Principles of Object-Oriented Design
Part 1

Oct. 6 2003

Quote of Today

**Good Design:** A system should be built with a minimum set of unchangeable parts; those parts should be as general as possible; and all parts of the system should be held in a uniform framework.

taken from “Design Principles behind Smalltalk”

**Daniel H. H. Ingalls**
Learning Research Group
Xerox Palo Alto Research Center
Common Design Principles

- Abstraction
- Interfaces
- Encapsulation
  - Using a well-defined interface to “protect” the implementation
  - Factoring out common structures Parameterizing the operation details

OO Design Principles

- The Open/Closed Principle (OCP)
- The Liskov Substitution Principle (LSP)
- The Dependency Inversion Principle (DIP)
- The Interface Segregation Principle (ISP)
- The Reuse/Release Equivalency Principle (REP)
- The Common Closure Principle (CCP)
- The Common Reuse Principle (CRP)
- The Acyclic Dependencies Principle (ADP)
- The Stable Abstractions Principle (SAP)
Open-Closed Principle (OCP)

- "Software Systems change during their life time"
  - both better designs and poor designs have to face the changes;
  - good designs are stable

  \[\text{Software entities should be open for extension,} \]
  \[\text{but closed for modification}\]
  \[\text{B. Meyer, 1988 / quoted by R. Martin, 1996}\]

- Be open for extension
  - module’s behavior can be extended
- Be closed for modification
  - source code for the module must not be changes
- Modules should be written so they can be extended without requiring them to be modified

Open the door ...

- How to make the Car run efficiently with a TurboEngine?
- Only by changing the Car!
  - ...in the given design
... But Keep It Closed!

- A class must not depend on a concrete class!
- It must depend on an abstract class ...
- ...using polymorphic dependencies (calls)

---

The open/ closed principle (OCP) Example (bad)

```java
public void drawShape(ShapeData shapeData)
{
  switch (shapeData.shapeType)
  {
    case SQUARE:
      drawSquare((SquareData)shapeData);
      break;
    case CIRCLE:
      drawCircle((CircleData)shapeData);
      break;
  }
}
```
The open/closed principle (OCP) Example (good)

```java
public void drawShape(ShapeInterface shape) {
    shape.draw();
}
```

The open/closed principle (OCP) Discussion

- If I need to create a new shape, such as a Triangle, I must modify the `drawShape()` function.
- In a complex application the switch/case statement above is repeated over and over again for every kind of operation that can be performed on a shape.
- Worse, every module that contains such a switch/case statement retains a dependency upon every possible shape that can be drawn, thus, whenever one of the shapes is modified in any way, the modules all need recompilation, and possibly modification.
- However, when the majority of modules in an application conform to the open/closed principle, then new features can be added to the application by adding new code rather than by changing working code. Thus, the working code is not exposed to breakage.
Strategic Closure

"No significant program can be 100% closed"


- Closure not complete but strategic

- Use abstraction to gain explicit closure
  - provide class methods which can be dynamically invoked
    - to determine general policy decisions
    - e.g. draw Squares before Circles
  - design using abstract ancestor classes

- Use "Data-Driven" approach to achieve closure
  - place volatile policy decisions in a separate location
    - e.g. a file or a separate object
  - minimizes future change locations

OCP Heuristics

Make all object-data private
No Global Variables!

- Changes to public data are always at risk to “open” the module
  - They may have a rippling effect requiring changes at many unexpected locations;
  - Errors can be difficult to completely find and fix. Fixes may cause errors elsewhere.

- Non-private members are modifiable
  - Case 1: "I swear it will not change"
    - may change the status of the class
  - Case 2: the Time class
    - may result in inconsistent times
OCP Heuristics (2)

**RTTI is Ugly and Dangerous!**

- RTTI is ugly and dangerous
  - If a module tries to dynamically cast a base class pointer to several derived classes, anytime you extend the inheritance hierarchy, you need to change the module
  - Recognize them by type `switch`-es or `if-else-if` structures
- Not all these situations violate OCP all the time
  - When used only as a “filter”

---

One More Example

- Design search filters for searching a collection of books (called “books” below)
  
  1. By the book’s publication year

```java
public class Filter {
    public Vector filterOnYearOfPublication(Vector books, int year) {
        Vector newBooks = new Vector();
        Enumeration enum = books.elements(); // the collection
        while(enum.hasMoreElements()){
            Book book = (Book)enum.nextObject();
            if(book.getYearOfPublication() == year) {
                newBooks.addElement(book);
            }
        }
        return newBooks;
    }
}
```
One More Example (Cont’d)

2. By the book’s language

```java
public class Filter {
    public Vector filterOnYearOfPublication(...) {
        ...
    }

    public Vector filterOnLanguage(Vector books, String language) {
        Vector newBooks = new Vector();
        Enumeration enum = books.elements();
        while (enum.hasMoreElements()) {
            Book book = (Book) enum.nextObject();
            if (book.getLanguage().equals(language)) {
                newBooks.addElement(book);
            }
        }
        return newBooks;
    }
}
```

Redesign the Example Using OCP

**Sketch:**

```java
public interface Selector {
    boolean shouldSelect(Book book);
}

public class Filter {
    ...
    public Vector filter(Vector books, Selector selector) {
        Vector newBooks = new Vector();
        Enumeration enum = books.elements();
        while (enum.hasMoreElements()) {
            Book book = (Book) enum.nextObject();
            if (selector.shouldSelect(book)) {
                newBooks.addElement(book);
            }
        }
        return newBooks;
    }
}
```
public class YearOfPublicationSelector implements Selector {
    private int year;
    public YearOfPublicationSelector(int year) {
        super(); this.year = year;
    }
    public boolean shouldSelect(Book book) {
        return book.getYearOfPublication() == year;
    }
}

public class LanguageSelector implements Selector {
    private String language;
    public LanguageSelector(String lang) {
        super(); language = lang;
    }
    public boolean shouldSelect(Book book) {
        return book.getLanguage().equals(language);
    }
}

Liskov Substitution Principle (LSP)

- The key of OCP: Abstraction and Polymorphism
  - Implemented by inheritance
  - How do we measure the quality of inheritance?

Inheritance should ensure that any property proved about supertype objects also holds for subtype objects
B. Liskov, 1987

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.
R. Martin, 1996
Inheritance *Appears* Simple

```cpp
class Bird { // has beak, wings,...
    public: virtual void fly(); // Bird can fly
};

class Parrot : public Bird { // Parrot is a bird
    public: virtual void mimic(); // Can Repeat words...
};

// ...
Parrot mypet;
mypet.mimic(); // my pet being a parrot can Mimic()
mypet.fly(); // my pet “is-a” bird, can fly
```

---

Penguins Fail to Fly!

```cpp
class Penguin : public Bird {
    public: void fly() {
        error ("Penguins don’t fly!");
    }
};

void PlayWithBird (Bird& abird) {
    abird.fly(); // OK if Parrot.
    // if bird happens to be Penguin...OOOPS!!
}
```

- Does not model: “*Penguins can’t fly*”
- It models “*Penguins may fly, but if they try it is error*”
- Run-time error if attempt to fly → not desirable
- **Think about Substitutability - Fails LSP**
Design by Contract

- Advertised Behavior of an object:
  - advertised Requirements (Preconditions)
  - advertised Promises (Postconditions)

When redefining a method in a derivate class, you may only replace its precondition by a weaker one, and its postcondition by a stronger one

B. Meyer, 1988

⇒ Derived class services should require no more and promise no less

```cpp
text
int Base::f(int x);
// REQUIRE: x is odd
// PROMISE: return even int

int Derived::f(int x);
// REQUIRE: x is int
// PROMISE: return 8
```

Example: Design By Contracts In Eiffel

defered class PLANE
inherit AIRCRAFT
feature
  start_take_off is
    require
    deferred
    ensure
    end
start_landing, increase_altitude, decrease_altitude, moving, altitude, speed, time_since_take_off

... [Other features] ...

invariant
  (time_since_take_off <= 20) implies (assigned_runway.owner = Current)
moving = (speed > 10)
end

Precondition

-- i.e. specified only.
-- not implemented.

Postcondition

Class invariant
Square IS-A Rectangle?

- Should I inherit Square from Rectangle?

The Answer is ...

- Override `setHeight` and `setWidth`
  - duplicated code...
  - static binding (in C++)
    - `void f(Rectangle& r) { r.setHeight(5); }`
    - change base class to set methods `virtual`

- The real problem
  ```cpp
  void g(Rectangle& r) {
    r.setWidth(5); r.setHeight(4);
    // How large is the area?
  }
  ```
  - 20! ... Are you sure? ;)

- IS-A relationship must refer to the behavior of the class!
LSP is about Semantics and Replacement

- The meaning and purpose of every method and class must be clearly documented
  - Lack of user understanding will induce de facto violations of LSP

- Replaceability is crucial
  - Whenever any class is referenced by any code in any system, any future or existing subclasses of that class must be 100% replaceable
  - Because, sooner or later, someone will substitute a subclass; it’s almost inevitable.

LSP and Replaceability

- Any code which can legally call another class’s methods must be able to substitute any subclass of that class without modification:
LSP Related Heuristic (2)

It is illegal for a derived class, to override a base-class method with a NOP method

- NOP = a method that does nothing
- **Solution 1**: Inverse Inheritance Relation
  - if the initial base-class has only additional behavior
    - e.g. Dog → DogNoWag
- **Solution 2**: Extract Common Base-Class
  - if both initial and derived classes have different behaviors
    - for Penguins → Birds, FlyingBirds, Penguins
- Classes with bad state
  - e.g. stupid or paralyzed dogs...

---

A Real Example

```
itsThirdPartyBoundedSet.Add(p);
```

Figure 2: Container Hierarchy
Problem

```
template <class T>
void PersistentSet::Add(const T& t)
{
    PersistentObject& p = // throw bad_cast
dynamic_cast<PersistentObject&>(t);
    itsThirdPartyPersistentSet.Add(p);
}
```

LSP Compliant Solution

Separate PersistentSet from Set.
Elements of Bad Design

Software Rigidity

- Rigidity is the tendency for software to be difficult to change, even in simple ways.
- Symptom: Every change causes a cascade of subsequent changes in dependent modules.
- Effect: When software behaves this way, managers fear to allow developers to fix non-critical problems. This reluctance derives from the fact that they don’t know, with any reliability, when the developers will be finished.
Software Fragility

- Fragility is the tendency of the software to break in many places every time it is changed. Often the breakage occurs in areas that have no conceptual relationship with the area that was changed.
- Symptom: Every fix makes it worse, introducing more problems than are solved.
- Effect: Every time managers/team leaders authorize a fix, they fear that the software will break in some unexpected way.

Software Immobility

- Immobility is the inability to reuse software from other projects or from parts of the same project.
- Symptom: A developer discovers that he needs a module that is similar to one that another developer wrote. But the module in question has too much baggage that it depends upon. After much work, the developer discovers that the work and risk required to separate the desirable parts of the software from the undesirable parts are too great to tolerate.
- Effect: And so the software is simply rewritten instead of reused.
Software Viscosity

- Viscosity is the tendency of the software/development environment to encourage software changes that are *hacks* rather than software changes that preserve original design intent.
- Symptom: It is easy to do the wrong thing, but hard to do the right thing.
- Effect: The software maintainability degenerates due to hacks, workarounds, shortcuts, temporary fixes etc.

Why bad design results?

- Obvious reasons: lack of design skills/design practices, changing technologies, time/resource constraints, domain complexity etc.
- Not so obvious:
  - Software rotting is a slow process .. Even originally clean and elegant design may degenerate over the months/years..
  - Unplanned and improper module dependencies creep in; Dependencies go unmanaged.
  - Requirements often change in the way the original design or designer did not anticipate ..
Example of Rigidity and Immobility

```c
enum OutputDevice {printer, disk};
void Copy(OutputDevice dev){
    int c;
    while((c = ReadKeyboard()) != EOF)
        if(dev == printer)
            WritePrinter(c);
        else
            WriteDisk(c);
}
```

Dependency Inversion Principle

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

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

- OCP states the **goal**; DIP states the **mechanism**

- A base class in an inheritance hierarchy should not know any of its subclasses
- Modules with detailed implementations are not depended upon, but depend themselves upon abstractions
Lecture 3

Object-Oriented Design

Procedural vs. OO Architecture

Procedural Architecture

Object-Oriented Architecture

DIP Applied on Example

```cpp
class Reader {
    public:
        virtual int read() = 0;
};

class Writer {
    public:
        virtual void write(int) = 0;
};

void Copy(Reader& r, Writer& w){
    int c;
    while((c = r.read()) != EOF)
        w.write(c);
}
```
DIP Related Heuristic

Design to an interface, not an implementation!

- Use inheritance to avoid direct bindings to classes:

  ![Diagram showing design to an interface](image)

Design to an Interface

- **Abstract classes/interfaces:**
  - tend to change much less frequently
  - abstractions are ‘hinge points’ where it is easier to extend/modify
  - shouldn’t have to modify classes/interfaces that represent the abstraction (OCP)

- **Exceptions**
  - Some classes are very unlikely to change;
    - therefore little benefit to inserting abstraction layer
    - Example: String class
  - In cases like this can use concrete class directly
    - as in Java or C++
Avoid Transitive Dependencies

- Avoid structures in which higher-level layers depend on lower-level abstractions:
  - In example below, Policy layer is ultimately dependant on Utility layer.

```
+----------------+       +----------------+       +----------------+
| Policy Layer   |   Depends on | Mechanism Layer | Depends on | Utility Layer |
+----------------+       +----------------+       +----------------+
```

Solution to Transitive Dependencies

- Use inheritance and abstract ancestor classes to effectively eliminate transitive dependencies:

```
+----------------+       +----------------+       +----------------+
| Policy Layer   |   depends on | Mechanism Interface |       | Utility Interface |
+----------------+       +----------------+       +----------------+       +----------------+
                               
| Mechanism Layer |       | Utility Interface |
+----------------+       +----------------+       +----------------+       +----------------+
```

DIP - Related Heuristic

When in doubt, add a level of indirection

- If you cannot find a satisfactory solution for the class you are designing, try delegating responsibility to one or more classes:

![Diagram showing Problem Holder, Problem Solver, and their relationships.]

When in doubt ...

- It is generally easier to remove or by-pass existing levels of indirection than it is to add them later:

![Diagram showing Blue class's indirect message calls to Red class fail to meet criteria, so Blue class re-implements responsibilities directly.]
Another Example

Figure 5: Naive Button/Lamp Model

Using the DIP

Figure 6: Inverted Button Model
Object-Oriented Design

----------buttonClient.h----------
class ButtonClient
{
    public:
        virtual void TurnOn() = 0;
        virtual void TurnOff() = 0;
};
----------button.h--------------
class ButtonClient;
class Button
{
    public:
        Button(ButtonClient&);
        void Detect();
        virtual bool GetState() = 0;
    private:
        ButtonClient* itsClient;
};

----------buttonImp.h----------
class ButtonImplementation:
public:
    ButtonImplementation(
        ButtonClient&);
    virtual bool GetState();
};

----------buttonImp.h----------
class ButtonImplementation:
public:
    ButtonImplementation(
        ButtonClient&);
    virtual bool GetState();
};

----------buttonImp.h----------
class ButtonImplementation:
public:
    ButtonImplementation(
        ButtonClient&);
    virtual bool GetState();
};

Listing 6: Inverted Button Model
The Founding Principles

- The three principles are closely related

- Violating either LSP or DIP invariably results in violating OCP
  - LSP violations are latent violations of OCP

- It is important to keep in mind these principles to get most out of OO development...

- ... and go beyond buzzwords and hype ;)