SOLID: Principles of OOD

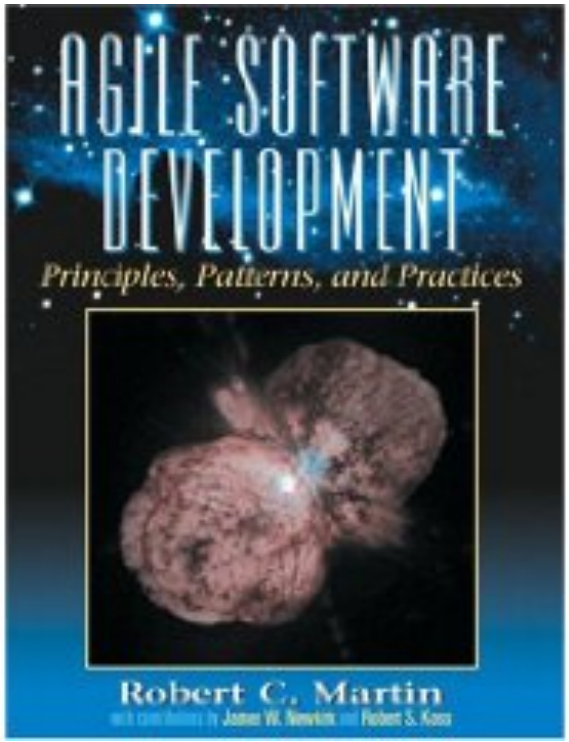
CS356 Object-Oriented Design and Programming
http://cs356.yusun.io
October 17, 2014

Yu Sun, Ph.D.
http://yusun.io
yusun@csupomona.edu

CAL POLY POMONA

Part of the presentation comes from

- Martin, Robert Cecil. Agile software development: principles, patterns, and practices. Prentice Hall PTR, 2003. APA



Author: Robert C. Martin
Nick Name: Uncle Bob

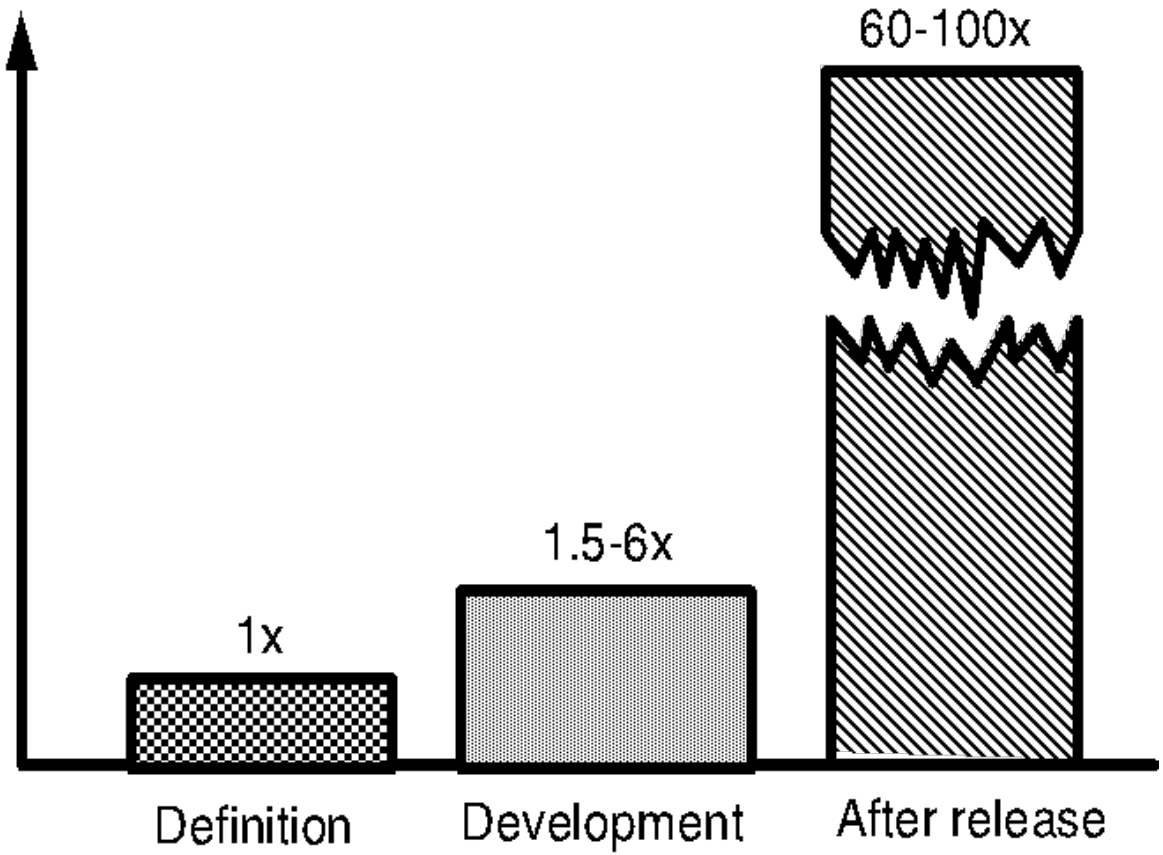
What is Software Design?

- The source code is the design
- UML diagram represents part of a design, but it is not the design
- Because the design can only be verified through source code
- The software design process includes coding, testing, refactoring…
- The programmer is real software designer

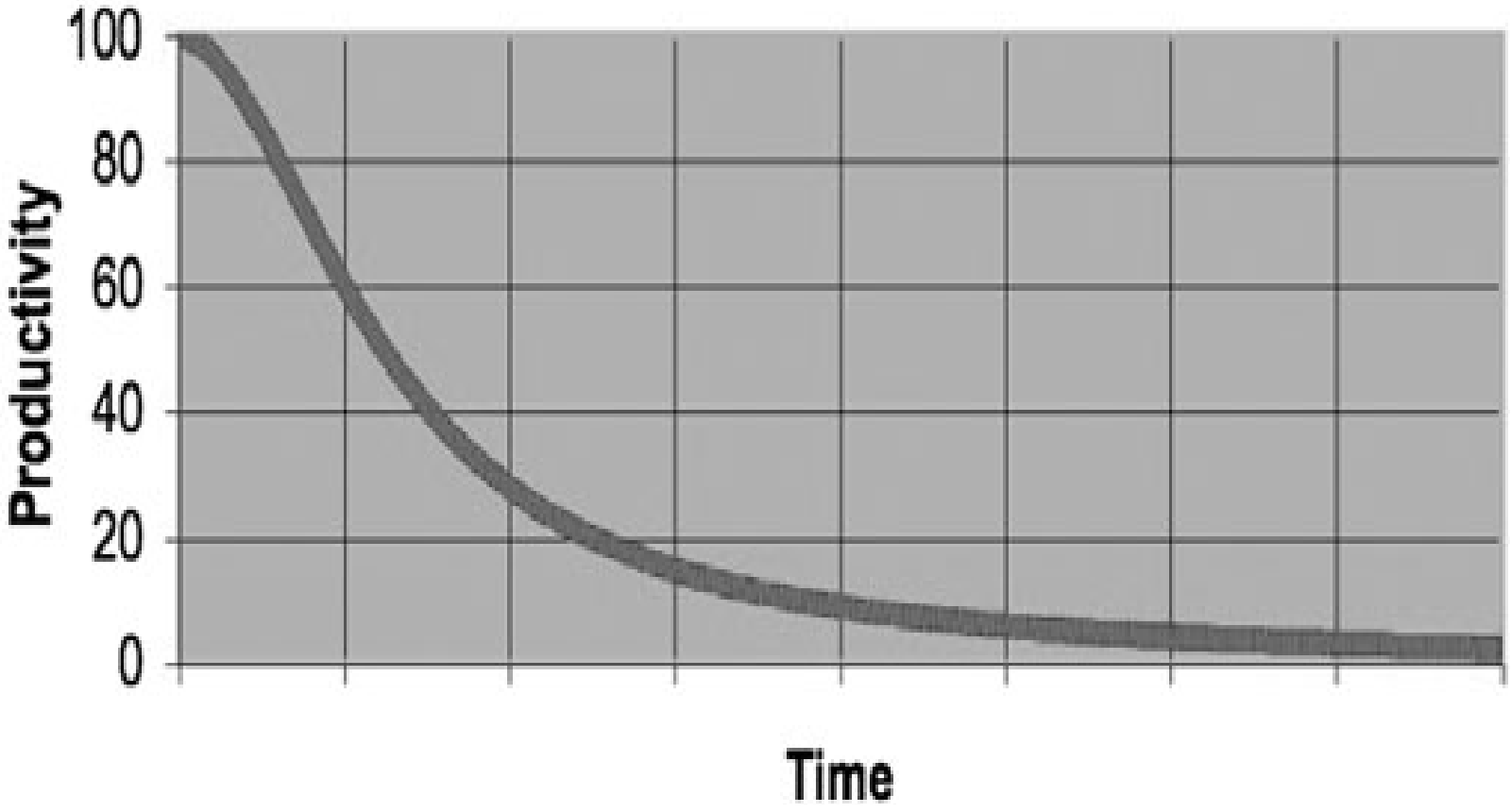
Software Nature – Software Entropy

- Software tends to degrade / decay
- Software rot – like a piece of bad meat

The Cost of Change



Developers Productivity vs. Time



Design Smells – The Odors of Rotting Software

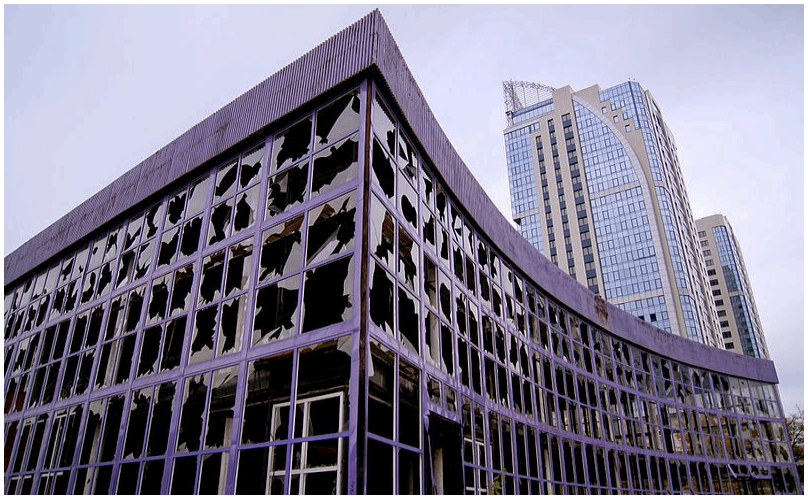
- Rigidity – The design is hard to change
- Fragility – The design is easy to break
- Immobility – The design is hard to reuse
- Viscosity – It is hard to do the right thing
- Needless complexity – Overdesign
- Needless Repetition – Mouse abuse
- Opacity – Disorganized expression

What Stimulates the Software to Rot?

- Requirements keep change – design degradation
- People change – violate the original design
- Tight schedule pressure

Psychology Reason: Broken Window Theory

- Came from city crime researcher
- A broken window will trigger a building into a smashed and abandoned derelict
- So does the software
- Don't live with the Broken window



How to Prevent Software from Rotting?

- Applies OO design principles
  - Bad design usually violates design principles
- Uses design patterns
- Follows agile practices
- Refactoring will reduce the software entropy

S.O.L.I.D Design Principles

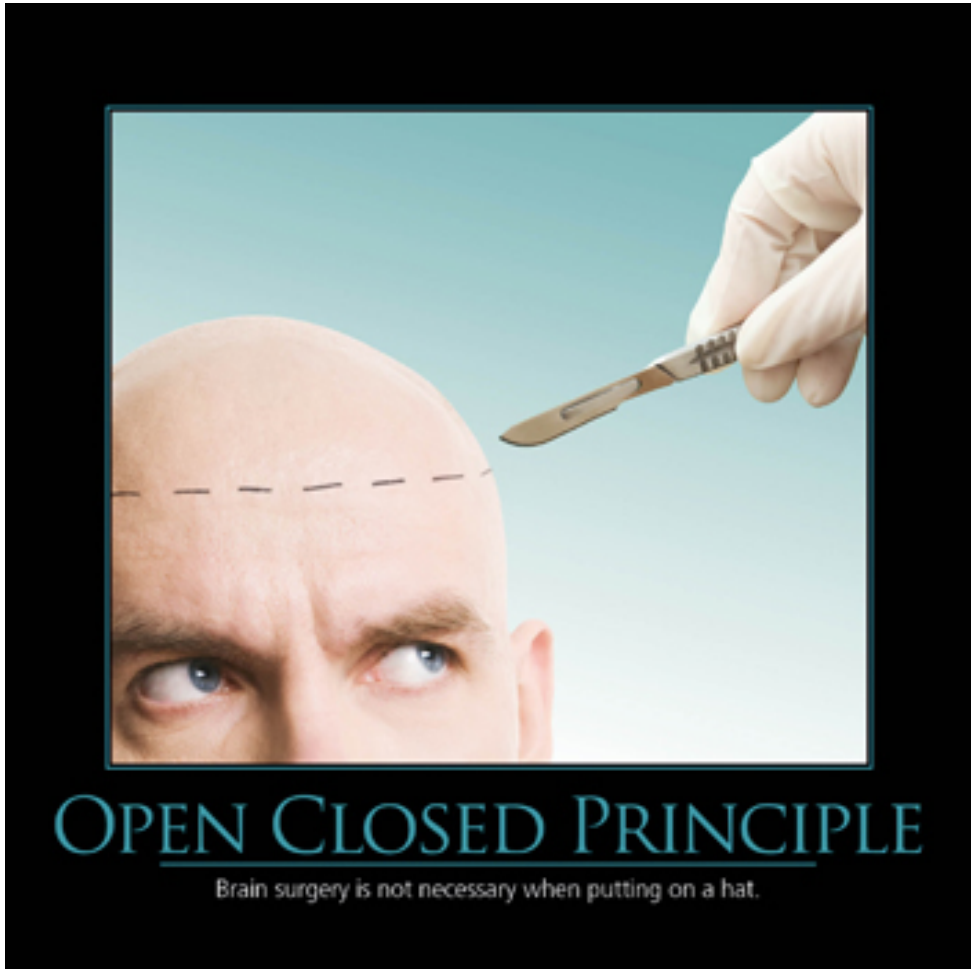
SOLID

Software Development is not a Jenga game

S.O.L.I.D Design Principles

- SRP – The **S**ingle Responsibility Principle
- OCP – The **O**pen-Closed Principle
- LSP – The **L**iskov Substitution Principle
- ISP – The **I**nterface Segregation Principle
- DIP – The **D**ependency Inversion Principle

1. Open Close Principle



Open-Closed Principle (OCP)

Software entities should be open for extension, but closed for modification

B. Meyer, 1988 / quoted by R. Martin, 1996

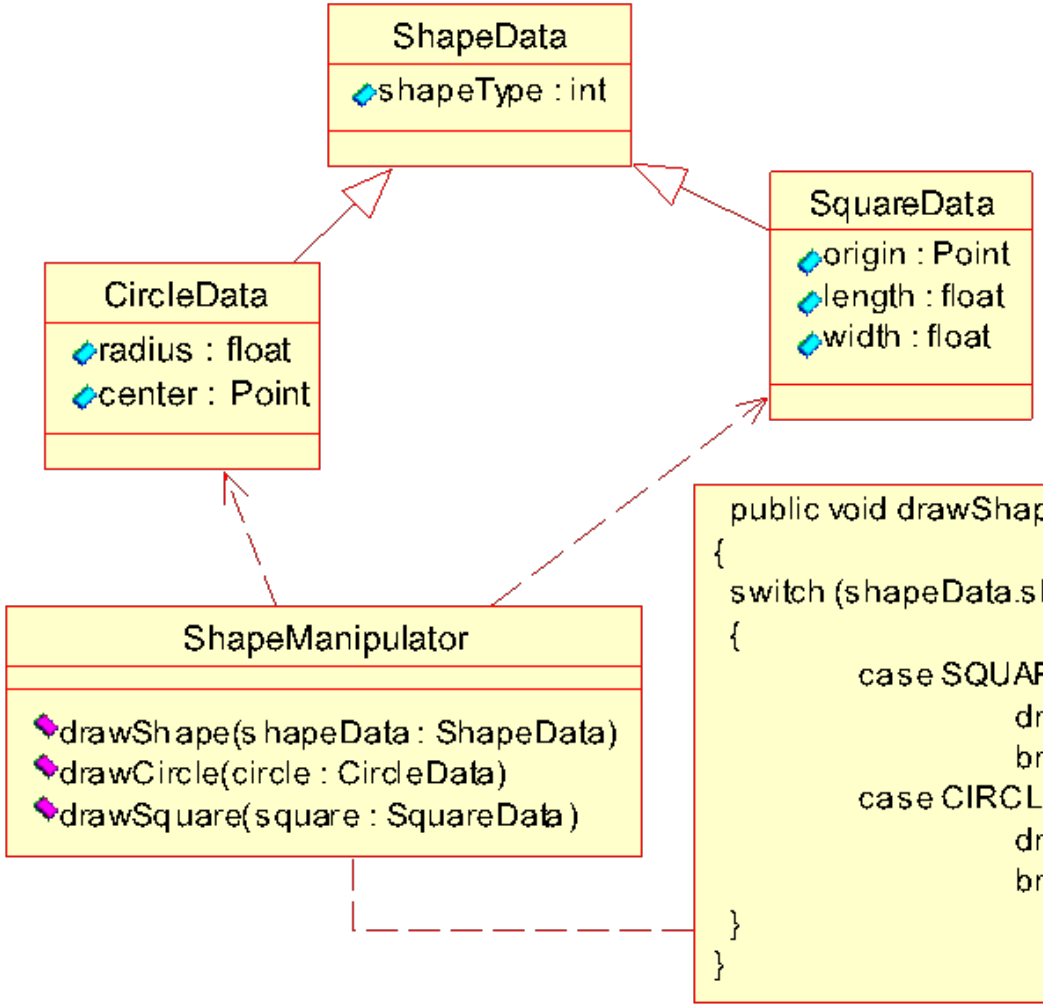
- "*Software Systems change during their life time*"
  - Both better designs and poor designs have to face the changes;
  - Good designs are stable

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

The Open-Closed Principle (OCP)

- We should write our modules so that they can be extended, without requiring them to be modified
- We want to change what the modules do, without changing the source code of the modules
- Why is it bad to change source code?
- How is OCP implemented?

The Open/Closed Principle (OCP) Example




```
public void drawShape(ShapeData shapeData)
{
 switch (shapeData.shapeType)
 {
        case SQUARE:
                drawSquare((SquareData)shapeData);
                break;
        case CIRCLE:
                drawCircle((CircleData)shapeData);
                break;
 }
}
```

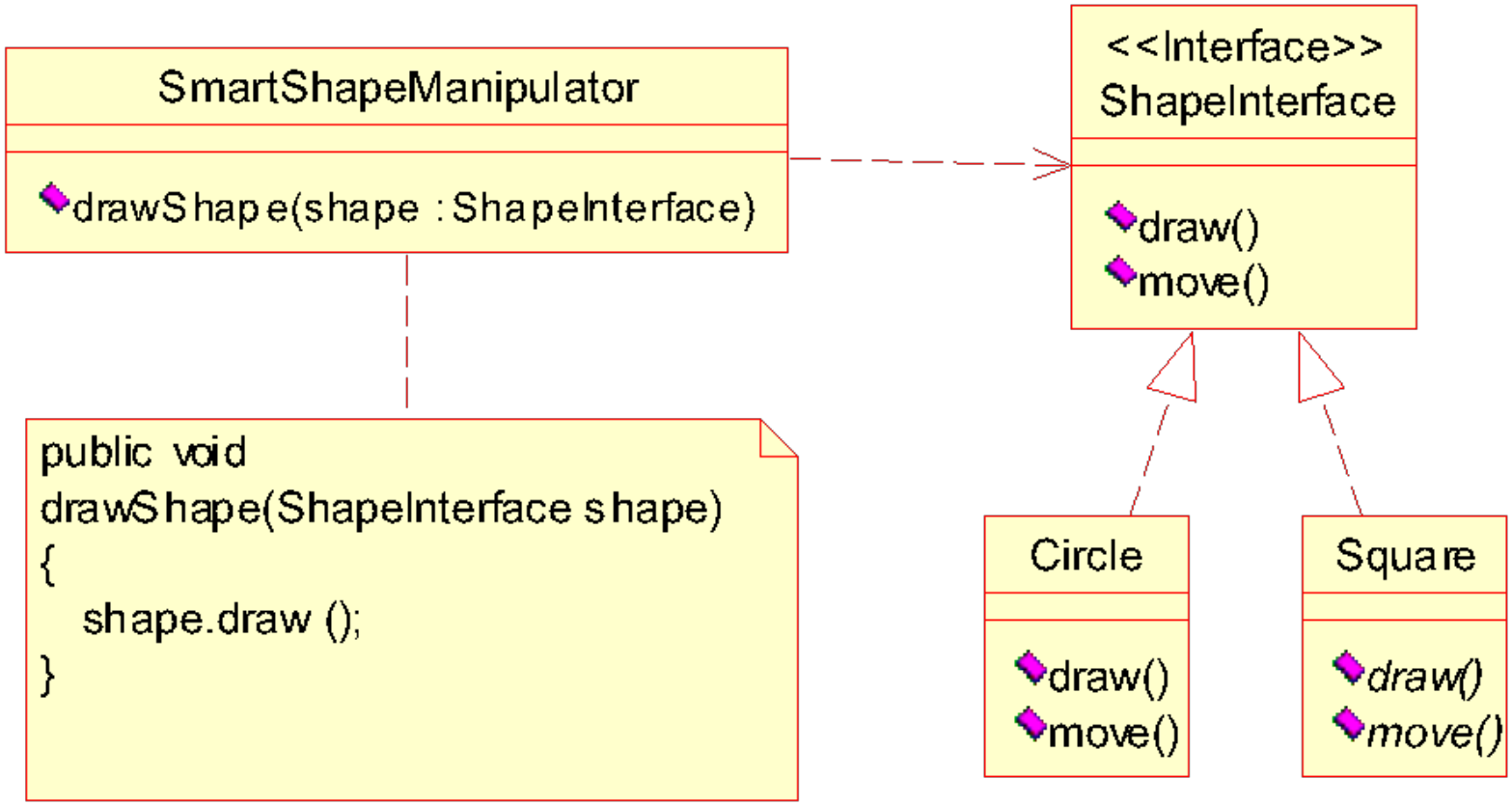
An Example of what not to do!

What is wrong with this code?

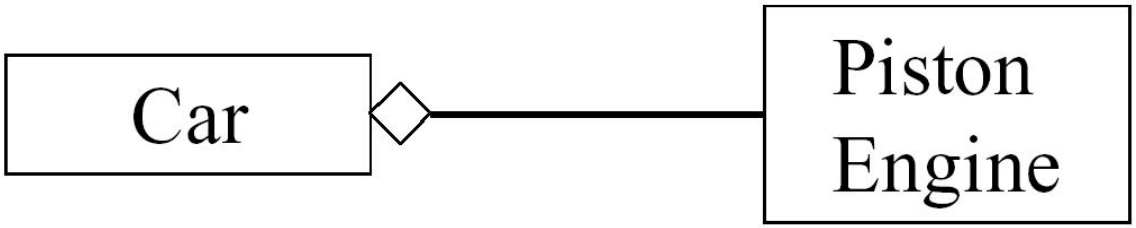
The Open/Closed Principle (OCP) Example

- The Problem: Changeability…
  - 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

The Open/Closed Principle (OCP) Example

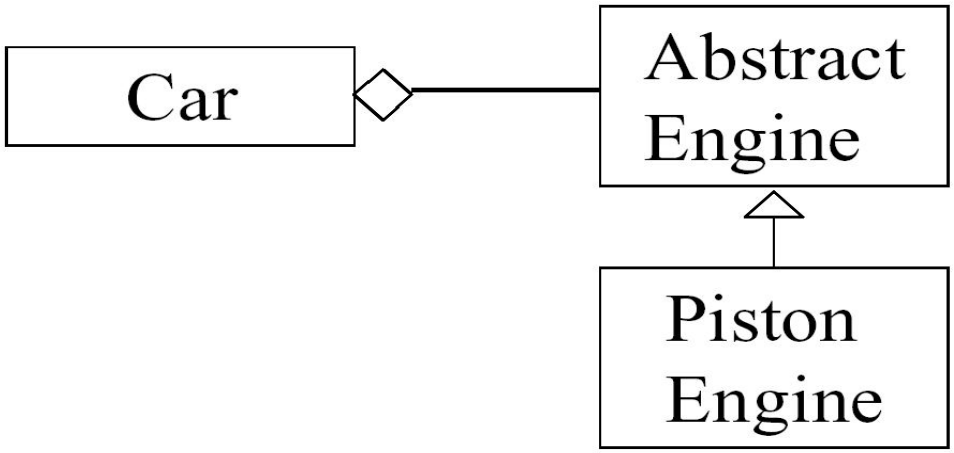


Open the Door…



- 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)

Another Example about the Car

- Different CD/Radio/MP3 players can be plugin to the car dashboard.
- …using polymorphic dependencies

OCP Heuristics

Make all object-data private

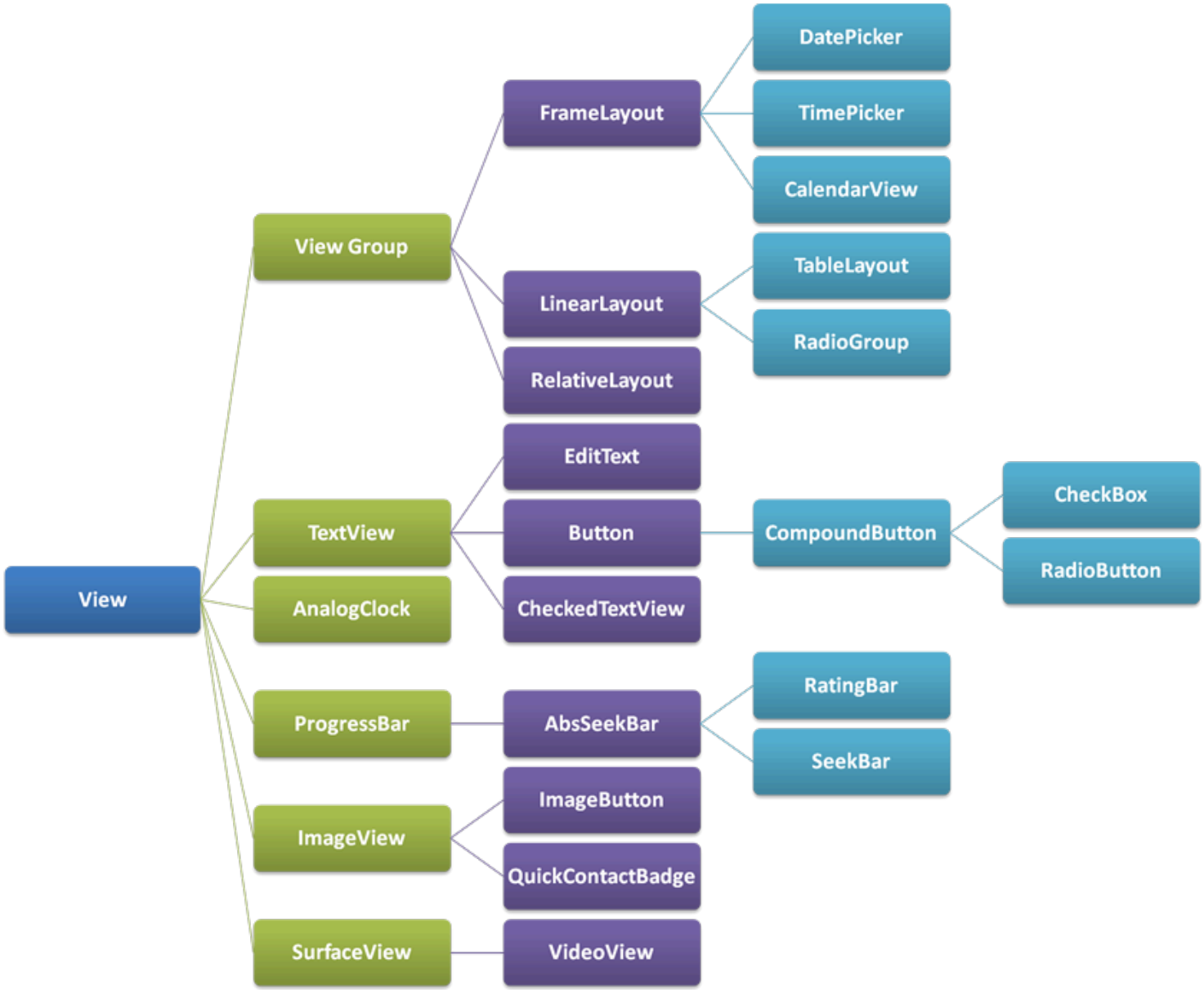
No Global Variables!

- 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: a `Time` class with open members
    - May result in inconsistent times

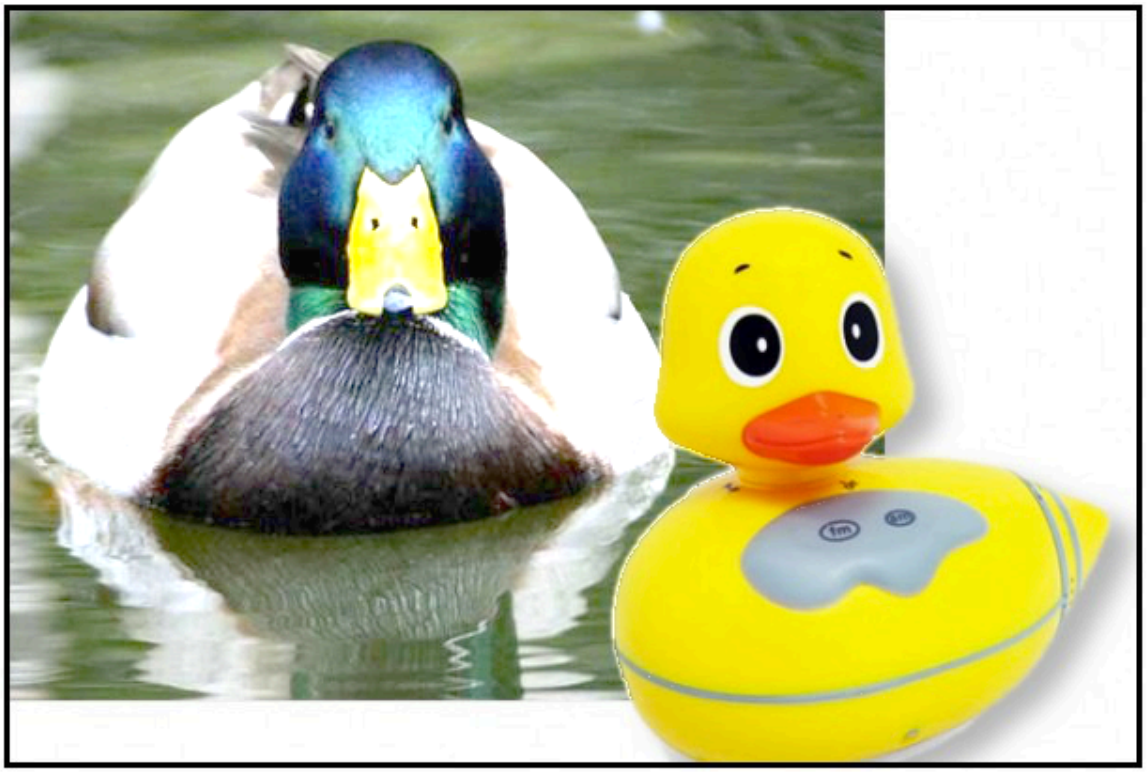
Importance of OCP

- This principle is at the heart of object oriented design. Conformance to this principle is what yields the greatest benefits claimed for object oriented technology (i.e. reusability and maintainability)
- Conformance to this principle is not achieved simply by using an object oriented programming language. Rather, it requires a dedication on the part of the designer to apply abstraction to those parts of the program that the designer feels are going to be subject to change

Example: Android Widgets



2. Liskov Substitution Principle



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

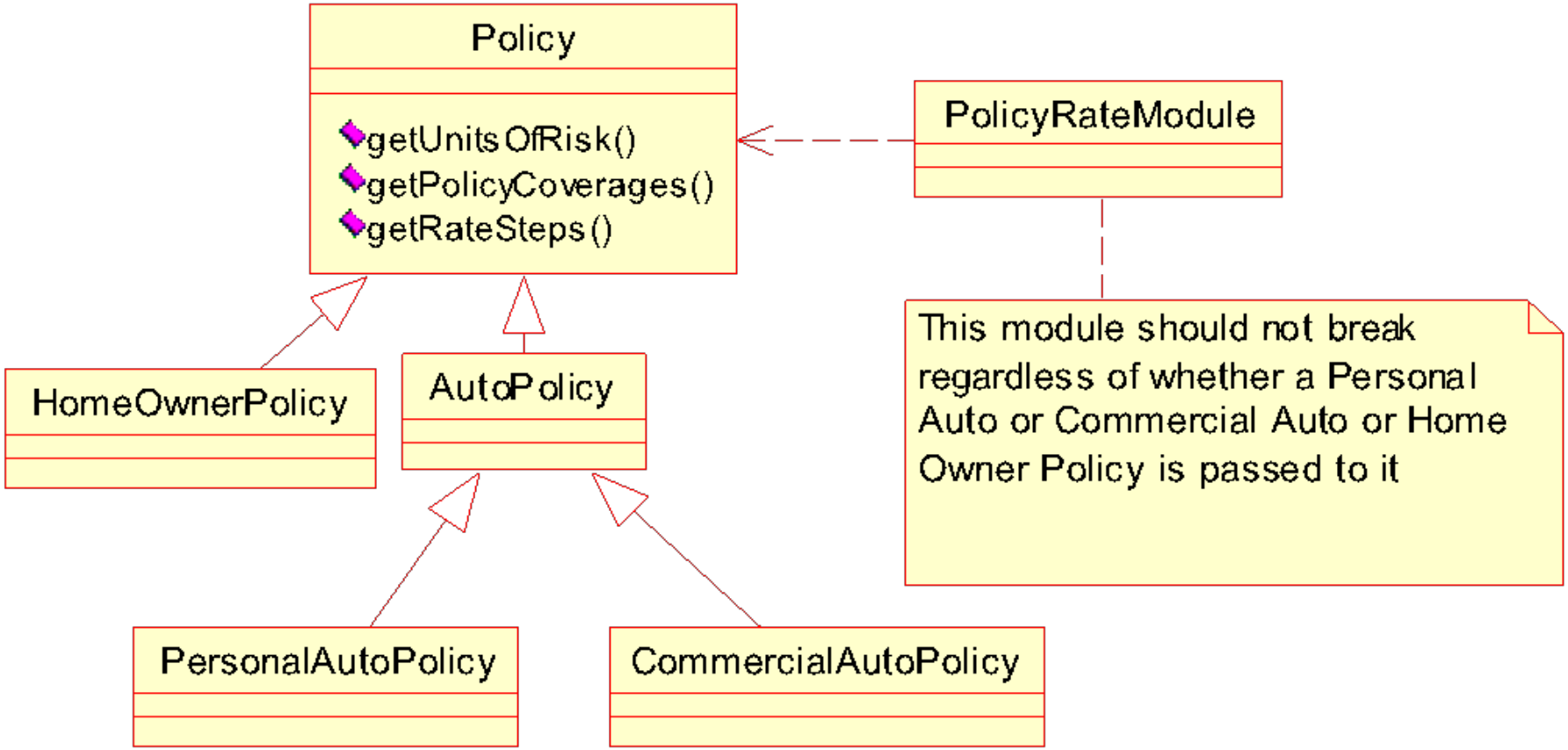
Liskov Substitution Principle (LSP)

- 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

The Liskov Substitution Principle (LCP) Example



Inheritance *Appears* Simple

```
interface Bird {                    // has beak, wings,...
  public void fly();                // Bird can fly
}

class Parrot implements Bird {    // Parrot is a bird
  public void fly() { … }      // Parrot can fly
  public 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 implements 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 an 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

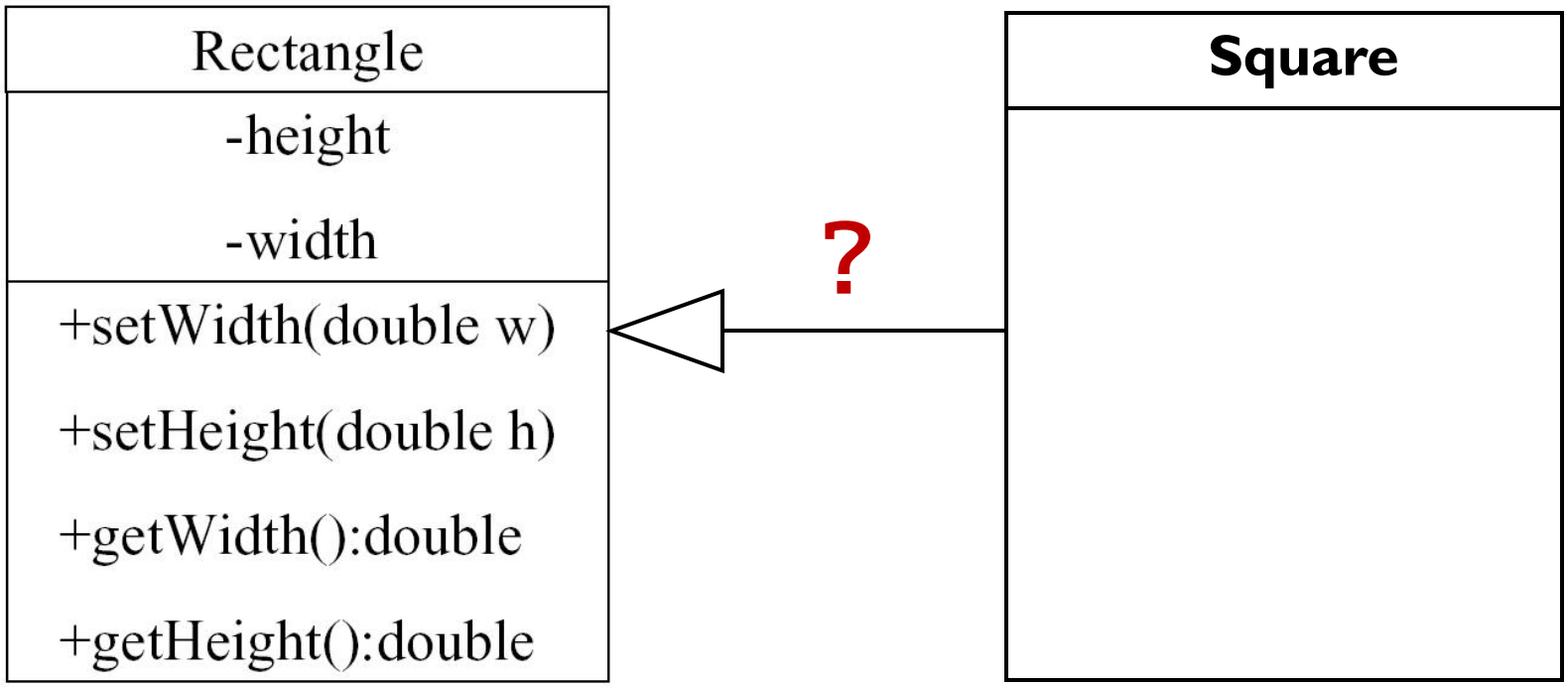
Derived class services should **require no more** and **promise no less**

```
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…

- The real problem

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

  - 20! ... Are you sure? ;-)

LSP is about Semantics and Replacement

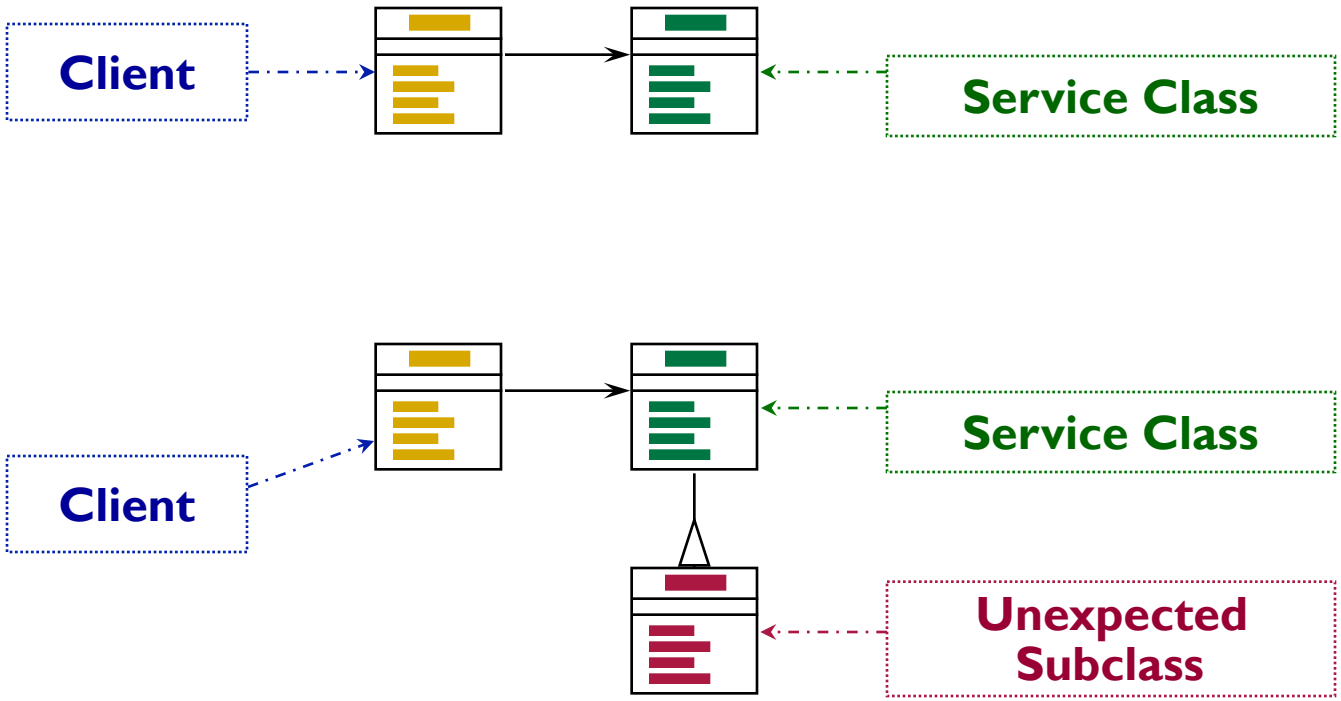
- The meaning and purpose of every method and class must be clearly documented
  - Lack of user understanding will induce violations of LSP
  - In previous example, we have intuition about squares/rectangles, but this is not the case in most other domains

- 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

- Violations of LSP are latent violations of OCP

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

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

- NOP = a method that does nothing
- Solution: Extract Common Base-Class
  - If both initial and derived classes have different behaviors
  - For `Penguins` →
    - `Birds`, `FlyingBirds`, `Penguins`

3. Single Responsibility Principle

SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

Can’t you do anything right?

What's the Issue?

```
Public class Customer {
    private String name;
    private String address;

    public void addCustomer(Customer c) {
        // database code goes here
    }

    public void generateReport(Customer c) {
        // set report formatting
    }
}
```

What does the following code do?

```
Public class Customer {
    private String name;
    private String address;

    public void addCustomer(Customer c) {
        // database code goes here
    }

    public void generateReport(Customer c) {
        // set report formatting
    }
}
```

Responsibility 1

Responsibility 2

Every time one gets changed there is a chance that the other also gets changed because both are staying in the same home and both have same parent. We can't control everything. So a single change leads to double testing (or maybe more).

OVERLOAD Kills

What is SRP?

> Every software module should have only one reason to change
>
> **R. Martin**

- Software Module – Class, Function, etc.
- Reason to Change – Responsibility

Solution which will not violate SRP

```
Public class Customer {
    private String name;
    private String address;
    // setter and getter methods
}

public class CustomerDB {
    public void addCustomer(Customer c) {
        // database login goes here
    }
}

public class CustomerReport {
    public void generateReport(Customer c) {
        // set report formatting
    }
}
```

Can a single class has multiple methods?

- YES!
- The class responsibility is described at a higher level, or is related to the context

```
public class CustomerDB {
    public void addCustomer(Customer c) {
        // database logic goes here
    }
    public Customer getCustomer(String name) {
         // database logic goes here
    }
}

public class CustomerReport {
    public void generateReport(Customer c) {
        // set report formatting
    }
    public void persistReport(Custerom c) {
        // save report in disk
    }
}
```

Methods should follow SRP, too

```
//Method with multiple responsibilities - violating SRP
public void Insert(Employee e)
{
    string StrConnectionString = "";
    SqlConnection objCon = new SqlConnection(StrConnectionString);
    SqlParameter[] SomeParameters=null;//Create Parameter array from values
    SqlCommand objCommand = new SqlCommand("InertQuery", objCon);
    objCommand.Parameters.AddRange(SomeParameters);
    ObjCommand.ExecuteNonQuery();
}
```

It does too many things:

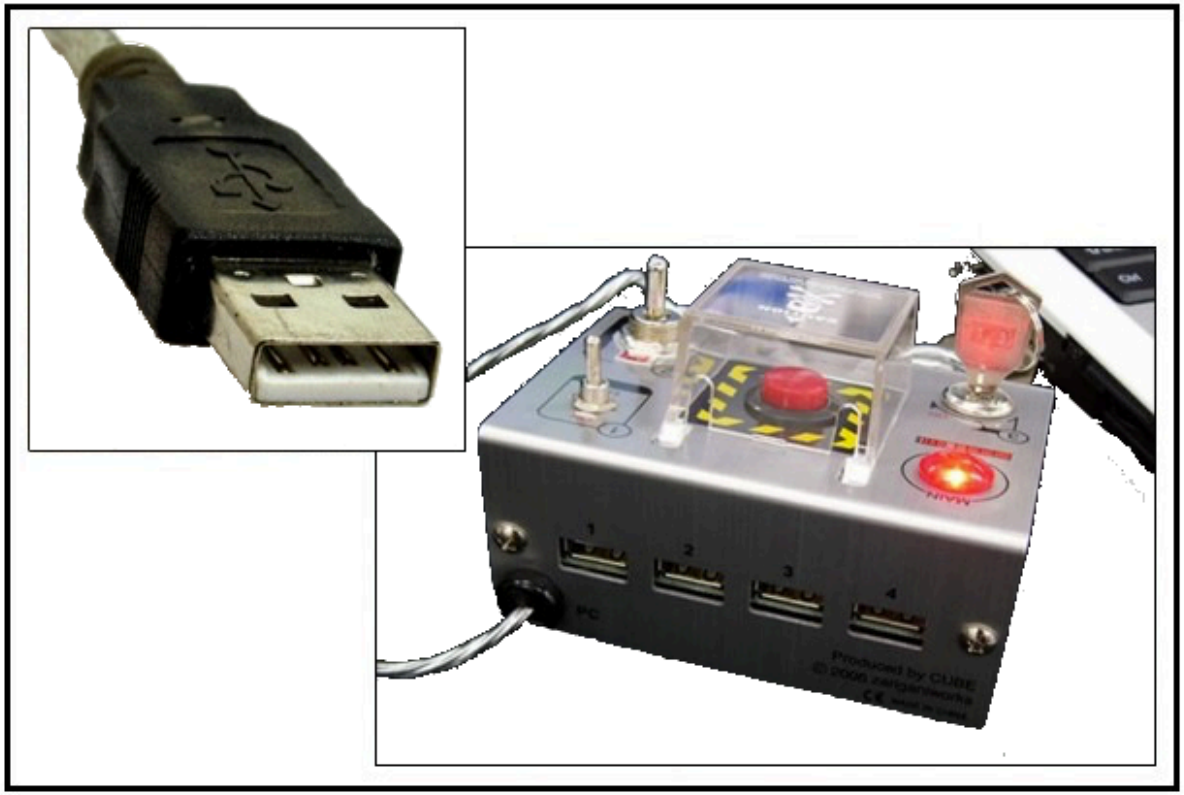
1. Build database connection
2. Form parameters
3. Generate command

Methods should follow SRP, too

```
//Method with single responsibility - follow SRP
public void Insert(Employee e)
{
    SqlConnection objCon = GetConnection();
    SqlParameter[] SomeParameters=GetParameters();
    SqlCommand ObjCommand = GetCommand(objCon,"InertQuery",SomeParameters);
    ObjCommand.ExecuteNonQuery();
}

private SqlCommand GetCommand(SqlConnection objCon, string InsertQuery, SqlParameter[]
SomeParameters)
{
    SqlCommand objCommand = new SqlCommand(InsertQuery, objCon);
    objCommand.Parameters.AddRange(SomeParameters);
    return objCommand;
}

private SqlParameter[] GetParaeters()
{
    //Create Paramter array from values
}

private SqlConnection GetConnection()
{
    string StrConnectionString = "";
    return new SqlConnection(StrConnectionString);
}
```

Rule: Keep It Simple Stupid

4. Interface Segregation Principle



INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

Really World Comparison

Report Management System

```
public class EmployeeUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
    }
}
public class ManagerUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport ();
        objBal.GenerateProjectSchedule ();
    }
}
public class AdminUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport();
        objBal.GenerateProjectSchedule();
        objBal.GenerateProfitReport();
    }
}
```

```
public interface IReportBAL
{
    void GeneratePFReport();
    void GenerateESICReport();

    void GenerateResourcePerformanceReport();
    void GenerateProjectSchedule();

    void GenerateProfitReport();
}
```

IReportBAL is used by all the 3 components:

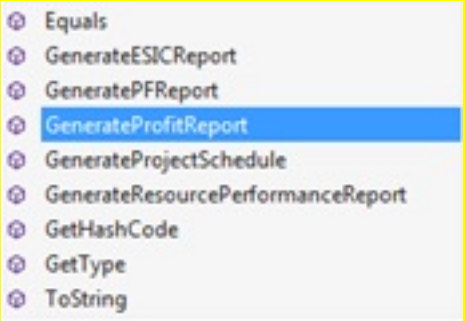
1. EmployeeUI
2. ManagerUI
3. AdminUI

The Problem

```
public class EmployeeUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
    }
}
public class ManagerUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceRep
        objBal.GenerateProjectSchedule ();
    }
}
public class AdminUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport();
        objBal.GenerateProjectSchedule();
        objBal.GenerateProfitReport();
    }
}
```

```
public interface IReportBAL
{
    void GeneratePFReport();
    void GenerateESICReport();

    void GenerateResourcePerformanceReport();
    void GenerateProjectSchedule();

    void GenerateProfitReport();
}
```

Everytime "objBal" is typed, all the methods will be shown, which is not always necessary:



What is ISP?

> Clients should not be forced to depend upon interfaces that they do not use.
>
> **R. Martin**

- Keep the interfaces concise and small

Interface Segregation

```
public interface IEmployeeReportBAL
{
    void GeneratePFReport();
    void GenerateESICReport();
}
public interface IManagerReportBAL : IEmployeeReportBAL
{
    void GenerateResourcePerformanceReport();
    void GenerateProjectSchedule();
}
public interface IAdminReportBAL : IManagerReportBAL
{
    void GenerateProfitReport();
}
public class ReportBAL : IAdminReportBAL
{
    public void GeneratePFReport()
    {/*...............*/}

    public void GenerateESICReport()
    {/*...............*/}

    public void GenerateResourcePerformanceReport()
    {/*...............*/}

    public void GenerateProjectSchedule()
    {/*...............*/}

    public void GenerateProfitReport()
    {/*...............*/}
}
```

Interface Segregation

- Equals
- GenerateESICReport
- GeneratePFReport
- GetHashCode
- GetType
- ToString

```
public class EmployeeUI
{
    public void DisplayUI()
    {
        IEmployeeReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
    }
}
```

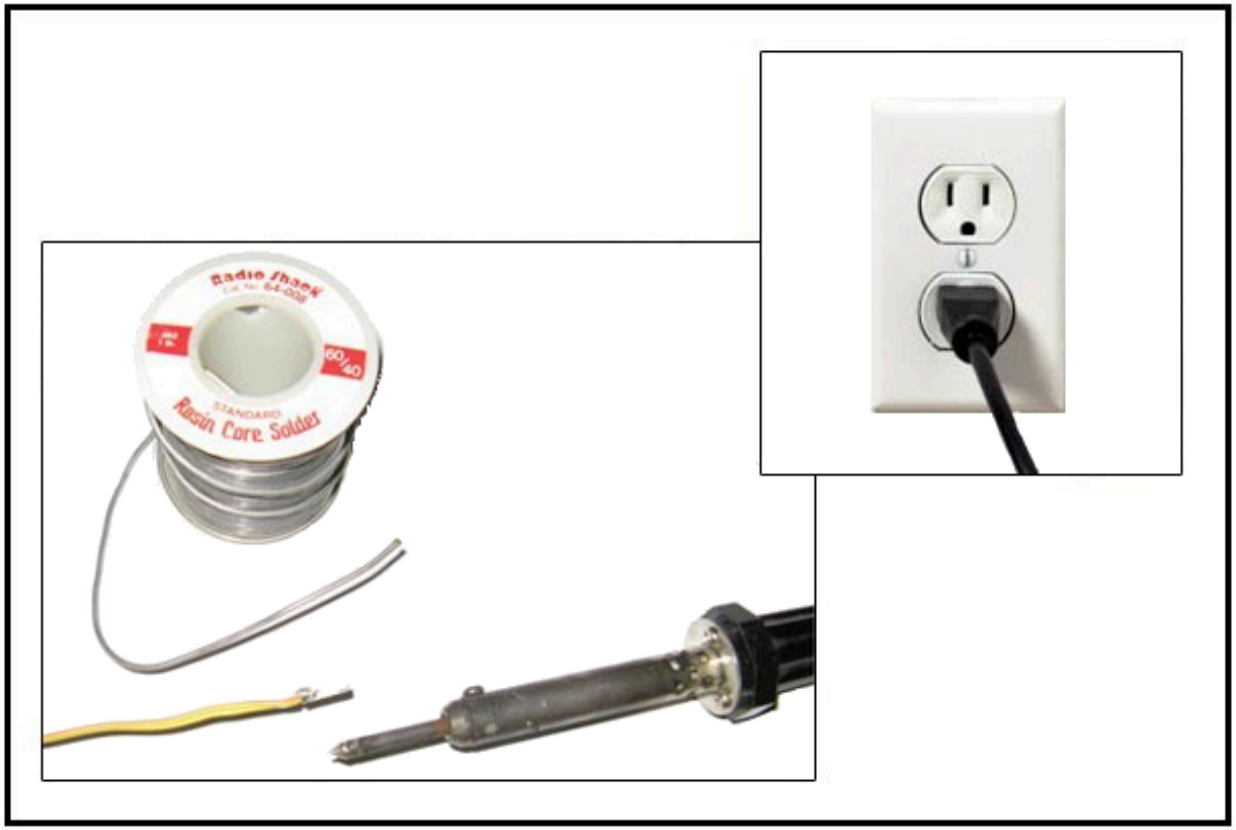
- Equals
- GenerateESICReport
- GeneratePFReport
- GenerateProjectSchedule
- GenerateResourcePerformanceReport
- GetHashCode
- GetType
- ToString

```
public class ManagerUI
{
    public void DisplayUI()
    {
        IManagerReportBAL  objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport ();
        objBal.GenerateProjectSchedule ();
    }
}
```

- Equals
- GenerateESICReport
- GeneratePFReport
- GenerateProfitReport
- GenerateProjectSchedule
- GenerateResourcePerformanceReport
- GetHashCode
- GetType
- ToString

```
public class AdminUI
{
    public void DisplayUI()
    {
        IAdminReportBAL  objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport();
        objBal.GenerateProjectSchedule();
        objBal.GenerateProfitReport();
    }
```

5. Dependency Inversion Principle



DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Dependency Inversion Principle

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

II. Abstractions should not depend on details

Details should depend on abstractions

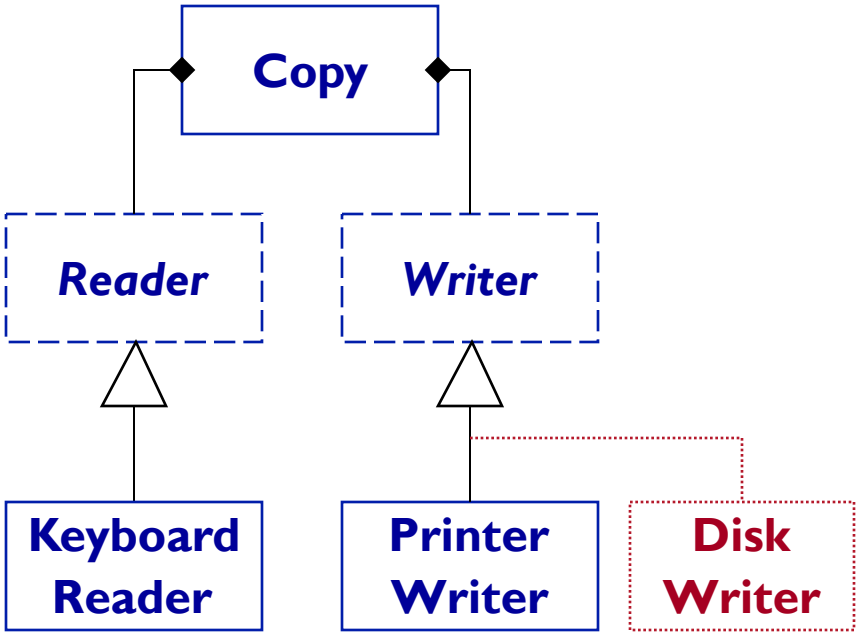
R. Martin, 1996

- 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 higher abstractions

Dependency Inversion Principle

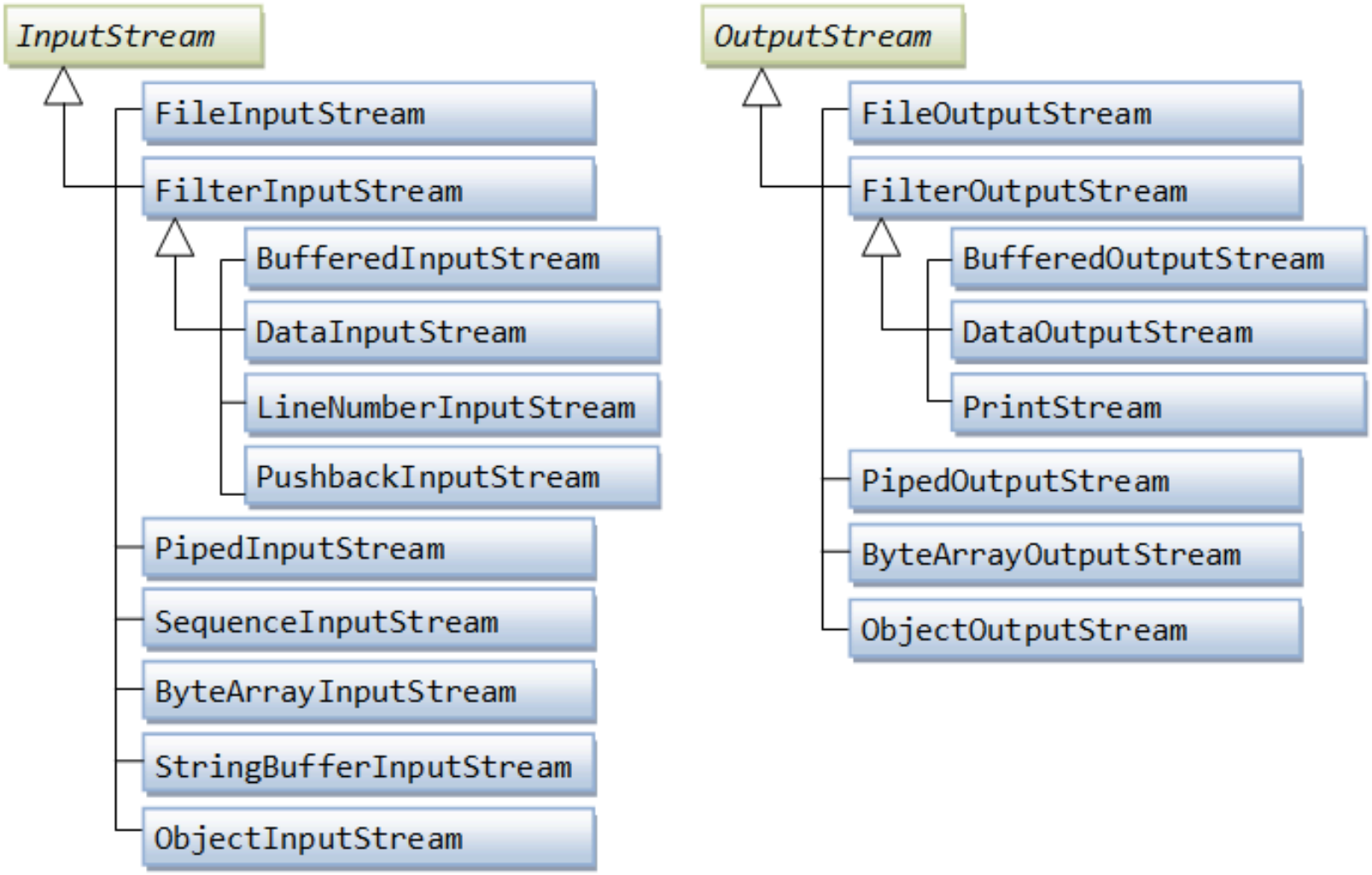
- Dependency Inversion is the strategy of depending upon interfaces or abstract functions and classes, rather than upon concrete functions and classes
- Every dependency in the design should target an interface, or an abstract class. No dependency should target a concrete class

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);
}
```

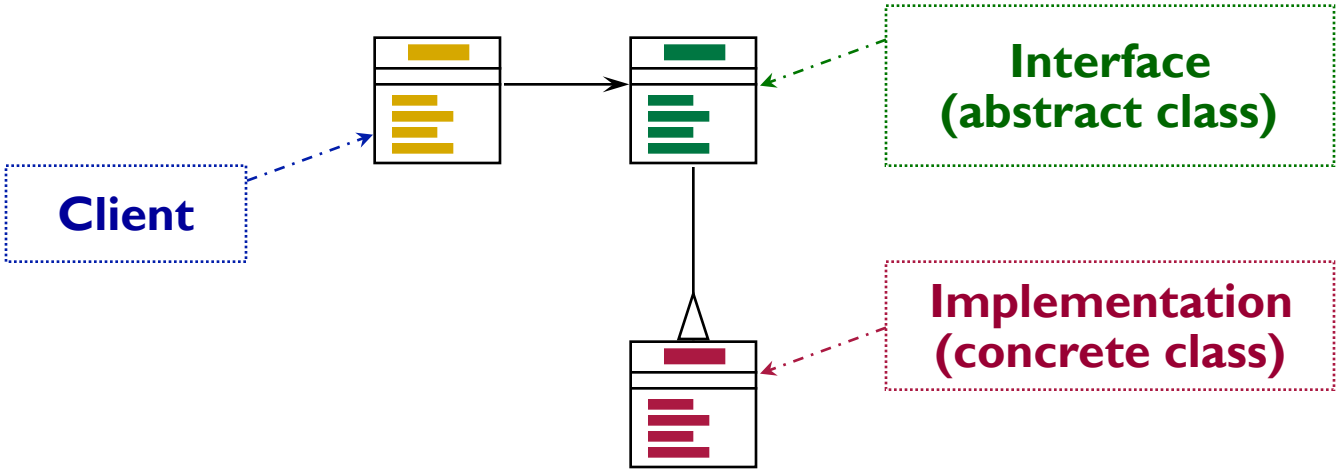
Java I/O


InputStream
FileInputStream
FilterInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
PushbackInputStream
PipedInputStream
SequenceInputStream
ByteArrayInputStream
StringBufferInputStream
ObjectInputStream
OutputStream
FileOutputStream
FilterOutputStream
BufferedOutputStream
DataOutputStream
PrintStream
PipedOutputStream
ByteArrayOutputStream
ObjectOutputStream

DIP Related Heuristic

Program to interface,
Not implementation!

- Use inheritance to avoid direct bindings to classes:



Design to an Interface

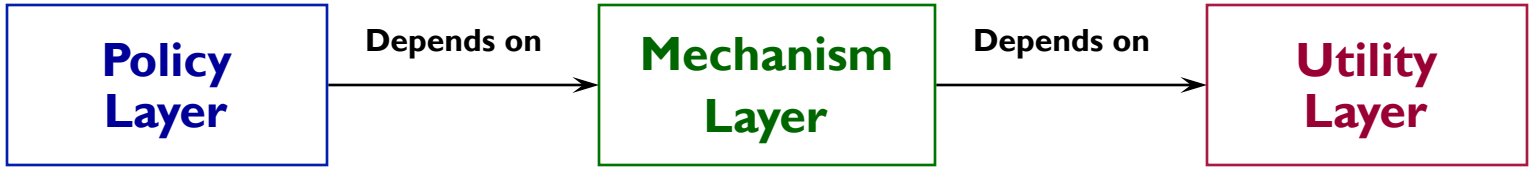
- Abstract classes/interfaces:
  - Tend to change much less frequently

- Exceptions
  - Some classes are very unlikely to change;
    - Therefore, little benefit to inserting abstraction layer
    - e.g., String class
  - In cases like this you can use concrete class directly
    - As in Java or C++

DIP Related Heuristic

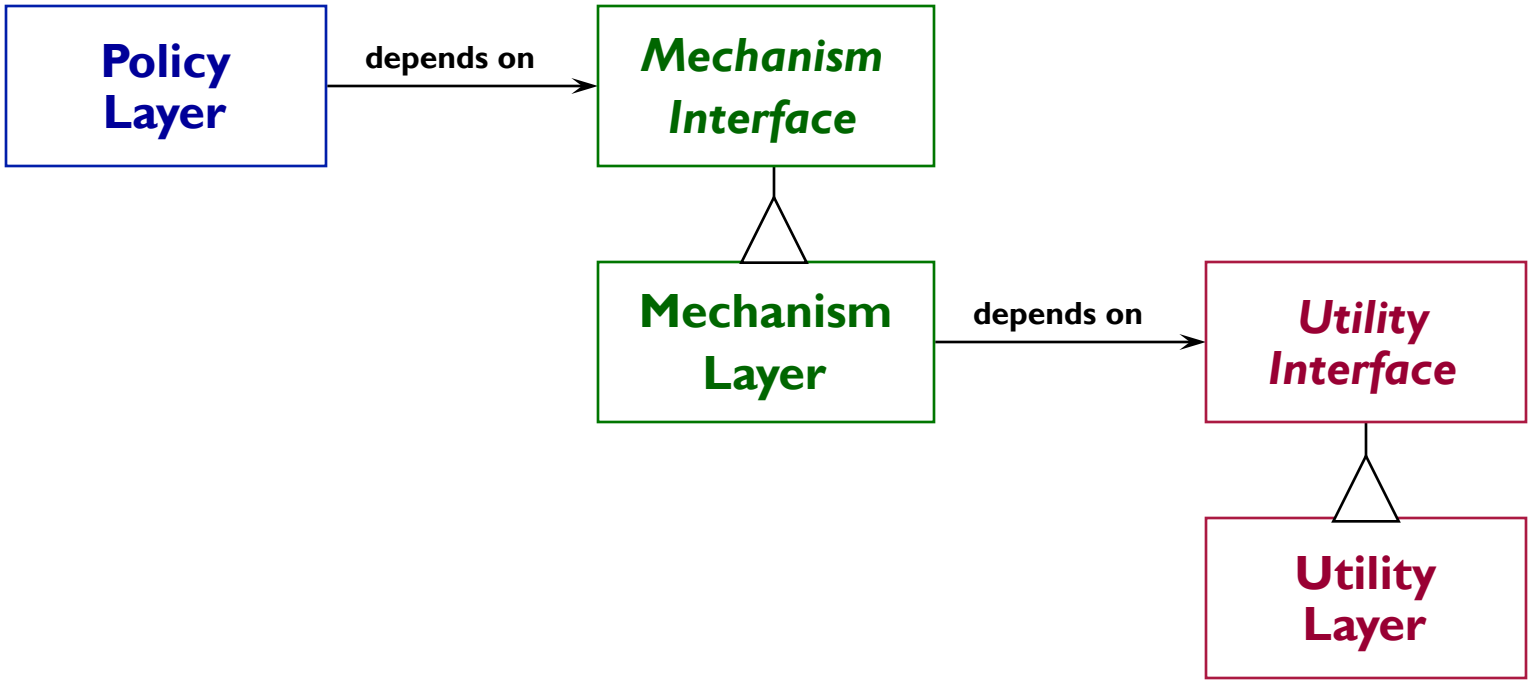
Avoid Transitive Dependencies

- 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 in Action – Google Guice

- Google Guice Demo
- https://github.com/google/guice

My Social App

- Integrate different social apps
- Send messages
- Get news feed

DIP in Action – Android Dagger

- Square Dagger Demo
- http://square.github.io/dagger/