Principles of Object-Oriented Design

The Object-Oriented … Hype

- What are object-oriented (OO) methods?
  - OO methods provide a set of techniques for analysing, decomposing, and modularising software system architectures
  - In general, OO methods are characterized by structuring the system architecture on the basis of its *objects* (and classes of objects) rather than the *actions* it performs
- What is the rationale for using OO?
  - In general, systems evolve and functionality changes, but objects and classes tend to remain stable over time
  - Use it for **large systems**
  - Use it for **systems that change often**

OO Design vs. OO Programming

- Object-Oriented Design
  - a method for decomposing software architectures
  - based on the objects every system or subsystem manipulates
  - relatively independent of the programming language used

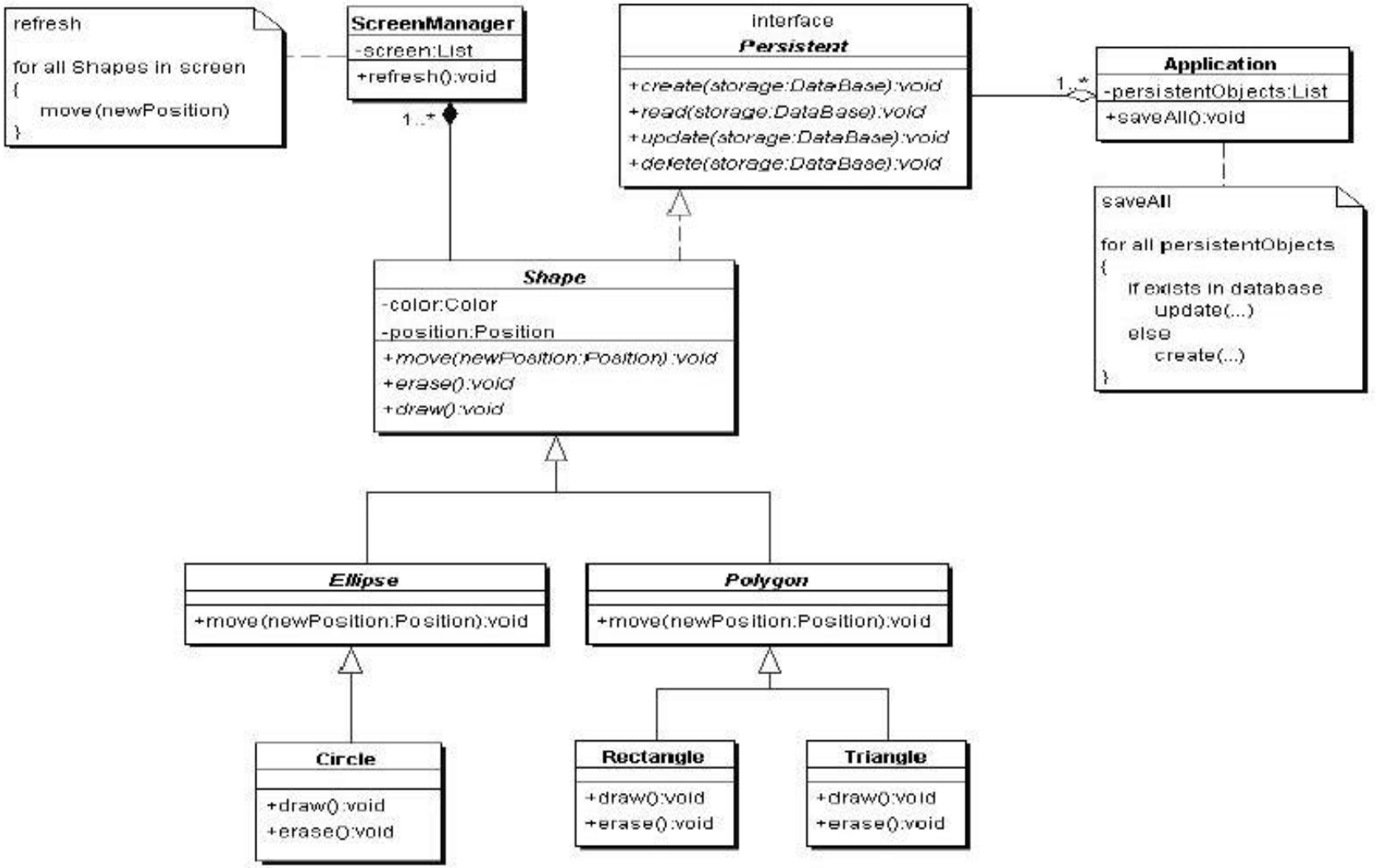
- Object-Oriented Programming
  - construction of software systems as
    - Structured collection of Abstract Data Types (ADT)
    - Inheritance
    - Polymorphism
  - concerned with programming languages and implementation issues

Polymorphism

- Behavior promised in the public interface of superclass objects
- implemented by subclass objects
  - in the specific way required for the subclass

- Why Is this Important?
  - Allow subclasses to be treated like instances of their superclasses
  - Flexible architectures and designs
    - high-level logic defined in terms of abstract interfaces
    - relying on the specific implementation provided by subclasses
    - subclasses can be *added without changing* high-level logic

Polymorphism Example



Signs of Rotting Design

- Rigidity
  - code difficult to change (*Continuity*)
  - management reluctance to change anything becomes policy
- Fragility
  - even small changes can cause cascading effects
  - code breaks in unexpected places (*Protection*)
- Immobility
  - code is so tangled that it's impossible to reuse anything
  - *Composability*
- Viscosity
  - much easier to hack than to preserve original design

Causes of Rotting Design

- Changing Requirements
  - is inevitable
  - "*All systems change during their life-cycles. This must be borne in mind when developing systems expected to last longer than the first version*". (I. Jacobson, OOSE, 1992)

- Dependency Management
  - the issue of coupling and cohesion
  - It can be controlled!
    - create *dependency firewalls*
    - see DIP example

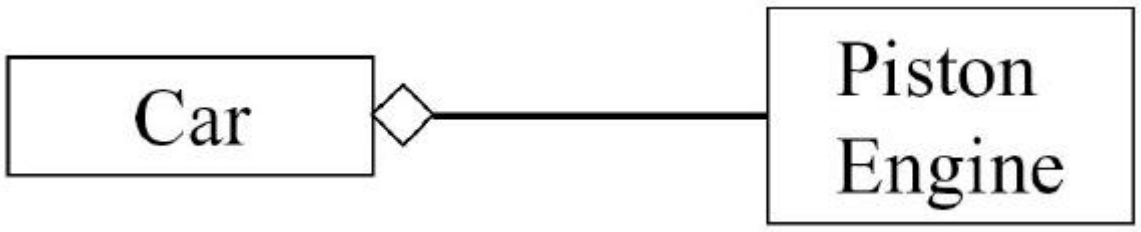
Open-Closed Principle (OCP)

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

> *Software entities should be open for extension, but closed for modification*
>
> **B. Meyer**, 1988 / quoted by **R. Martin**, 1996

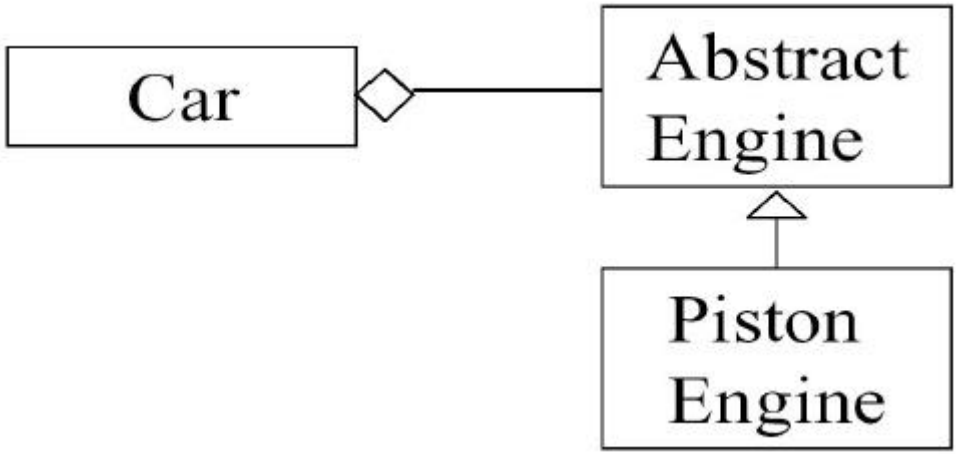
- Be open for extension
  - module's behavior can be extended
- Be closed for modification
  - source code for the module must not be changes
- *Modules should be written so they can be extended without requiring them to be modified*

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)

Strategic Closure

"*No significant program can be 100% closed* "

R.Martin, *"The Open-Closed Principle," 1996*

- 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, any time 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"

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

```
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!

```
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)

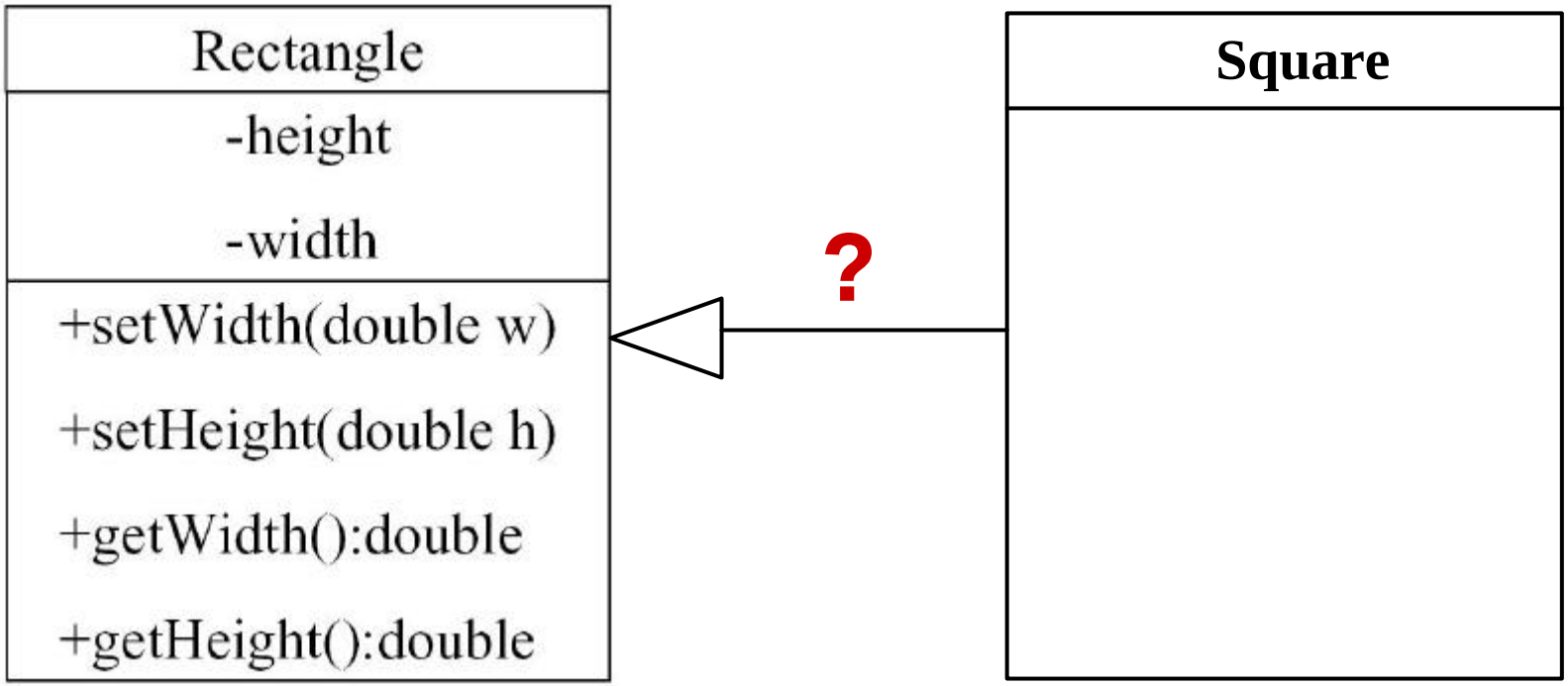
> *When redefining a method in a derivate class, you may only replace its* precondition by a weaker one*, and its* postcondition by a stronger one
>
> **B. Meyer**, 1988

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

```
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
```

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

```
void g(Rectangle& r) {
  r.setWidth(5); r.setHeight(4);
  // How large is the area?
}
```

  - 20! … Are you sure? ;-)

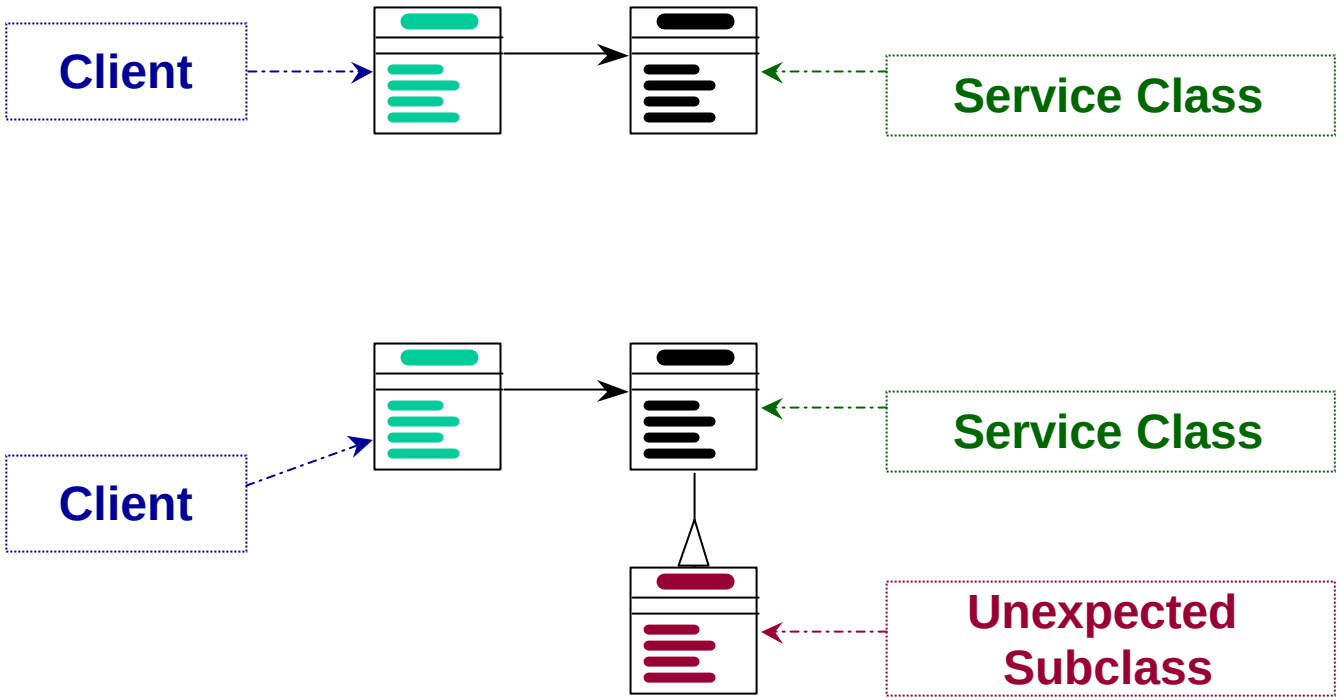
- IS-A relationship must refer to the **behavior** of the class!

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:

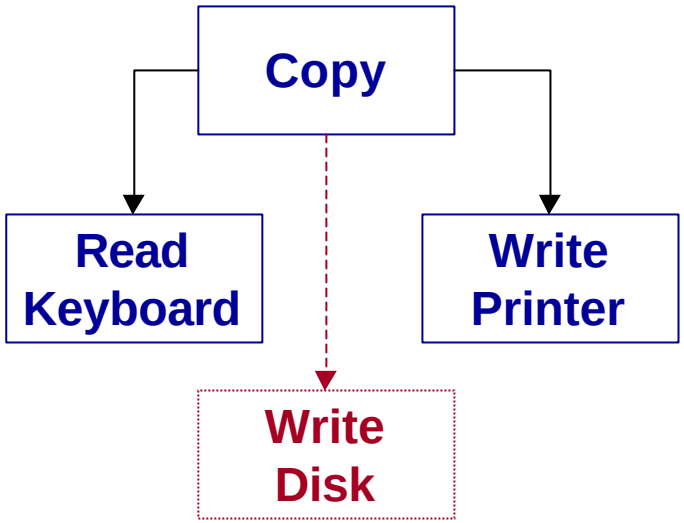


LSP Related Heuristic (2)

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...

Example of Rigidity and Immobility




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

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

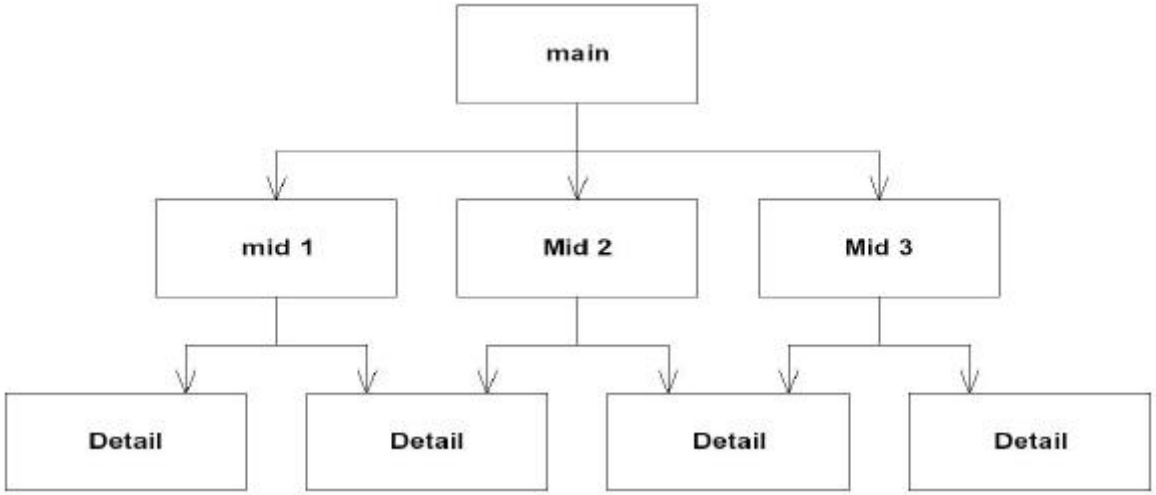
R. Martin, 1996

- OCP states the goal; DIP states the mechanism

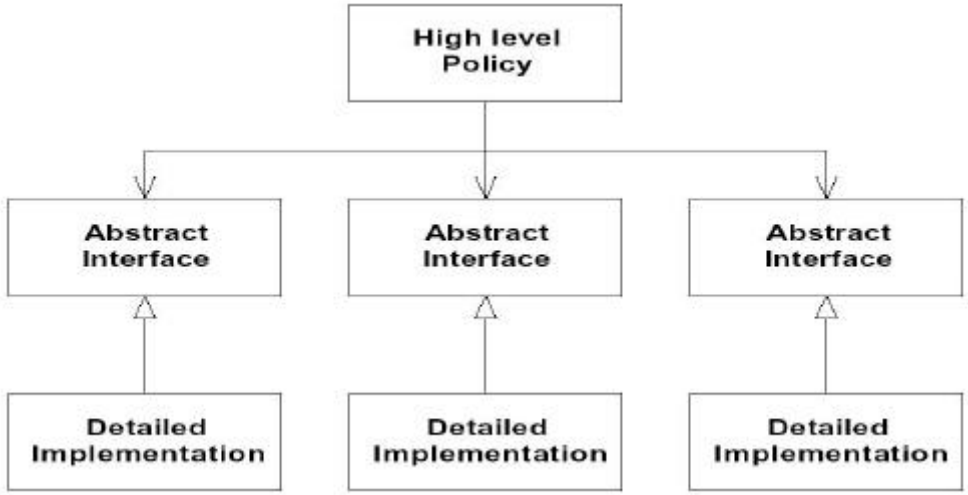
- A base class in an inheritance hierarchy should not know any of its subclasses
- Modules with detailed implementations are not depended upon, but depend themselves upon abstractions

Procedural vs. OO Architecture

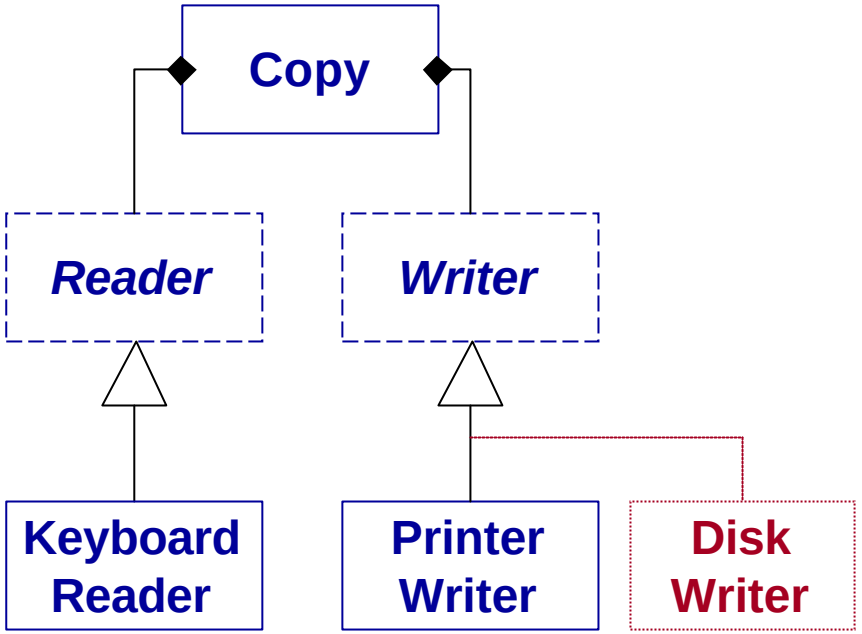
Procedural Architecture



Object-Oriented Architecture



DIP Applied on Example

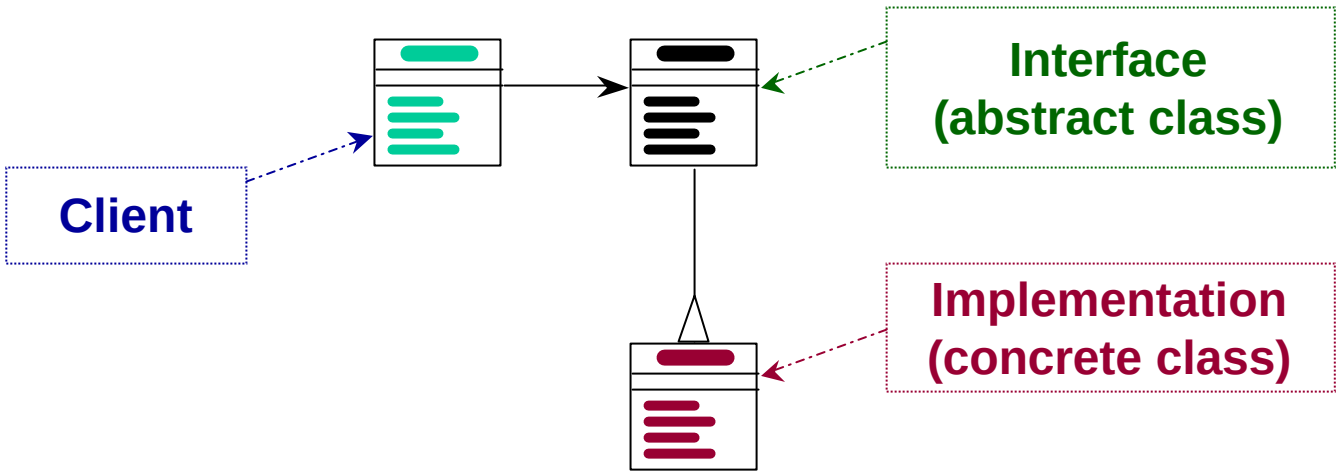



```
class Reader {
  public:
    virtual int read()=0;
};

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:



Design to an Interface

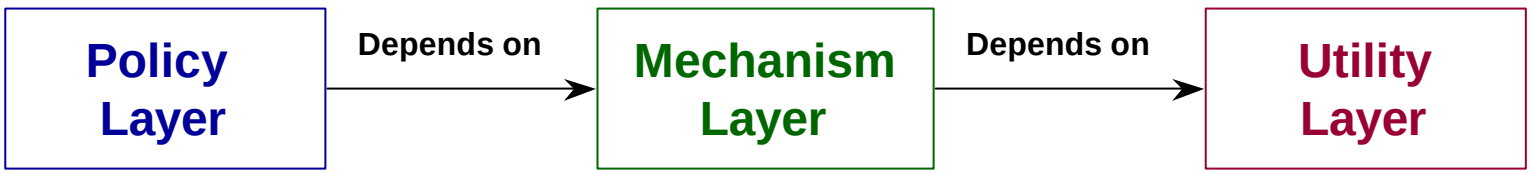
- **Abstract classes / interfaces:**
  - tend to change much less frequently
  - abstractions are 'hinge points' where it is easier to extend/modify
  - shouldn't have to modify classes/interfaces that represent the abstraction (OCP)

- 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++

DIP Related Heuristic

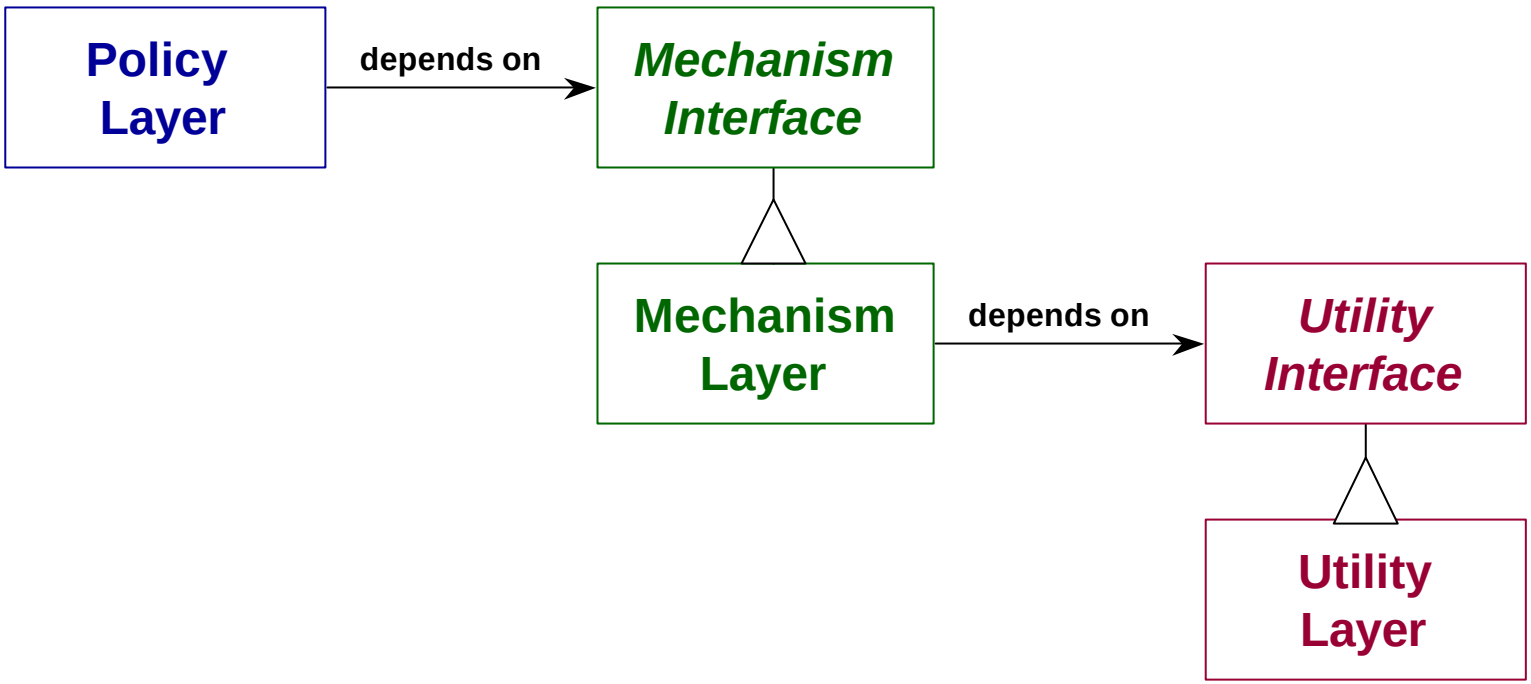
Avoid Transitive Dependencies

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



Solution to Transitive Dependencies

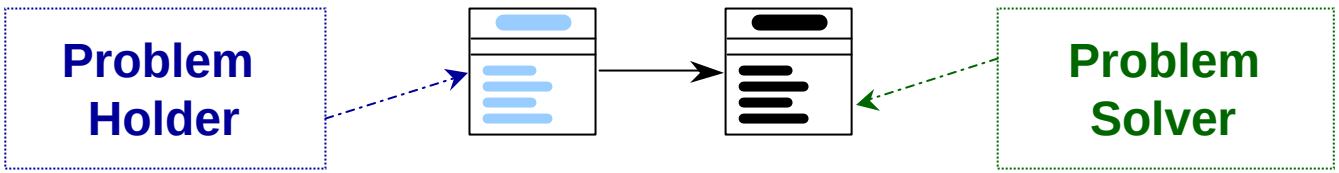
- Use inheritance and abstract ancestor classes to effectively eliminate transitive dependencies:



DIP - Related Heuristic

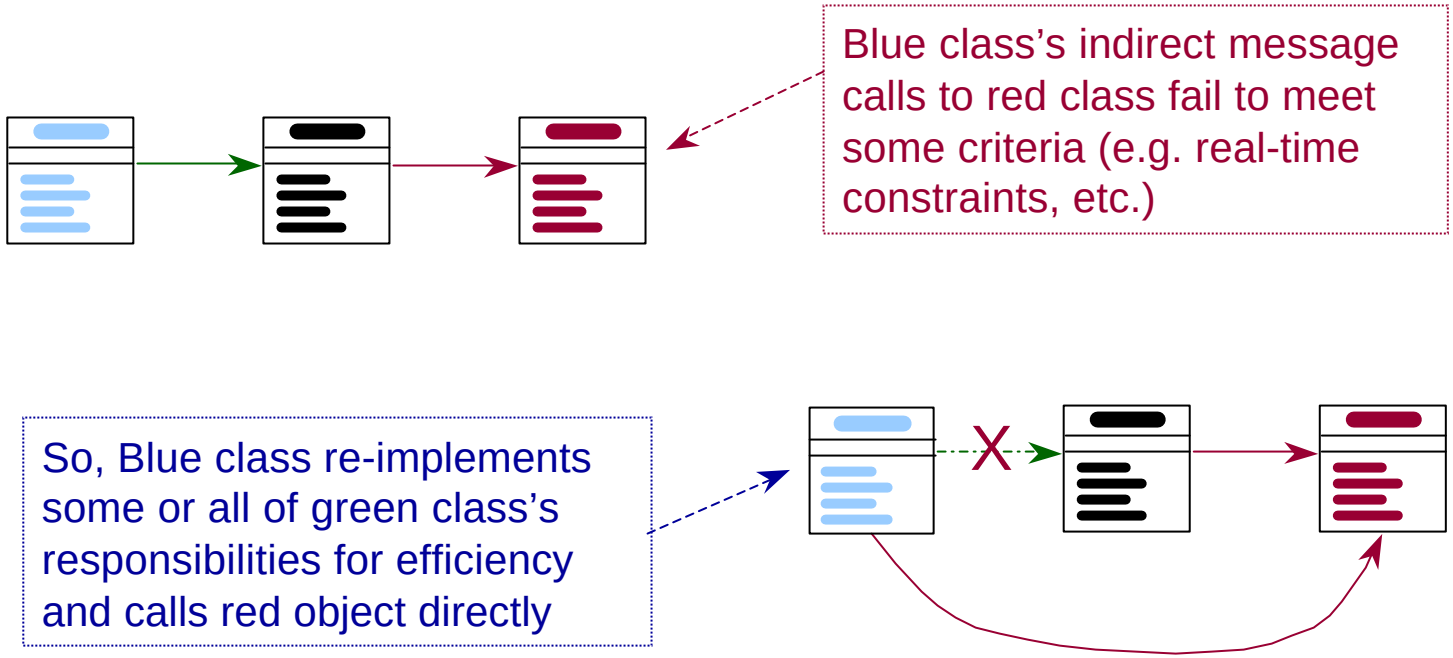
When in doubt, add a level of indirection

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



When in doubt ...

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



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 ;)