Design principles

OOD: Lecture 4
Next lecture

• UML: Thursday, Sep 20, at 8:15 am in 1211
Reminder: Readings


• Wikipedia's entry on Object-oriented Design
“Software rot”

• Increased difficulty to adapt and maintain
• Causes
  – Communication/Documentation breakdown
    • Maintainers not fully familiar with the original design principles --> change works, but...
  – Design is not resilient in the face of change
Symptoms of rotting design

• **Rigidity**
  – Every change causes a cascade of subsequent changes in dependent modules. “2 days -> 2 months”

• **Fragility**
  – Breaks in many places when a change is made

• **Immobility**
  – Reuse is more work than creating from scratch

• **Viscosity**: The law of least resistance when faced with a choice
  – *Design* viscosity: Hacks are easier/faster than preserving the design
  – *Environment* viscosity: Slow cycle time -> fastest choice
Dependency management

• Rigidity, fragility, immobility, and viscosity are all four – arguably – caused by an improper dependency structure

• Three groups of preventive principles / guidelines
  – Class design
  – Package cohesion
  – Package coupling
Principles of object-oriented class design

SOLID:

• **SRP**: The single responsibility principle
• **OCP**: The Open Closed principle
• **LSP**: The Liskov substitution principle
• **ISP**: The interface segregation principle
• **DIP**: The dependency inversion principle
SRP
The single responsibility principle

• A class should have one, and only one, reason to change.

• A class should have a single responsibility

• Example

```java
interface Modem {
    // Modem.java -- SRP Violation
    public void dial(String phoneNumber);
    public void hangup();
    public void send(char c);
    public char receive();
}
```
SRP continued

• Two responsibilities
  – Connection management: *dial* and *hangup*
  – Data communication: *send* and *receive*

• Better

  – Nothing depends on the modem implementation class
OCP
The Open Closed principle

• A module should be open for extension but closed for modification.
  – Ability to change what the module does, without changing its source code
  – Techniques based on abstraction
    • Dynamic polymorphism
    • Static polymorphism
Dynamic polymorphism
OCP violation example

```cpp
struct Modem {
    enum Type {hayes, courier, ernie} type;
};
struct Hayes {
    Modem::Type type;
    // Hayes related stuff
};
struct Courier {
    Modem::Type type;
    // Courier 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::courier) {
        DialCourier((Courier&)m, pno);
    } else if (m.type == Modem::ernie) {
        DialErnie((Ernie&)m, pno)
        // ...
    }
}
```
OCP: Dynamic polymorphism continued

class Modem {
    public:
    virtual void Dial(const string& pno) = 0;
    virtual void Send(char) = 0;
    virtual char Recv() = 0;
    virtual void Hangup() = 0;
};

void LogOn(Modem& m, string& pno,
            string& user, string& pw) {
    m.Dial(pno);
    // you get the idea.
OCP: Static polymorphism

• Templates/Generics

```cpp
template <typename MODEM>
    void LogOn(MODEM& m, string& pno,
                string& user, string& pw) {
        m.Dial(pno);
        // ...
    }
```
LSP

The Liskov substitution principle

- Derived classes must be substitutable for their base classes.
- The contract of the base class must be honoured by the derived class.
- A derived class is substitutable for its base class if:
  - Its pre-conditions are no stronger than the base class method.
  - Its post-conditions are no weaker than the base class method.
- Or, in other words, derived methods should expect no more and provide no less.
LSP violation
The Circle/Ellipse dilemma

• A circle is an ellipse
LSP violation 2

• A client code fragment:

```java
void f(Ellipse e) {
    Point a = new Point(0, 1);
    Point b = new Point(1, 0);
    e.setFoci(a, b);
    e.setMajorAxis(3);
    assert e.getFocus1() == a;
    assert e.getFocus2() == b;
    assert e.getMajorAxis() == 3;
}
```
LSP violation 3

• Ugly client-side fix

```java
def f(Ellipse e) {
    if (e.getClass().equals(Ellipse.class)) {
        //...
    } else {
        throw new Exception("Not a real ellipse");
    }
}
```
ISP
The interface segregation principle

• Make fine grained interfaces that are client specific.
Or
Many client specific interfaces are better than one general purpose interface

• [Kent] Do not change interfaces unless absolutely necessary, and especially do not change method signatures
DIP
The dependency inversion principle

• *Depend on abstractions, not on concretions.*

• The primary mechanism of OO design

• No dependency should target a concrete class
  – Non-volatile classes (e.g. Java core library classes) tend to cause less problems
Principles of package cohesion

- **REP**: The release reuse equivalency principle
- **CCP**: The common closure principle
- **CRP**: The common reuse principle

Note that these three exist in a balance. They can’t all three be completely satisfied at the same time.
REP

The release reuse equivalency principle

• *The granule of reuse is the granule of release.*
• Package together what would be reused together
• Support and maintain older versions
• Simplifies reuse
CCP
The common closure principle

- *Classes that change together are packaged together.*
- Minimizes configuration management (CM) work
  - I.e. management, test, and release of packages
- Simplifies development and maintenance
- Tends towards big packages
CRP
The Common Reuse Principle

- *Classes that aren’t reused together should not be grouped together.*
- Complement of REP
- Avoid forcing unnecessary client re-building
- Simplifies reuse
- Tends to small packages
Principles of package coupling

- **ADP**: The acyclic dependencies principle
- **SDP**: The stable dependencies principle
- **SAP**: The stable abstractions principle
ADP
The acyclic dependencies principle

• *The dependency graph of packages must have no cycles.*

• Cycles increase the work to re-build and eventually make every package depend on every other package

• Breaking a cycle
  – New package: Break out of dependency target
  – Apply dependency inversion (DIP) + interface segregation (ISP)
ADP: Breaking a cycle
Applying DIP & ISP
The stable dependencies principle

- *Depend in the direction of stability.*
- A way of reducing the number of packages that are hard to change because changes would propagate to many other packages

\[ \text{Instability} = \frac{C_e}{C_a + C_e} \]

- Depend upon packages whose Instability metric is lower than yours
The stable abstractions principle

- Abstractness increases with stability or Stable packages should be abstract packages.
- Can be seen as a re-formulation of dependency inversion (DIP)
- Abstract – stable – easy to extend (OCP)
- Concrete – instable – easy to change
Next lecture

• Thursday, Sep 20, at 8:15 am in 1211