OOD Smells and Principles

C++ Object Oriented Programming

Pei-yih Ting

NTOUCS

Contents

Unpleasant Code Smells

Refactoring: Improving the Design of Existing Code by M. Fowler et. al.

1. Duplicated Code
2. Long Method
3. Large Class
4. Long Parameter List
5. Divergent Change
6. Shotgun Surgery
7. Feature Envy
8. Data Clumps
9. Primitive Obsession
10. Switch Statements
11. Parallel Inheritance Hierarchies
12. Lazy Class
13. Speculative Generality
14. Temporary Field
15. Message Chains
16. Middle Man
17. Inappropriate Intimacy
18. Alternative Classes with Different Interfaces
19. Incomplete Library Class
20. Data Class
21. Refused Bequest
22. Comments

https://sourcemaking.com/refactoring/bad-smells-in-code

Refactoring

- **Refactoring**: A change made to the internal structure of software to make it **easier to understand** and **cheaper to modify without changing its observable behavior**.
- **Refactor**: Restructure software by applying a series of refactorings **without changing its observable behavior**.
- Kent Beck's **two hats** metaphor in developing software:
  - You try to **add a new functionality**, and realize that it would be much easier if the code were structured differently.
  - So you swap hats and **refactor** for a while.
- **Refactorings**: https://sourcemaking.com/refactoring
  - Composing methods (Extract method, Inline method, Inline temp, …)
  - Moving features between objects (Move method, …)
  - Organizing data (Self encapsulate field, …)
  - Simplifying conditional expression (…)
  - Making method call simpler (…)
  - Dealing with generalization (…)

https://refactoring.com/catalog/

Bad Design Smells

- **Rigidity –** The system is hard to change because every change forces many other changes to other unrelated parts of the system
- **Fragility –** Changes cause the system to break in places that have no conceptual relationship to the part that was changed
- **Immobility –** It is hard to disentangle the system into components that can be reused in other systems.
- **Viscosity –** Doing things right is harder than doing things wrong.
- **Needless Complexity –** The design contains infrastructure that adds no direct benefit.
- **Needless Repetition –** The design contains repeating structures that could be unified under a single abstraction.
- **Opacity –** The design is hard to read and hard to understand. It does not express its intents well.

Agile Design

Software design involves iterations of the following steps:

- **Step 1**: Design and implement the required functions
- **Step 2**: Diagnose the problem following the **smell** of poor design and applying **design principles**
- **Step 3**: Solve the problem by applying appropriate **design pattern**

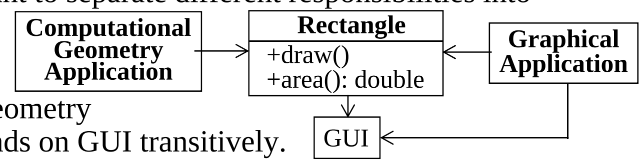
- Agile teams apply principles to remove bad smells. They don't apply principles when there are no smells.

- It is a mistake to unconditionally conform to a principle. Indeed, over-conformance to a principle leads to the design smell of Needless complexity.

Single Responsibility Principle

A class should have only one reason to change.

- Each responsibility is an axis of change. When the requirements change, that change is likely manifest through a change in responsibility amongst the classes.
- If a class has more than one responsibility, then the responsibilities become coupled. Changes to one responsibility may impair or inhibit the ability of the class to meet other requirements.
- Thus, it is important to separate different responsibilities into separate classes.



Possible problems:

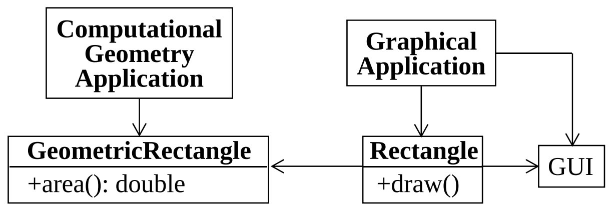
❶ Computational Geometry Application depends on GUI transitively.

❷ area() and draw() are two unrelated responsibilities

If **GraphicalApplication** causes draw() to change or GUI changes somehow, these changes force us to rebuild, retest, and redeploy the ComputationalGeometryApplication.

Separated Responsibilities

- Separate two responsibilities into two completely different classes by moving the computational portions of the **Rectangle** into the **GeometricRectangle** class.



- Now changes made to the way rectangles are rendered cannot affect the **ComputationalGeometryApplication**.

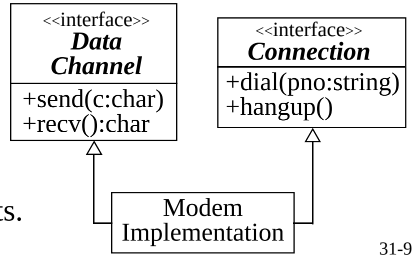
SRP Violation

```
class Modem {
public:
    void dial(string phoneNo);
    void hangup();
    void send(char c);
    char recv();
};
```

- Two responsibilities:
  - **connection management**
  - **data communication**

Should these two responsibilities be separated as two classes?

- Maybe not, it depends on how the application is changing.
  - If **connection management signature changes alone**, then the clients that use send() and recv() have to be recompiled and redeployed.
  - If, on the other hand, the application is not changing in ways that cause the two responsibilities to change at different times.
- Using separate interfaces (as used by Interface Segregation Principle) is another way to decouple the clients.



Open Closed Principle

***Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*.**

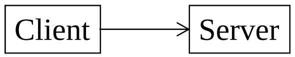
- **Open for extension:** the behavior of the module can be extended. As the requirements of the application change, we are able to extend the module with new behaviors that satisfy those requirement changes.
- **Closed for modification**: Extending the behavior of a module does not result in changes to the source or object code of the module, even the binary executable version of the module remains untouched.
- **How is it possible that the behaviors of a module can be modified without changing its source code? How can one change what a module does, without changing the module?**

the key is Abstraction

Interface (Design by Contract, DbC)

w/o Suitable Abstraction

- When a single change to a program results in a cascade of changes to dependent modules, the design smells of **Rigidity**.
- Violation of OCP: simple client-server
  Client is not open and closed.
  Whenever the server code changes, the client code must change.




```
struct Modem {
    enum Type {hayes, courrier, ernie} type;
};
struct Hayes {
    Modem::Type type;
    // Hayes related stuff
};
struct Courrier {
    Modem::Type type;
    // Courrier related stuff
};
struct Ernie {
    Modem::Type type;
    // Ernie related stuff
};
```

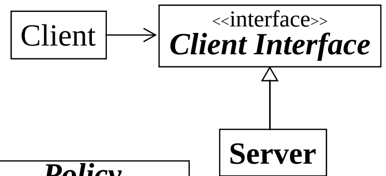
```
void logOn(Modem &m, string& pno, string& user, string& pw) {
    if (m.type == Modem::hayes)
        dialHayes((Hayes&)m, pno);
    else if (m.type == Modem::courrier)
        dialCourrier((Courrier&)m, pno, user);
    else if (m.type == Modem::ernie)
        dialErnie((Ernie&)m, pno, user, pw);
    // …
}
```

Adding a new modem would add
  else if (m.type == Modem::xxx)
    …
everywhere in its client programs

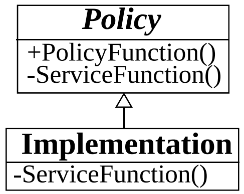
With Good Abstraction

- In C++, it is possible to create abstractions that are *fixed* and yet represent an *unbounded group of possible behaviors*. The abstractions are **abstract base classes**, and the unbounded group of possible behaviors is represented by all possible **derived classes**
- OCP conforming designs:

  ❶ **Strategy pattern**
  Client and Client Interface are both open and closed. program to a fixed interface (design-by-contract).



  ❷ **Template Method pattern**
  Policy is both open and closed.



- If OCP is applied well, further changes of that kind will be achieved by **adding new codes**, not by changing old codes that already work.

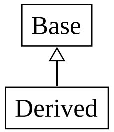
Liskov Substitution Principle

Subtypes must be substitutable for their base types.

- The importance of this principle becomes obvious when you **consider the consequences** of violating it.

```
void client(Base *bp) {
  ….
}
```

```
void main() {
  Derived dObj;
  f(&dObj);
}
```



- Will **client**() behaves normally when dObj is passed as a Base?

  If the functionality of client(&dObj) breaks down, then dObj is not substitutable for a Base object.

- The author of client() will be tempted to put in some kind of test for Derived so that client() can behave properly when Derived is passed to it. Typically, this violates also OCP because now client() is not closed to various derived classes of Base.

Violation of LSP

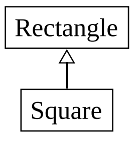
- Symptoms: “Using code to select code”, “downcast”, “type-flags”
- Usually cause violation of OCP

```
struct Point {
  double x, y;
};
struct Circle: public Point {
  double radius;
};

double areaTriangle(Point *vertices[3]) { // not closed
  for (int i=0; i<3; i++)
    if (dynamic_cast<Circle *>(vertices[i])) // cannot take a Circle
      return -1.0;
  … // calculate the area
}
```


Rectangle and Square

- A square IS-A rectangle with equal width and height in mathematical sense. A sort of **specialization**.
- Implementation:




```
class Rectangle {
public:
  virtual void setWidth(double w) {m_width=w;}
  virtual void setHeight(double h) {m_height=h;}
  double getWidth() {return m_width;}
  double getHeight() {return m_height;}
private:
  Point m_topLeft; double m_width, m_height;
};
```

```
class Square: public Rectangle {
public:
  void setWidth(double w) {Rectangle::setWidth(w); Rectangle::setHeight(w);}
  void setHeight(double h) {Rectangle::setWidth(h); Rectangle::setHeight(h);}
};
```

- **Is a Square substitutable for a Rectangle in all sorts of clients?**

Rectangle and Square (cont’d)

```
Square s;
s.setWidth(1); // set both width and height to 1
s.setHeight(2); // set both width and height to 2
// good, won’t be able to mess a square with different width and height
```

```
void f(Rectangle& r) {
  r.setWidth(32); // if r is a Square, width and height will be set to 32
}                 // if r is a Rectangle, only width is set to 32
```

```
void g(Rectangle& r) {  // this function breaks down if r is a Square
  r.setWidth(5);
  r.setHeight(4);
  assert(r.area() == 20);
}
```

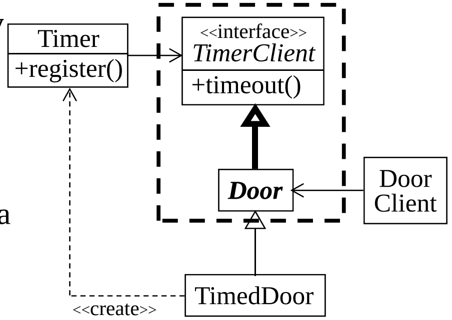
Violate LSP

```
void g(Rectangle& r) {
  if (dynamic_cast<Square *>(&r)==0) {
    r.setWidth(5); r.setHeight(4);
    assert(r.area() == 20);
  }
}
```

Interface Segregation Principle

- "**Fat**" interface: non-cohesive interface with diverse functionalities.
- Smells of **Rigidity** and **Viscosity**
- The interfaces of the class should be dissected into groups of methods. Each serves a different set of clients.

Example: In a security application, a door needs to sound an alarm when it has been left open for too long.




```
class Door: public TimerClient
{
public:
    virtual void lock() = 0;
    virtual void unlock();
    virtual bool isDoorOpen();
};
```

Interface Pollution

```
class Timer {
public:
    void register(int timeout, TimerClient *client);
};
class TimerClient {
public:
    virtual void timeout() = 0;
};
```


Separate Interfaces

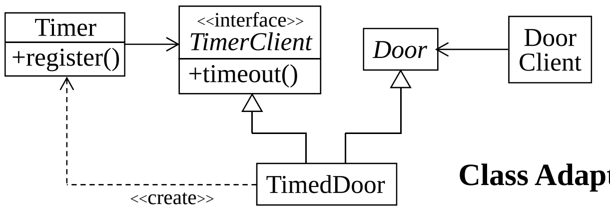
- Smells of **Rigidity** and **Viscosity**: changes of ***TimerClient*** interface affect the **clients of *Door* interface** and force recompilation.
- **Violation of LSP**: if a door does not have timeout feature, this new Door-derived class, although inherit *Door* interface, has to give a nil implementation of timeout().

- If classes with multiple responsibilities are unavoidable, at least avoiding fat/non-cohesive interface, so that clients of a particular interface do not know and affected by changes on unrelated interface.
- **Decoupling** clients means separate interfaces: since the clients Timer and DoorClient are separate, the interfaces should also be separate.
- **Interface Segregation Principle**:

Client should not be forced to depend on methods that they do not use.

Separation of Interfaces

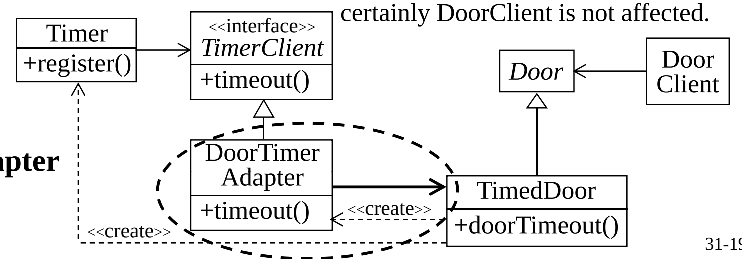
- Separation through **Multiple Inheritance**



Class Adapter Pattern

- Separation through **Delegation**

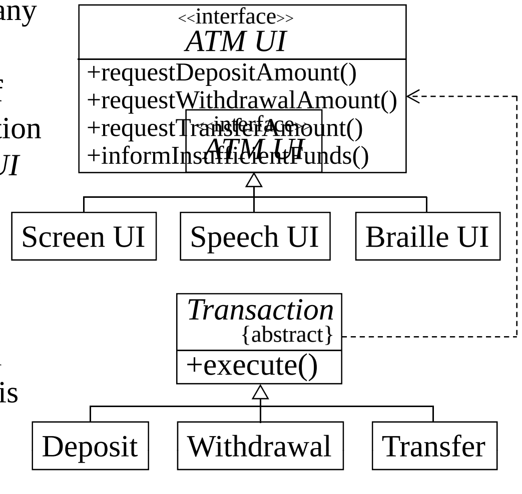
Even if *TimerClient* interface changes, doorTimeout() is not affected and certainly DoorClient is not affected.



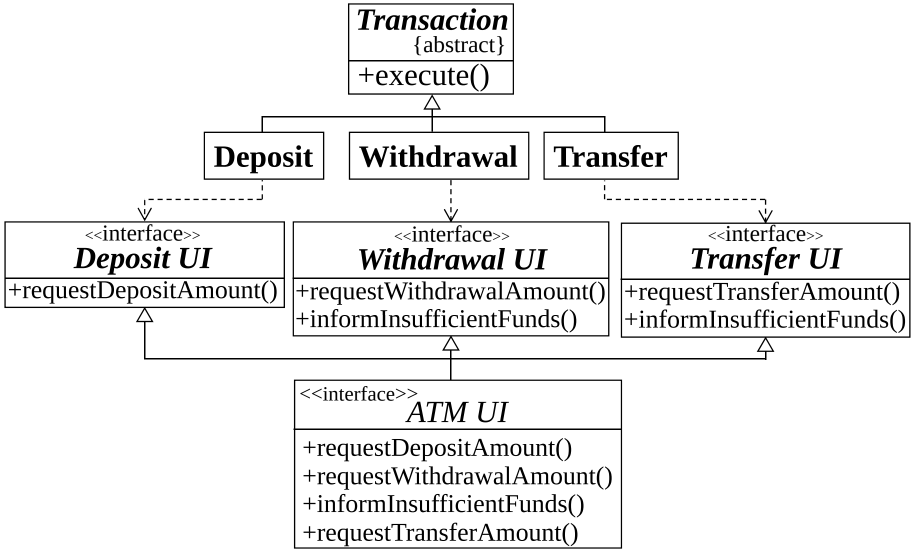
Object Adapter Pattern

ATM User Interface Example

- The user interface of an automated teller machine (ATM) needs to be very flexible – there are many forms of interfaces.
- There are different types of transactions. Each transaction uses methods of the *ATM UI* that no other classes uses.
- If we want to add a PayGasBill transaction, we would have to add new methods to *ATM UI* to deal with specific messages. This change would affect all transaction classes.
- Smells of **Rigidity** and **Viscosity**



Separation of *ATM UI* Interfaces



Dependency Inversion Principle

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

b. Abstractions should not depend on details. Instead, Details should depend on Policy.

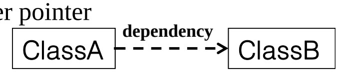
- **Traditional top-down "structured analysis and design"** tends to create software structures in which
  - high-level modules depend on well-developed low-level modules
  - policy depends on details

  because high-level policy modules make function calls to low-level library modules.
- The dependency structure of a **well-designed, object-oriented program** is "**inverted**" with respect to the dependency structure that normally results from traditional procedural designs.

Dependency Management

- Dependency between ClassA and ClassB: a change in the interface of ClassB necessitate changes in the implementation of ClassA
  - ClassA has a ClassB member object or member pointer
  - ClassA is derived from ClassB
  - ClassA has a function that takes a parameter of type ClassB
  - ClassA has a function that uses a static member of ClassB
  - ClassA sends a message (a method call) to ClassB

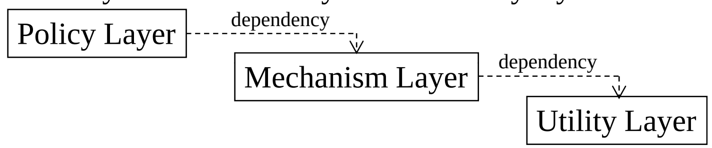
  In each case, it is necessary to #include "classB.h" in classA.cpp.



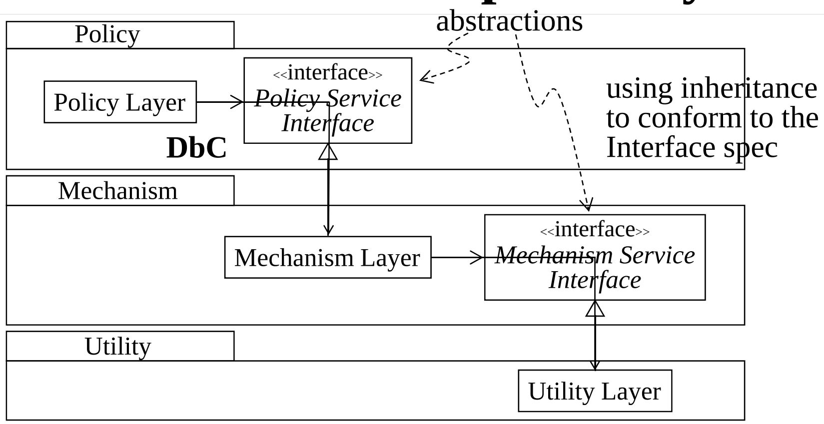
- Code reuse, an important goal, always produces dependencies.
- When designing classes and libraries it is important to make sure that we produce **as few** unnecessary or unintentional dependencies **as possible** because they slow down compile and reduce reusability.
- Forward class declarations make it possible for classes to have *circular relationships* without having *circular dependencies* between header files.

Application's Most Valuable Part

- The high-level modules contain the important **policy decisions** and **business models** of an application.
- It is the high-level, policy-setting modules that ought to be influencing the low-level, detailed modules (Mechanism and Utility).
- It is the high-level, policy-setting modules that we want to **reuse**, i.e. the "**factoring**" style of reuse. When high-level modules depend on low-level modules, it becomes very difficult to reuse those high-level modules in different contexts.
- **DIP is at the very heart of framework design.**
- **Naïve layering scheme**: policy layer is sensitive to changes in mechanism layer and all the way down to utility layer



Inversion of Dependency



- ✧ Lower-level modules provide the implementation for interfaces.
- ✧ **Inversion of interface ownership:** interface belongs to its client, instead of the class that implements it.
- ✧ Policy Layer is unaffected by any changes to Mechanism Layer or Utility Layer

Fundamental Theorem of Software Engineering (FTSE)

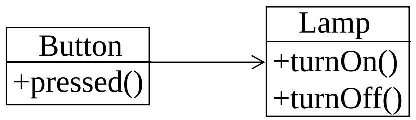
"We can solve any problem by introducing an extra level of indirection."

originated by Andrew Koenig

- ✧ This is a general principle for managing complexity through **abstraction**.
- ✧ except for the problem of too many levels of indirection

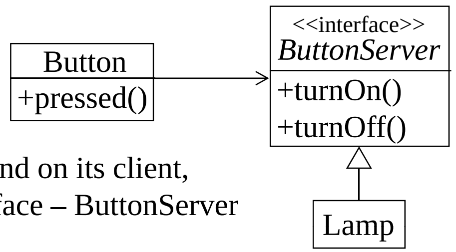
Another DIP Example

- ✧ Dependency inversion can be applied **wherever one class sends a message to another**.
- ✧ Naïve Model



Should a Button class always depend on the Lamp class?

- ✧ DIP applied



An interface does not depend on its client, thus, the name of the interface – ButtonServer

can be renamed to something more generic like SwithchableDevice

Law of Demeter (LoD)

- ✧ A specific case of **loose coupling**
  - ✶ Each unit should have only limited knowledge about other units
  - ✶ Each unit should only talk to its immediate friends (do not pry into the privacy of your friend)

a given object should assume as little as possible about the structure or properties of anything else (including its subcomponents), in accordance with the principle of information hiding

- ✧ **Least Knowledge Principle**
- ✧ The method **m** of an object **O** may only invoke methods of
  - ✶ **O** itself
  - ✶ **m**'s parameters
  - ✶ **O**'s direct components
  - ✶ Any objects created within **m**
- ✧ **avoid invoking methods of an object returned by another method**
  - ✶ **a.b.c.method()** e.g. when one wants a dog to walk, one does not command the dog's legs to walk directly; instead one commands the dog which then commands its own legs.

Law of Demeter (cont'd)

- **Example** from Apache that might violate this rule:
  ctxt.getOptions().getScratchDir().getAbsolutePath()
- It's not the problem of chaining calls. It could still violate the rule if decomposed as ops = ctxt.getOptions();
  scratchDir = opts.getScratchDir();
  scratchDir.getAbsolutePath();
- **Consider** instead: ctxt.createScratchFileStream(classFileName);

- **Example** "Paperboy & Wallet" that violates this rule:

```
if (myCustomer.getWallet().getTotalMoney() > bill)
    myCustomer.getWallet().subtractMoney(bill);
```

- **Wrapper** solution: if (myCustomer.getPayment(bill)) …
- Again, chaining calls is not the problem, it's only a phenomenom. The real issue is whether **Walltet Customer::getWallet()** breaks the encapsulation of class **Customer**.

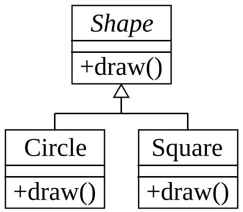
Law of Demeter (cont'd)

- Chaining calls are fine if target object is public or is itself or a friend
  - canvas.getDimension().getWidth()
  - stringBuilder.append(..).delete(..).insert(..)
- Unplesant code smells - **Feature Envy**: A method accesses the data of another object more than its own data
- Advantages:
  - resulting software are more **maintainable** and **adaptable** since objects are less dependent on the internal structure of other objects.
  - **narrower** interface in the method level
- Disadvantages:
  - have to write **many wrapper methods** to propagate calls to components
  - **wider** interface in the class level

Single Choice Principle

Whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list.

- Assume we have a graphic system with the Shape- Circle-Square class hierarchy describing objects drawable on the screen.



- Assume that these graphical objects are serialized in the file as

```
define share {
type=circle
location=25,6
…
}
define shape {
type=square
location=36,10
…
}
```

```
ArrayList shapes;
if (type=="circle")
   shapes.add(new Circle(filestream));
else if (type=="square")
   shapes.add(new Square(filestream));
…
else if (type=="XXX")
   shapes.add(new XXX(filestream));
```

This exhaustive list should appear only once in the program and no more.

Other OOD Principles

- Don't Repeat Yourself
- Program to an Interface, Not an Implementation (DbC)
- Depend on Abstractions, Not Concrete classes
- Hollywood Principle - Don'd call us, we'll call you (DIP)
- Encapsulate What Varies.
- Favor Composition over Inheritance
- Apply Design Pattern wherever possible
- Strive for Loosely Coupled System
- Keep it Simple and Sweet / Stupid
- Principle of Least Astonishment
- Package Cohesion Principles
- Package Coupling principle