Need separating? Almost certainly.

- Two sets of functions with nothing in common
- Used by different parts of an application
- Different reasons for change

```
<<Interface>>
Modem
+ dial (phoneNo : String)
+ hangup ()
+ void send (c : char)
+ recv () : char
```

Modem Implementation
Solution: separated responsibilities

- If there really are 2 different axes of change
- Needless complexity if responsibilities never change independently

Apply when there are symptoms of bad design!
Software entities should be open for extension, but closed for modification.
Open Closed Principle

Open for extension
  ● Write code so that entities can be extended with new behavior to satisfy changing requirements.
    ○ We rarely understand the system at first...

Closed for modification
  ● Extending an entity must not result in changes to the code of the entity.
    ○ Need to get (and keep) the system up and running...

Avoid modifying code that already works!
  ○ Strive to implement changes by adding code
Satisfying the OCP

Reduces rigidity
  ○ Prevents cascading changes in dependent entities due to change.

Seemingly contradictory
  ● How can we change entity behavior without changing its source code?

Avoid depending on concrete classes
  ● Abstraction
  ● Polymorphism (both static and dynamic)
Example: OCP violation

The client uses the server directly

- Client must be modified to use a different server
- Changes in server force changes in client
Solution: encapsulate what changes

Strategy pattern (more on that later)

- Server can be changed without affecting the client
Strategic closure in OCP

Strict adherence to OCP is costly
- Abstraction can incur needless complexity!

No program can be 100% closed
- All changes cannot be anticipated.
- Closure needs to be strategic!
  - Choose the kind of changes to close against.

Which changes to close against?
- Focus on the most obvious/likely changes.
- Apply OCP when needed for the first time.

Strategy: stimulate early changes
- Fast iterations, constant feedback on design
Example: OCP violation

The `drawShapes()` method violates OCP

- Not closed against new shape types
Solution: abstraction & polymorphism

Closed against adding new shapes...

- No need to modify the `drawShapes()` method

What if circles need to be drawn before rectangles?
Solution: abstraction

Shapes can be ordered before drawing

- But **all** `precedes()` methods violate OCP
  - Not closed against new Shape implementations
Solution: data driven approach

- The table in the ShapeTypeOrder comparator is not closed against new Shape implementations
  - But provides closure against different ordering policies

Shape derivatives closed against new derivatives

```java
private class <?> [] order = new Class <?> [] {
    Circle.class, Rectangle.class
};

int compare (a, b: Shape) {
    // find index of a.getClass()
    // find index of b.getClass()
    // compare indices
}
```
Subtypes must be substitutable for their base types.
Liskov Substitution Principle

Functions should not know about derived classes
● User of a base class should continue to function properly if a derived class is passed to it.
  ○ Both existing and future derived classes.

Related to Design by Contract
● The (implied) contract of the base class must be honored by all derived classes.
● Expect no more and provide no less
  ○ Derived method pre-conditions are no stronger than the base class method.
  ○ Derived method post-conditions are no weaker.
Example: LSP violation

Classic Circle/Ellipse (Square/Rectangle) dilemma
- Math: circle is a degenerate form of ellipse
- Math: square is degenerate form of rectangle
Example: LSP violation

Minor issue: inheriting unnecessary attributes
- Nothing we could fix with a bit of creative overriding...
Example: LSP violation

Major issue: mutation methods
- Opportunity for clients to break things...
Example: LSP violation

Clients indeed do "ruin" everything...

```java
void useEllipse (Ellipse e) {
    Point a = new Point (-1, 0);
    Point b = new Point (1, 0);
    e.setFoci (a, b);
    e.setMajorAxis (3.0);
    assert a.equals (e.getFocusA ());
    assert b.equals (e.getFocusB ());
    assert e.getMajorAxis == 3.0;
}
```

Clients expect to be able to set foci and major axis...

... and verify that they have been set.

But the function fails if we pass Circle to it...

The Circle subclass violates the Ellipse contract!

- Weakens (implicit) postcondition in `setFoci()` ensuring that arguments get copied to member variables.
LSP and OCP are related

OCP enabled by abstraction and polymorphism
- Key mechanisms: subtyping and inheritance

LSP restricts the use of inheritance
- Provide designs that conform to OCP

LSP violations are latent violations of OCP!
- Syntactically correct violations of semantics.
- Difficult to detect until it is too late.
  - Redesign can be difficult/impossible.
  - May be “fixed” by using type checks to ensure that a method operates on a compatible subtype.
  - But: the use of type checks violates OCP!
Clients should not be forced to depend on interfaces they do not use.
Interface Segregation Principle

Many client-specific interfaces are better than one general purpose interface.

● Avoid “fat” and non-cohesive interfaces.
● Create client-specific interfaces.

Related to Single Responsibility Principle

● Decoupling responsibilities.
Interface Segregation Principle

Avoid fat (general purpose) interfaces
- Contains methods needed by all clients.
- Can introduce coupling between clients.
  - Recall: clients own interfaces!
  - When one client forces interface change, others will be affected as well.

Create client-specific interfaces
- Targeted to a particular type of clients.
- Clients depend only on methods they need/use.
- Probability of change is reduced.
- Impact of changes to one interface is smaller.
- No interface pollution.
Example: interface pollution

Security door

- Can be locked and unlocked, and knows whether it is open or closed.

We need a TimedDoor

- Raises an alarm if it remains open for too long.
- Uses a Timer object to implement timeout.

<table>
<thead>
<tr>
<th>Timer</th>
<th>&lt;&lt;Interface&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ register (timeout: int, handler: TimeoutHandler)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;&lt;Interface&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Door</td>
</tr>
<tr>
<td>+ lock()</td>
</tr>
<tr>
<td>+ unlock()</td>
</tr>
<tr>
<td>+ isOpen() : bool</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TimeoutHandler</th>
<th>&lt;&lt;Interface&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ callback()</td>
<td></td>
</tr>
</tbody>
</table>

How can TimeoutHandler notify TimedDoor?
Example: interface pollution

Common solution
● Force Door to inherit from TimeoutHandler

Consequences
● Door interface polluted with method from TimeoutHandler
● Most Door implementations will provide a degenerated callback() method: violates LSP
● Changes in TimeoutHandler affect all Door implementations
**Example: avoiding interface pollution**

- **TimedDoor used through two separate interfaces**
  - Correspond to two roles assumed by its clients.
  - Changes in `TimeoutHandler` only affect clients that actually use it.
Role-based interface design

Interfaces designed from the service user's perspective
- Not from the service provider's perspective
- Recall: clients own interfaces

Interfaces should represent client roles
- Clients assume roles when using a service.
- Classes implement many interfaces, interfaces implemented by many classes.

Client/role version control
- New interfaces for clients requiring new services
- Easier to preserve design (less viscosity)
- Old clients keep working
Dependency Inversion Principle

Depend upon abstractions.
Dependency Inversion Principle

High-level modules should not depend on low-level ones
- Both should depend on abstractions.

Abstractions should not depend on details
- Details should depend on abstractions.

Rationale
- Concrete (low-level) things change a lot.
- Abstract (high-level) things change less frequently.
- Abstractions are “hinge” points.
  - Places where design can bend or be extended without being modified (OCP).

Overall purpose
- Prevent dependency on volatile modules.
Why dependency *inversion*?

**Traditional (structural, procedural) design**

- High-level modules depend upon low-level modules.
- Abstractions depend upon details.

Changes in low-level modules can *force* high-level modules to change.... Preposterous!

- Modules containing business model or policy.
- Exactly the modules we want to be able to reuse in different contexts!
Why dependency inversion?

Good object-oriented architecture design

- Inverted dependency structure and interface ownership.
- Dependencies (mostly) point to abstractions.

Clients own abstract interfaces, servers derive from them.

Policy layer not affected by changes to Mechanism and Utility layers.

Policy can be reused in contexts where lower-level modules conform to PolicyService interface.
Depending upon abstractions

Guiding principle

● Every dependency in a design should target an interface or an abstract class.
  ○ No variable should refer to concrete class.
  ○ No class should derive from concrete class.
  ○ No method should override an implemented method in any of its base classes.

Mitigating forces

● Concrete class that is very unlikely to change.
  ○ Avoids needless complexity, especially if a concrete class is non-volatile (e.g., java.lang.String).
● Modules creating class instances depend on them.

*Encapsulate what changes!*
Example: naive model

- Button depends on Lamp
  - Affected by changes of Lamp
  - Cannot be used to control other objects.

DIP violation
- Abstraction not isolated from implementation.
Example: dependency inversion

Abstraction isolated from implementation

- Switch can control any Switchable device.
- Any kind of Switch can be used, not only a Button.
Relation to other principles

- OCP states the goal of OO architecture.
- LSP enables OCP and restricts inheritance.
- DIP provides the primary design mechanism.
Based on the *Engineering Notebook* columns by *Robert C. Martin* published in *The C++ Report*

- The Open-Closed Principle, 1996
- The Liskov Substitution Principle, 1996
- The Dependency Inversion Principle, 1996
- The Interface Segregation Principle, 1996

And other articles/book chapters by *Robert C. Martin*

- Design Principles and Design Patterns
- The Single Responsibility Principle