

# 17.10 Java Beans

- Java beans are a framework for creating components in Java.
- AWT and Swing packages are built within this framework
- Made to fit in with graphic development environments such as Jbuilder and Forte
- An extension is Java Enterprise Beans

# 17.11 Bean Rules

- Code must be written correctly
- Access methods must begin with **get**
- Mutation methods must begin with **set**
- Code must support event handling
- Objects must be *persistent*, i.e. implement serializable

# 17.12 Java Beans: Development Environments

- Introspection used to assess what a bean can do (which methods it has)
- There is also an interface **BeanInfo** to give development tools more information
- Tools can change a bean's properties, e.g. background color in a window if the window is regarded as a bean.

# 17.13 .NET

- Microsoft calls components **assemblies**
- .NET is an environment similar to Java
- E.g. code is compiled to **Microsoft IL** (Intermediate Language)
- Executed code is handled by the **CLR** (Common Language Runtime)
- Programming language **C#** is an OO language that borrows from C++
- Replaces COM (Component Object Model)
- Avoids registration and the DLL Hell!

# 18. Security and Sandboxes

- JDK version 1.0
- Applets always executed in a sandbox
- Sandbox contains access to hardware

# Security (Cont.)

- JDK 1.1
- Applets delivered in signed JAR files

# Security (Cont.)

- JDK 1.2
- Applets and programs have an individual security policy which can be modified by `policytool`

# 19. Principles of O-O Design

- How to avoid common mistakes and develop a good architecture that supports change
- Developed by R.C. Martin, B. Meyer, B. Liskov et al.
- **A suite of 11 principles!**
- 5 principles of class design
- 3 principles of package cohesion
- 3 principles of package coupling

# 19.1. The 5 Principles of Class Design

- Let's look at some principles of class design which should be satisfied by any modular decomposition technique.

# 1. OCP Open-Closed Principle

- Bertrand Meyer, *Object-Oriented Software Construction*, 1988, p.23.
- **Open module:** behaviour can be extended
- **Closed module:** what exists is available for use by other modules through interface.
- Good decomposition technique should allow modules to be open and closed.

- Openness is necessary, since we rarely understand a system at first.
- Closedness is necessary because we need to get a system up and running.
- Project manager ”*dreams to be able to close a project for good*”. Rarely possible.

- Inheritance is the best way to solve the problem of writing modules that are open and closed.
- In other words, (in an ideal world...) you should never need to change existing code or classes: All new functionality can be added by adding new subclasses and overriding methods, or by reusing existing code through delegation.

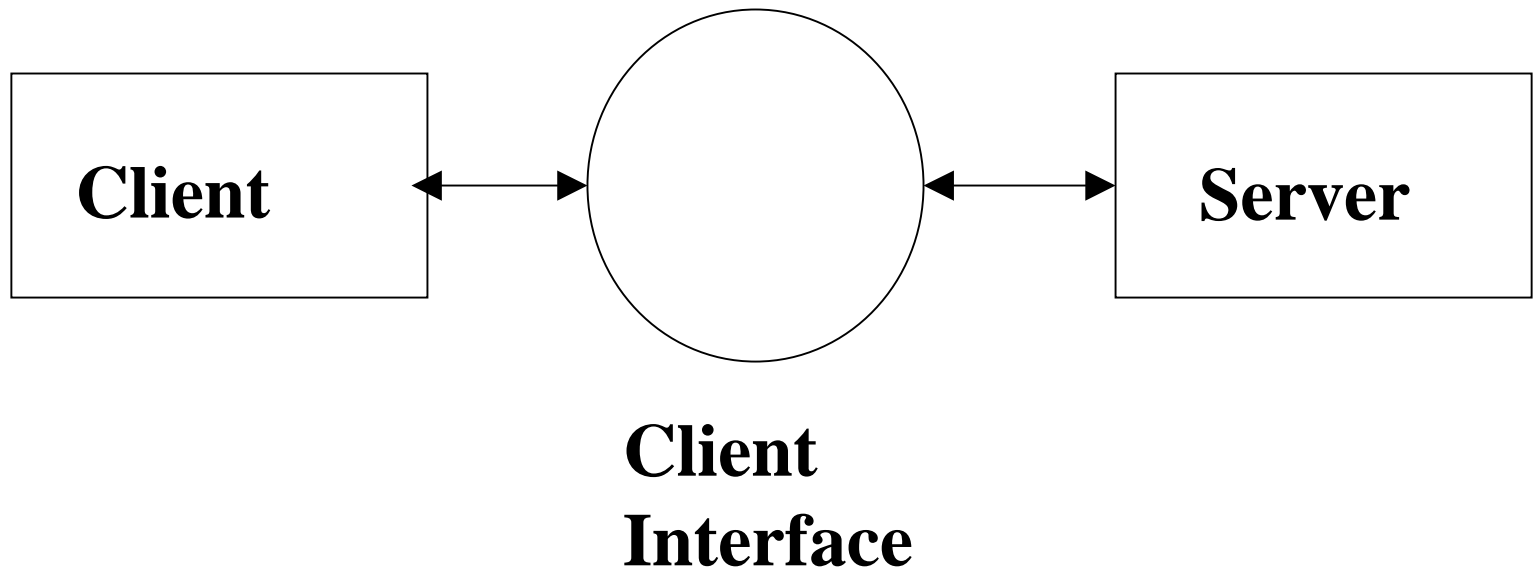
# OCP Example

- Problem: changes in the server often force changes in the client



# OCP Example

- Solution: now client can be left unchanged



## 2. SRP Single Responsibility Principle

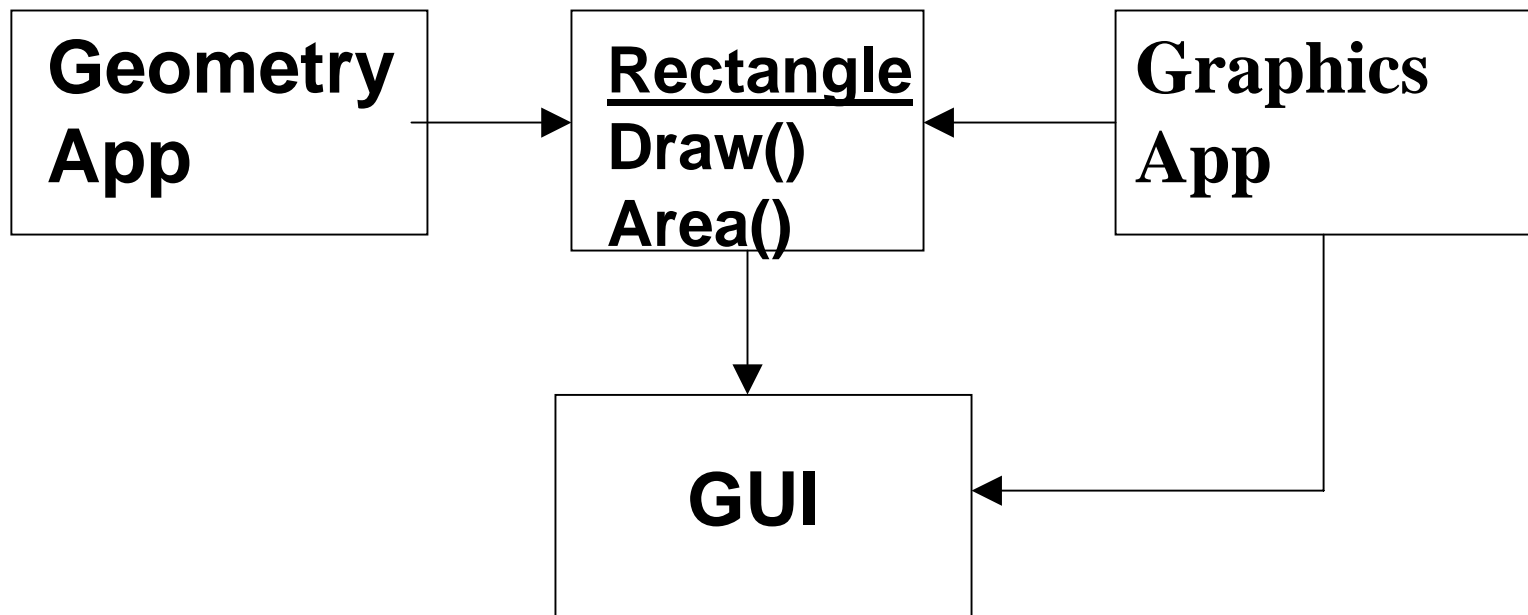
- Each class has one responsibility
- Otherwise changes in requirements that affect one responsibility also affect other responsibilities.
  - *“One of the criteria I use is to try to describe a class in 25 words or less, and not to use "and" or "or". If I can't do this, then I may actually have more than one class. “*

- A class should have one, and only one, reason to change.
- If a change to the business rules causes a class to change, then a change to the database schema, GUI, report format, or any other segment of the system should not force that class to change.
- Each responsibility should be a separate class, because each responsibility is an axis of change.

- *A class has a single responsibility: it does it all, does it well, and does it only.* Bertrand Meyer.
- Classes, interfaces, functions, etc. all become large and bloated when they're trying to do too many things. When a function has too many responsibilities, it becomes buried in deep if-then indentations ([SpecialFormatting](#)), which has a smell ([CodeSmell](#)).
- To avoid bloat and confusion, and ensure that code is truly simple (not just quick to hack out) we have to practice [CodeNormalization](#), which seems to be a variation on [OnceAndOnlyOnce](#) and also [DoTheSimplestThingThatCouldPossiblyWork](#).

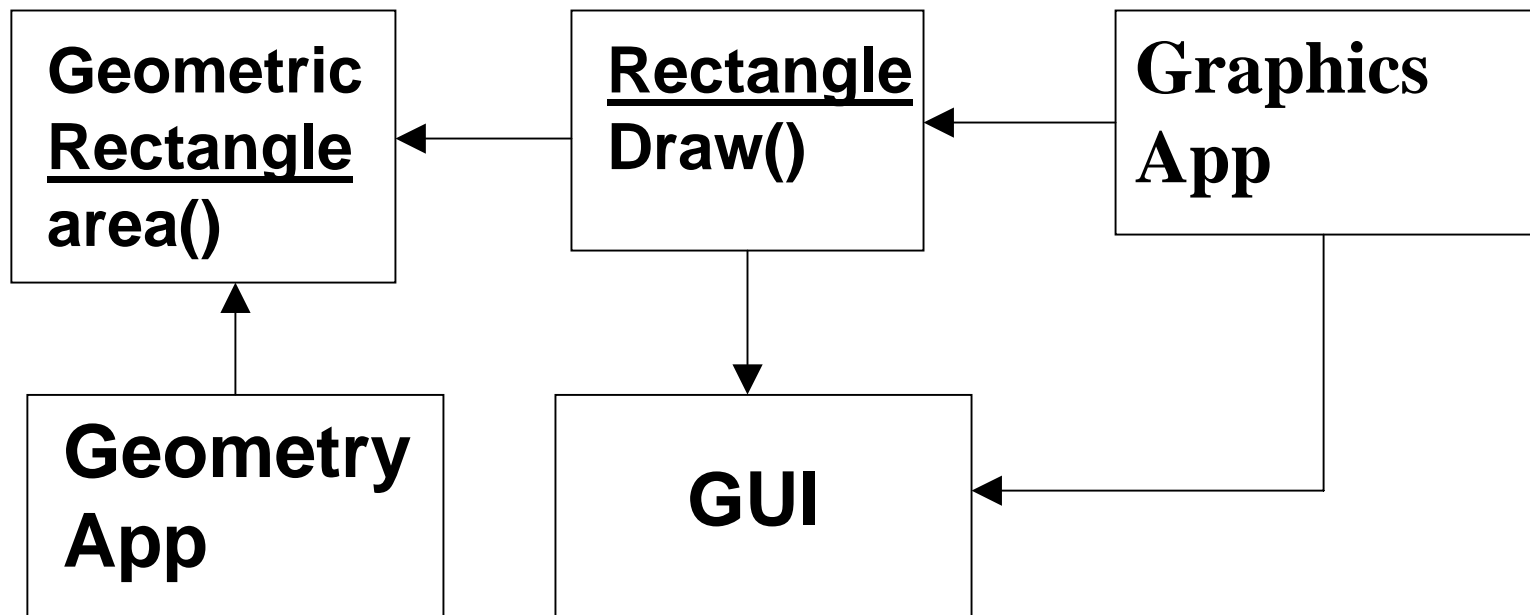
# SRP Example

- Problem: a rectangle has several responsibilities



# SRP Example

- Solution: responsibility is now divided into logical modules



# 3. LSP Liskov Substitution Principle

“If for each object  $o_1$  of type  $S$  there is an object  $o_2$  of type  $T$  such that for all programs  $P$  defined in terms of  $T$ , the behavior of  $P$  is unchanged when  $o_1$  is substituted for  $o_2$  then  $S$  is a subtype of  $T$ .”

[BarbaraLiskov](#), Data Abstraction and Hierarchy, *SIGPLAN Notices*, 23,5 (May, 1988).

# Meaning?

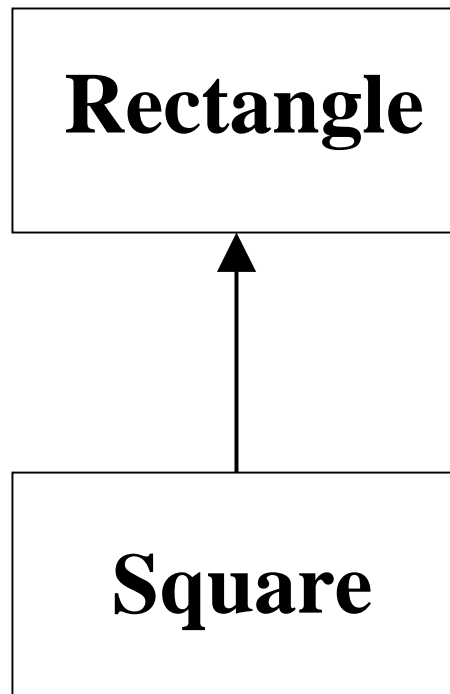
- A subclass should always be usable wherever any superclass is used.
- Means, don't use inheritance for trivial purposes, but only where there really exists an "is a" relation, so superclass is a genuine subset of subclass.

# Why?

1. Because if not, then class hierarchies would be a mess. Mess being that whenever a subclass instance was passed as parameter to any method, strange behaviour would occur.
2. Because if not, unit tests for the superclass would never succeed for the subclass.

# LSP Example

- Problem: subclass restricts attributes and/or functionality in the superclass



# LSP (Cont.)

- Typical symptoms: method calling base class has a nest of if/else statements to test which subclass during a method call
- Solution: override method causing the problem in the subclass exploiting polymorphism.
- Don't need any special cases, one kind of method call suffices.

- “I don't think that LSP is academic. Take a look at JUnit. The [TestCase](#) and [TestSuite](#) classes do not violate any expectations of their superclass (Test). I tend to think that LSP is actually a special case of a more general principle of engineering: "you can't do with less than what you need." If you are going to substitute this bit in for another bit, it had better satisfy all contextual expectations. It can do more, but it can never break the expectations of the context, or else the context must be modified. The fact is, there can be a disconnect between LSP and representational modeling. The reason is that our cognitive processes throw a lot more baggage into a concept than what we can place in an artifact that must deal with contextual expectations.” -- [MichaelFeathers](#)

# 4. DIP Dependency Inversion Principle

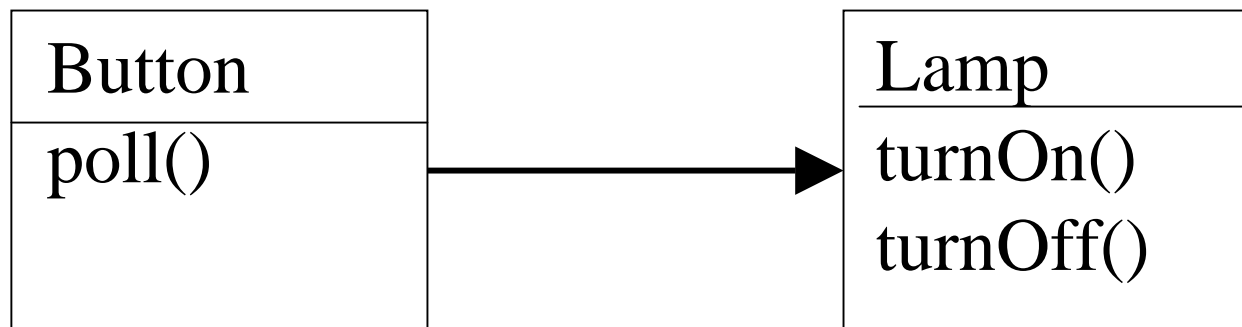
- Habits from procedural programming (C, Pascal, etc) mean that high-level modules depend on low-level modules and their details
  - High level modules should not depend upon low level modules. Both should depend upon abstractions.
  - Abstractions should not depend upon details. Details should depend upon abstractions.

- We wish to avoid designs which are:
- *Rigid* (Hard to change due to dependencies. Especially since dependencies are transitive.)
- *Fragile* (Changes cause unexpected bugs.)
- *Immobile* (Difficult to reuse due to implicit dependence on current application code.)

- "Structured" methods of the 1970's tended towards a "top-down decomposition", which encouraged high-level modules to depend on modules written at a lower level of abstraction. (More modern procedural approaches depend more on databases and less on functional decomposition at the large-scale level. Thus, they are often "flatter" now.) This principle stems from the realization that we must reverse the direction of these dependencies to avoid *rigidity, fragility* and *immobility*.

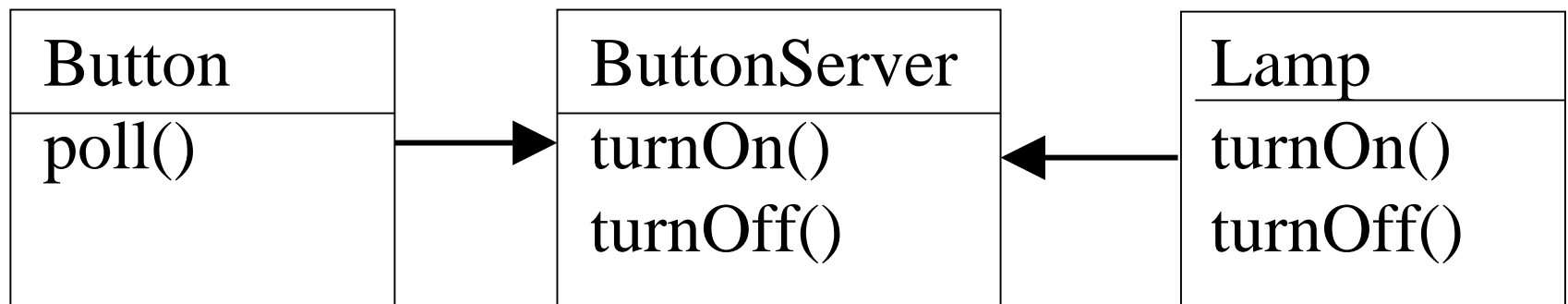
# DIP Example

- Problem: Poll() in Button depends on implementation details in Lamp
- Lamp cannot be replaced by another class



# DIP Example (Cont.)

- Solution: ButtonServer or SwitchableDevice, not AbstractLamp!



Can now replace Lamp or Button

# 5. ISP Interface Segregation

## Principle

- The dependency of one class to another one should depend on the smallest possible interface.
- Clients should not depend on methods they don't use.
- Divide up the interface or delegate responsibilities
- Delegation common in design patterns

# ISP Example

- Problem: One class  $C$  with many methods.
- Risk is that other classes  $C_i$  only use subsets of the methods. If one client  $C_i$  forces a change of  $C$  then all other clients are exposed to this change in  $C$ .

# ISP Example (Cont.)

- Solution 1: divide up the methods into several interfaces  $F_i$ , each specific to its client class  $C_i$ . Let  $C$  implement these.
- Solution 2: Let  $C$  delegate responsibilities. So  $C$  instantiates a help class which implements one of the interfaces. Help class is an intermediary between  $C$  and  $C_i$ . Changes are made to help class not  $C$ .

# 6. Summary

- OCP open-closed principle
- SRP single responsibility principle
- LSP Liskov substitution principle
- DIP dependency inversion principle
- ISP interface segregation principle