Common mistakes
Basic Design Principles

Amit Shabtay

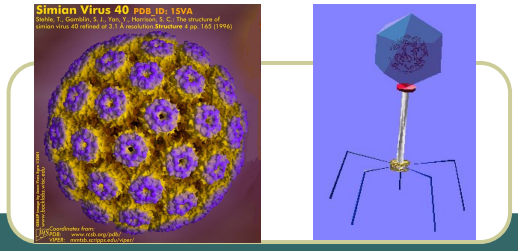
Tirgul Summery

- Basic design principles
- Advanced design principles (LSP, …)
- Intro to eclipse, unit testing, JUnit
- Generic programming (STL, Java generics)
- AspectWerkz- AOP framework
- ODBC,JDBC
- Exercise previews and reviews

Course Requirement

- Basic understanding of OOD
- Basic knowledge of C++, Java
- 3 programming exercises
- 2 theoretical exercises
- Exam

Basic Design Principles



Common Mistakes

- Repeated often
  - Especially with the inexperienced
- Don't you make them!
- How to recognize the danger signals?

Danger Signals (1)

```
public class Counter {
  public int howManyA(String s) {
    int conut = 0;
    for(int i = 0; i < s.length(); ++i)
      if(s.charAt(i) == 'a')
        ++count;
    return count;
  }
}
```

Is this a class? Too simple to be called an object.

Danger Signals (2)

```
Class City extends Place { … }
Class Jerusalem extends City
  implements Capital { … }
Class TelAviv extends City { … }
```

- What is wrong here?
  There can only one capital, so usually this is a bad design to have an interface for that (Not always the case)

Danger Signals (3)

```
Class Person {
   String getName(); void setName(String
     name);
   int getAge(); void setAge(int age);
   Car getCar(); void setCar(Car car);
}
```

- What do we see ?
  An Object that is strictly composed of getter and setter methods with no other functionaliry is also considered bad design usualy.

Basic Design Principles

- The Open Closed Principle
- The Dependency Inversion Principle
- The Interface Segregation Principle
- The Acyclic Dependencies Principle

- These principles and more: http://www.codeguru.com/forum/showpost.php?p=1092794&postcount=1

The Open Closed Principle

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

- Existing code should not be changed – new features can be added using inheritance or composition.
- Which is preferred? Composition.

Example

```
enum ShapeType
  {circle, square};
struct Shape {
  ShapeType _type;
};
struct Circle {
  ShapeType _type;
  double _radius;
  Point _center;
};
```

```
struct Square {
  ShapeType _type;
  double _side;
  Point _topLeft;
};
void DrawSquare(struct
  Square*)
void DrawCircle(struct
  Circle*);
```


Example (cont.)

```
void DrawAllShapes(struct Shape* list[], int n) {
  int i;
  for (i=0; i<n; i++) {
      struct Shape* s = list[i];
      switch (s->_type) {
            case square:
            DrawSquare((struct Square*)s);
            break;
            case circle:
            DrawCircle((struct Circle*)s);
            break;
      }
  }
}
```

Where is the violation?
If we want to add another shape.

Correct Form

```
class Shape {
public:
  virtual void Draw() const = 0;
};
class Square : public Shape {
public:
  virtual void Draw() const;
};
class Circle : public Shape {
pub
  v
};
```

```
void DrawAllShapes(Set<Shape*>& list) {
      for (Iterator<Shape*>i(list); i; i++)
            (*i)->Draw();
```

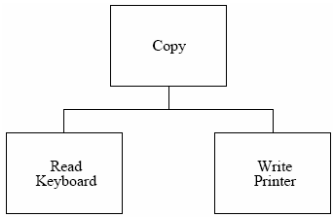

The Dependency Inversion Principle

A. High level modules should not depend upon low level modules. Both should depend upon abstractions.

B. Abstractions should not depend upon details. Details should depend upon abstractions.

Example

Where is the violation?
Adding another writing or reading device- there is a strong dependency on the implementation details.




```
void Copy() {
   int c;
   while ((c = ReadKeyboard()) != EOF)
     WritePrinter(c);
}
```


Example (cont.)

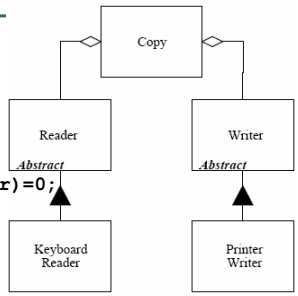
- Now we have a second writing device – disk

```
enum OutputDevice {printer, disk};

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


Correct form

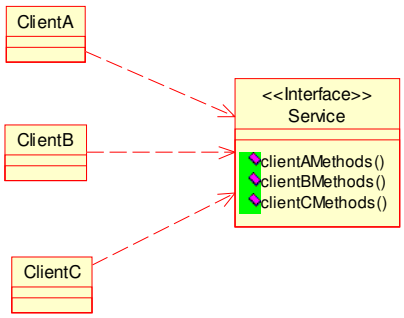
```
class Reader {
  public:
  virtual int Read() = 0;
};
class Writer {
  public:
    virtual void Write(char)=0;
};
void Copy(Reader& r,
          Writer& w) {
  int c;
  while((c=r.Read()) != EOF)
    w.Write(c);
}
```



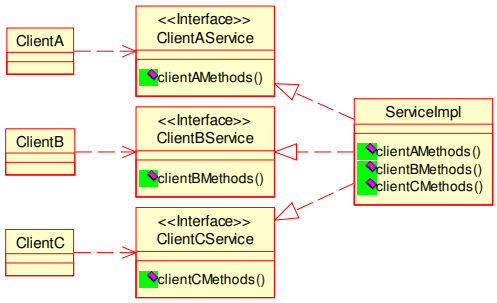
The Interface Segregation Principle

- The dependency of one class to another one should depend on the smallest possible interface.
- Avoid "fat" interfaces
- Example: Word toolbars
  (You can add or remove them as you like for simpler interface)

The Interface Segregation Principle



The Interface Segregation Principle



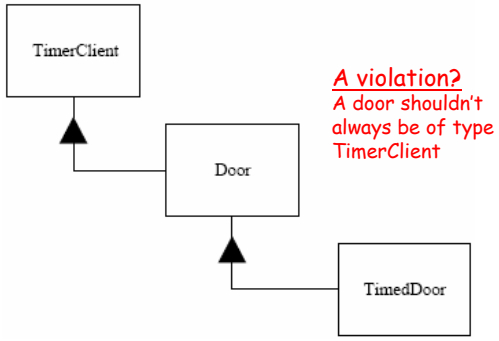
Example

```
class Timer {
  public:
  void Regsiter(int timeout,
       TimerClient* client);
};
```

```
class TimerClient {
  public:
  virtual void TimeOut() = 0;
};
```

```
class Door {
  public:
  virtual void Lock() = 0;
  virtual void Unlock() = 0;
  virtual bool IsDoorOpen() = 0;
};
```

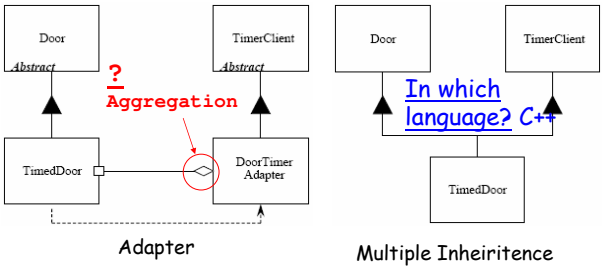

A Timed Door



A violation?
A door shouldn't always be of type TimerClient

Correct Form

- Two options:



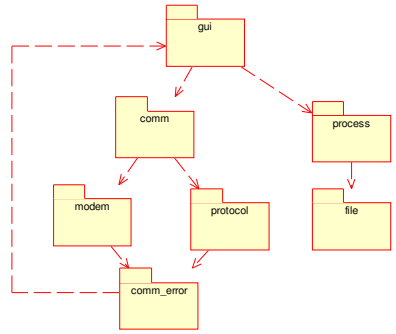
Adapter

Multiple Inheiritence

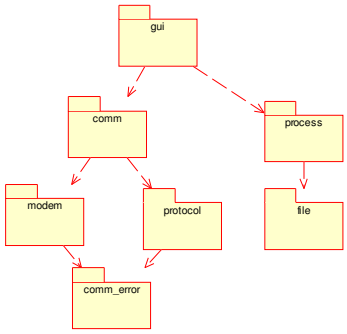
The Acyclic Dependencies Principle

- The dependency structure between packages must not contain cyclic dependencies.

Example



Correct Form

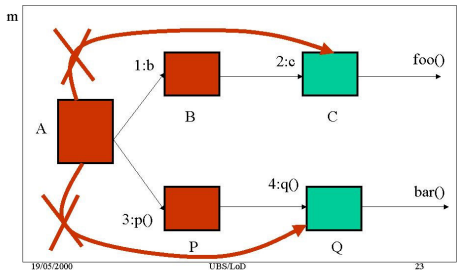


The Law Of Demeter

- Only talk to your immediate friends.
- In other words:
  - You can play with yourself. (`this.method()`)
  - You can play with your own toys (but you can't take them apart). (`field.method(), field.getX()`)
  - You can play with toys that were given to you. (`arg.method()`)
  - And you can play with toys you've made yourself. (`A a = new A(); a.method()`)

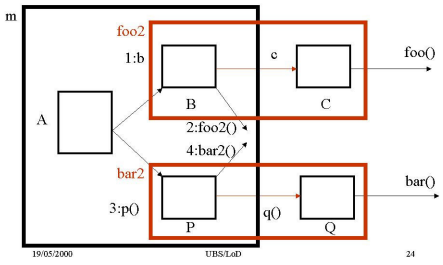
Example

Violations: Dataflow Diagram



How to correct

OO Following of LoD



Example Code

The Law of Demeter (cont.)
Violation of the Law

```
class A {public: void m(); P p(); B b; };
class B {public: C c; };
class C {public: void foo(); };
class P {public: Q q(); };
class Q {public: void bar(); };
void A::m() {
   this.b.c.foo(); this.p().q().bar();}
```

Resources

- A nice resources page for OOD:
- http://www.objectmentor.com

- About the principles (same site):

http://www.objectmentor.com/mentoring/OOPrinciples

Package cohesion

- The Common Closure Principle
  - Classes within a released component should share common closure. That is, if one needs to be changed, they all are likely to need to be changed.
- The Common Reuse Principle
  - The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.