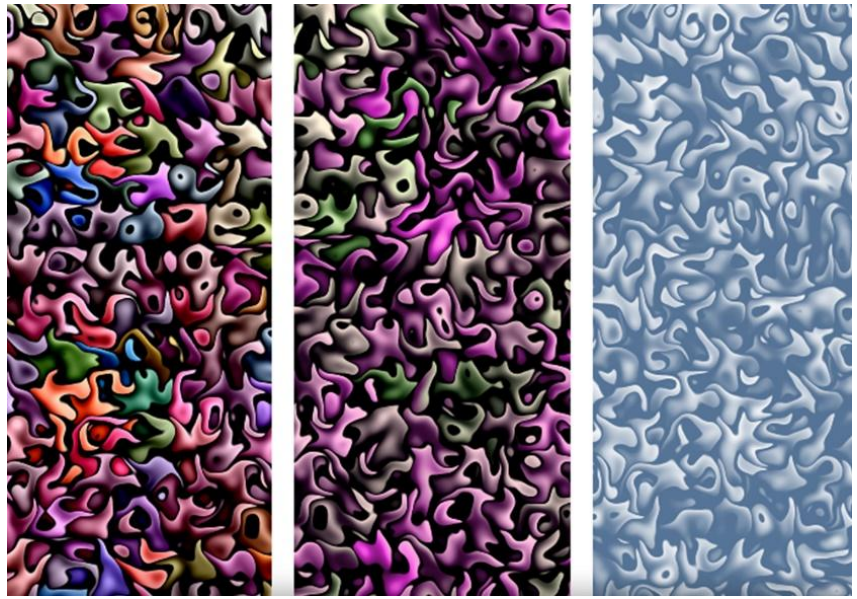
Design Patterns, SOLID

Lecture 12

19 June 2025

Slides created by
Prof Amir Tomer
tomera@cs.technion.ac.il



Topics for Today

- Software Design Patterns
- SOLID

Design patterns

Fundamental solutions to common OOP design problems

- Repeatable way to solve a problem
- How to organize classes to solve the problem

Must adapt the pattern to specific problem's context

A pattern contains

- Problem description
- Solution description
- When to use the pattern
- Outcome of the pattern
- Implementation instructions
- Implementation examples

Design Pattern Types

- Based on the Gang of Four book (E. Gamma, R. Helm, R. Johnson & J. Vlissides, Design Patterns, Addison-Wesley, 1995)

Structural Patterns

- Define complex structures

Creational Patterns

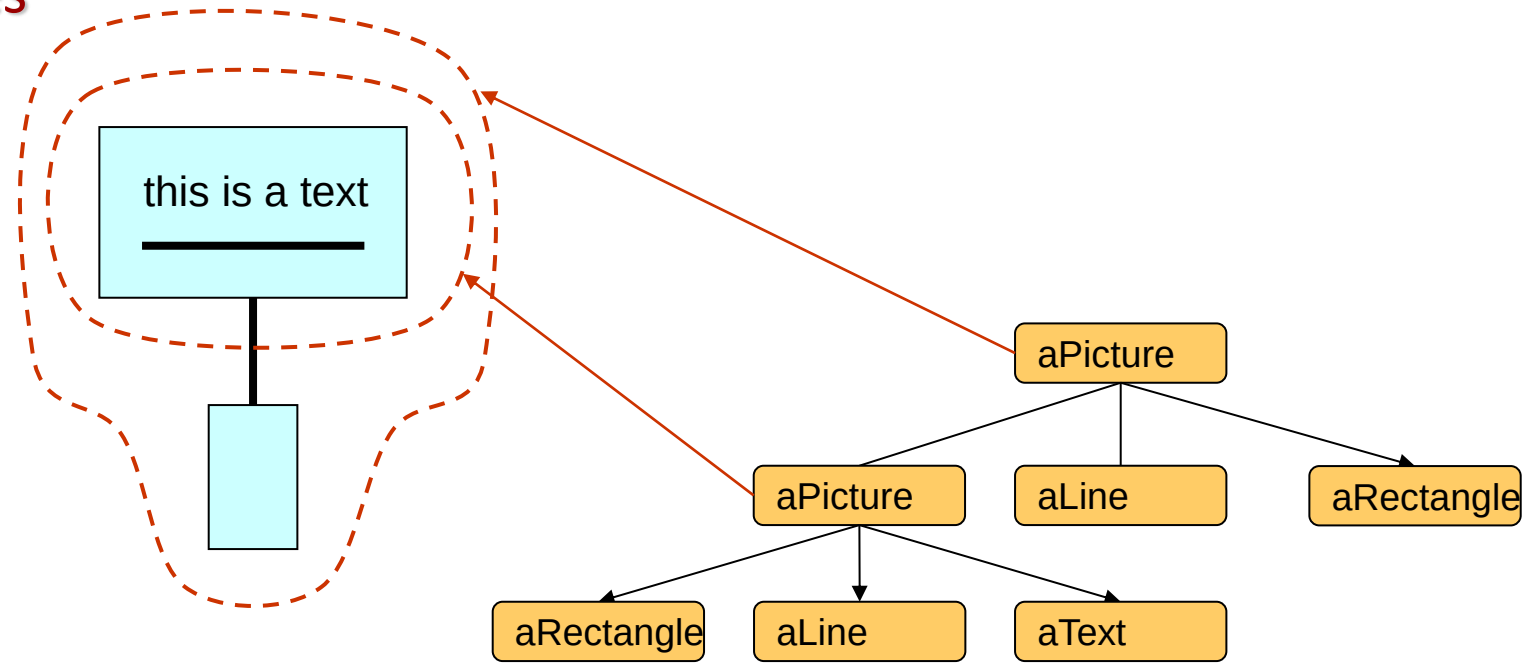
- Creating objects at run time
- Separate class declaration (specification) and instantiation (code)
- “Don’t let the user run the constructor”

Behavioral Patterns

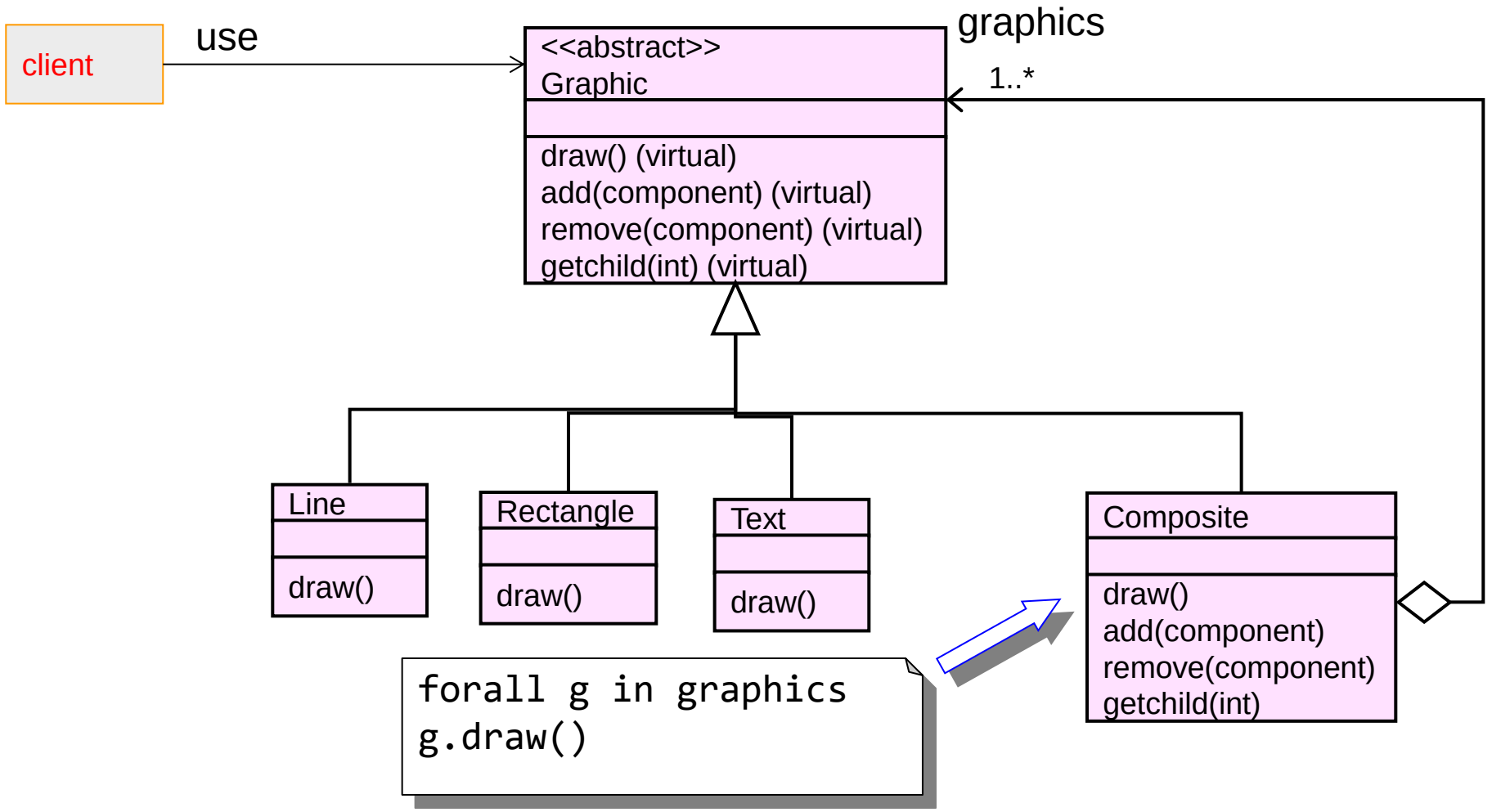
- Assign behavior to objects at runtime
- Change programs from processing objects to making objects communicate

Composite Pattern (Structural)

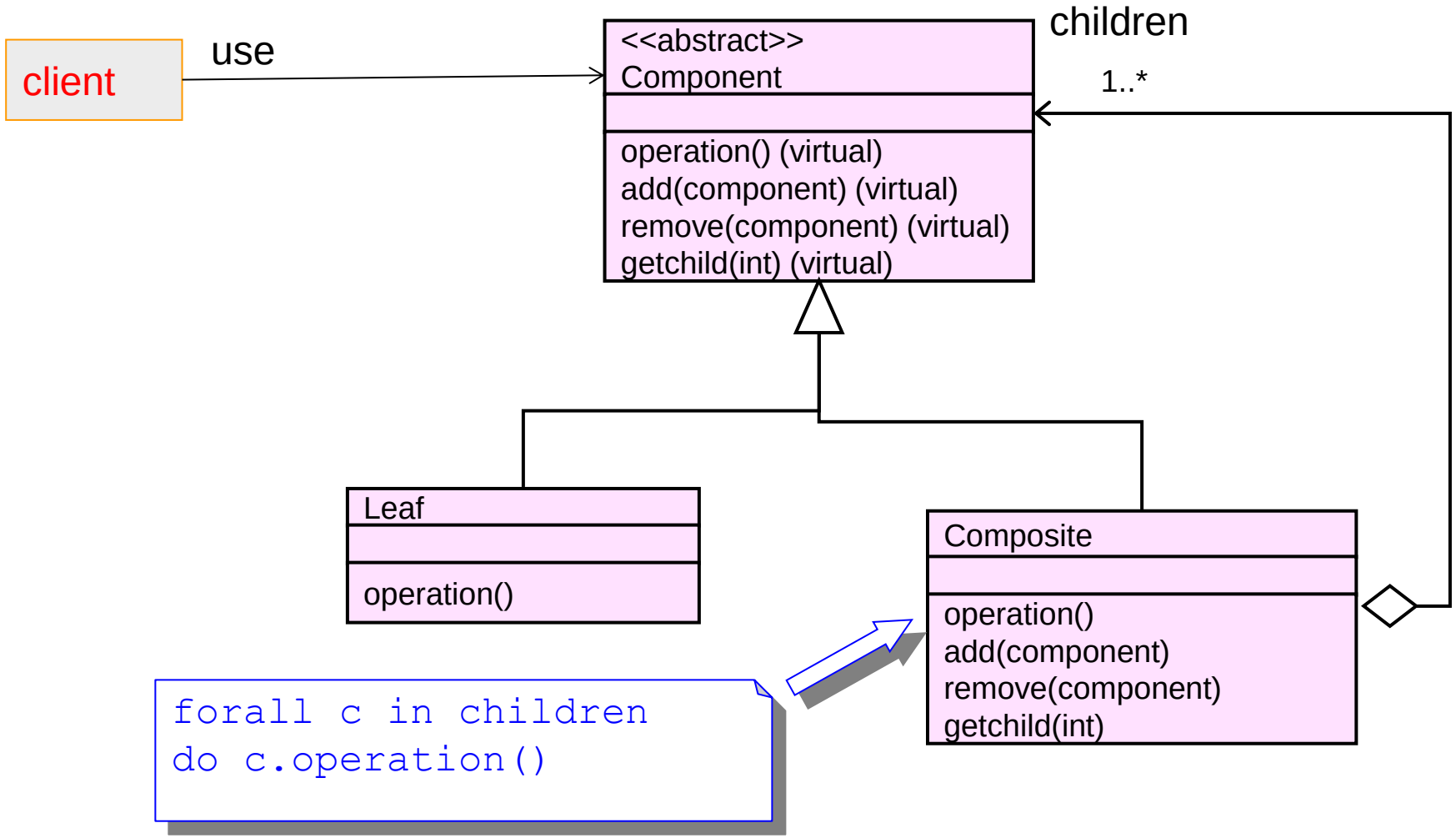
- Problem: Create recursive hierarchical structure
  - Objects can be lone or contain others of the same type
  - Want to treat a collection of objects the same as a single one
- Example: Graphical structures



Composite Pattern: Example

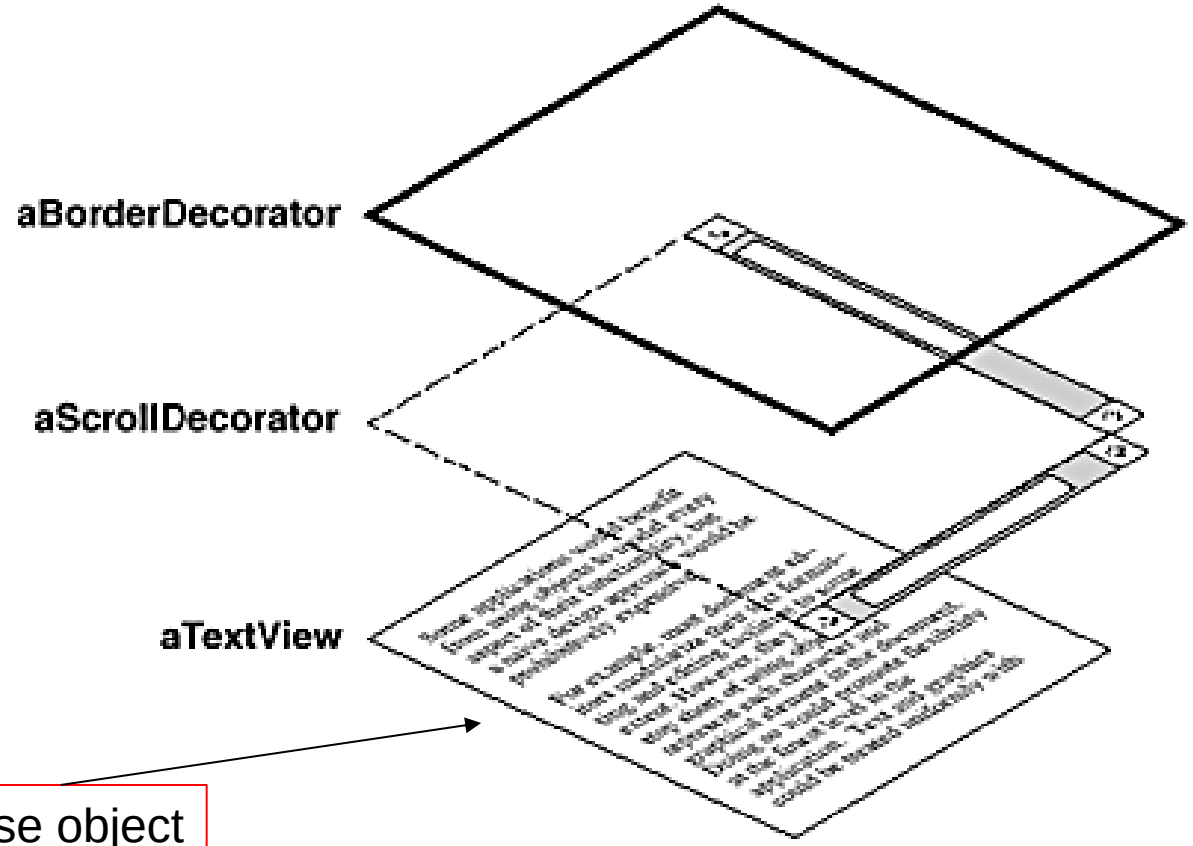


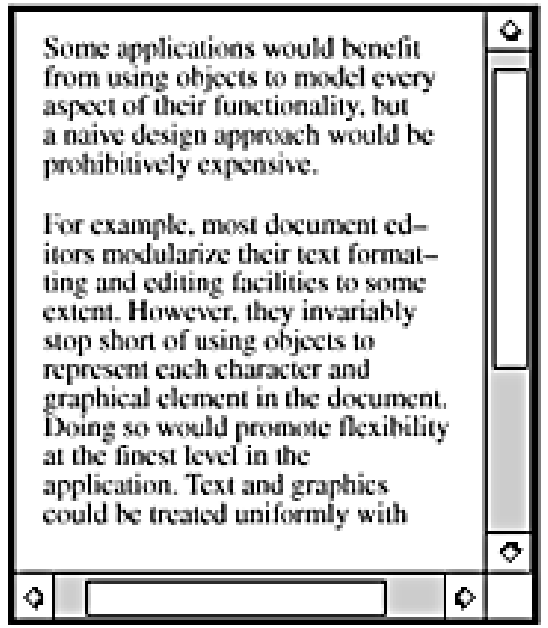
Composite Pattern: Generic Structure



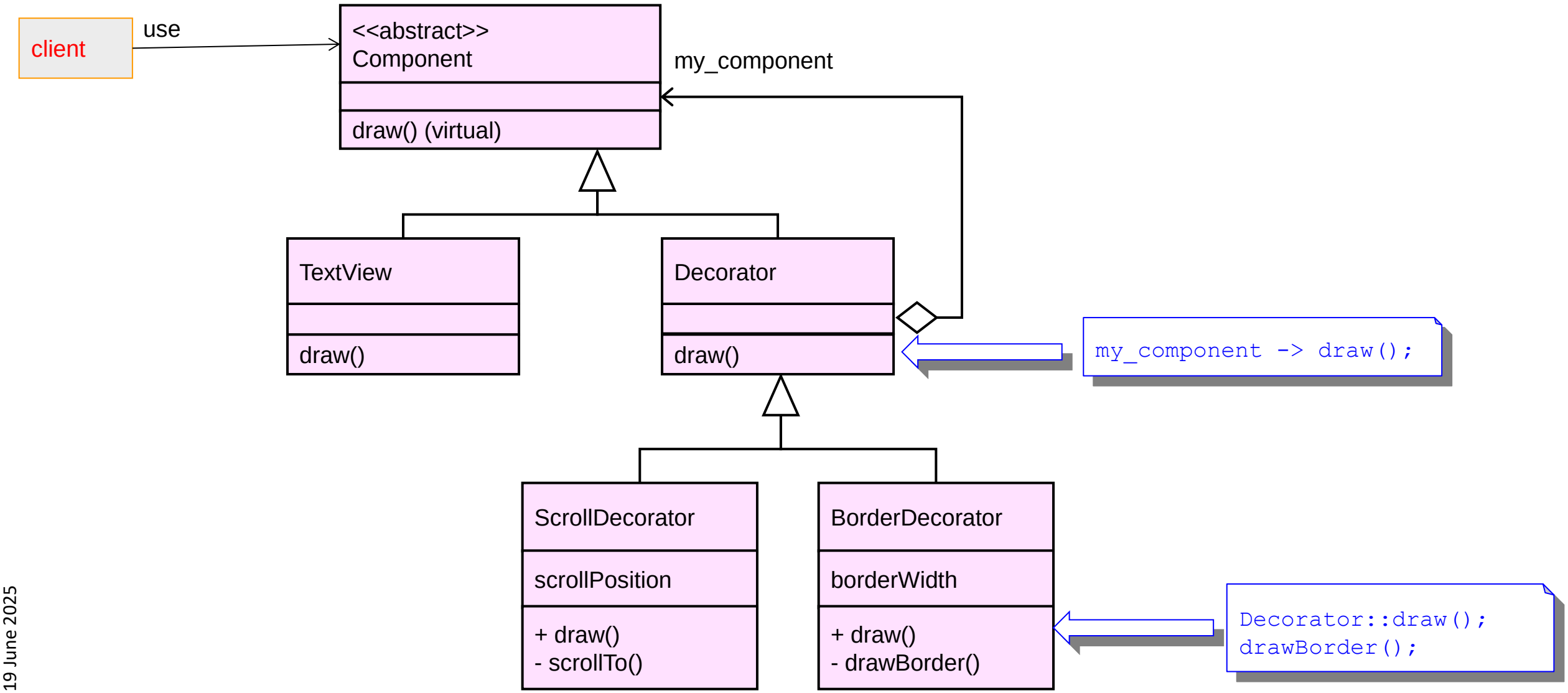
Decorator Pattern (Structural)

- Problem: Give individual object instances extra functionality at run time
- Example: Text display area

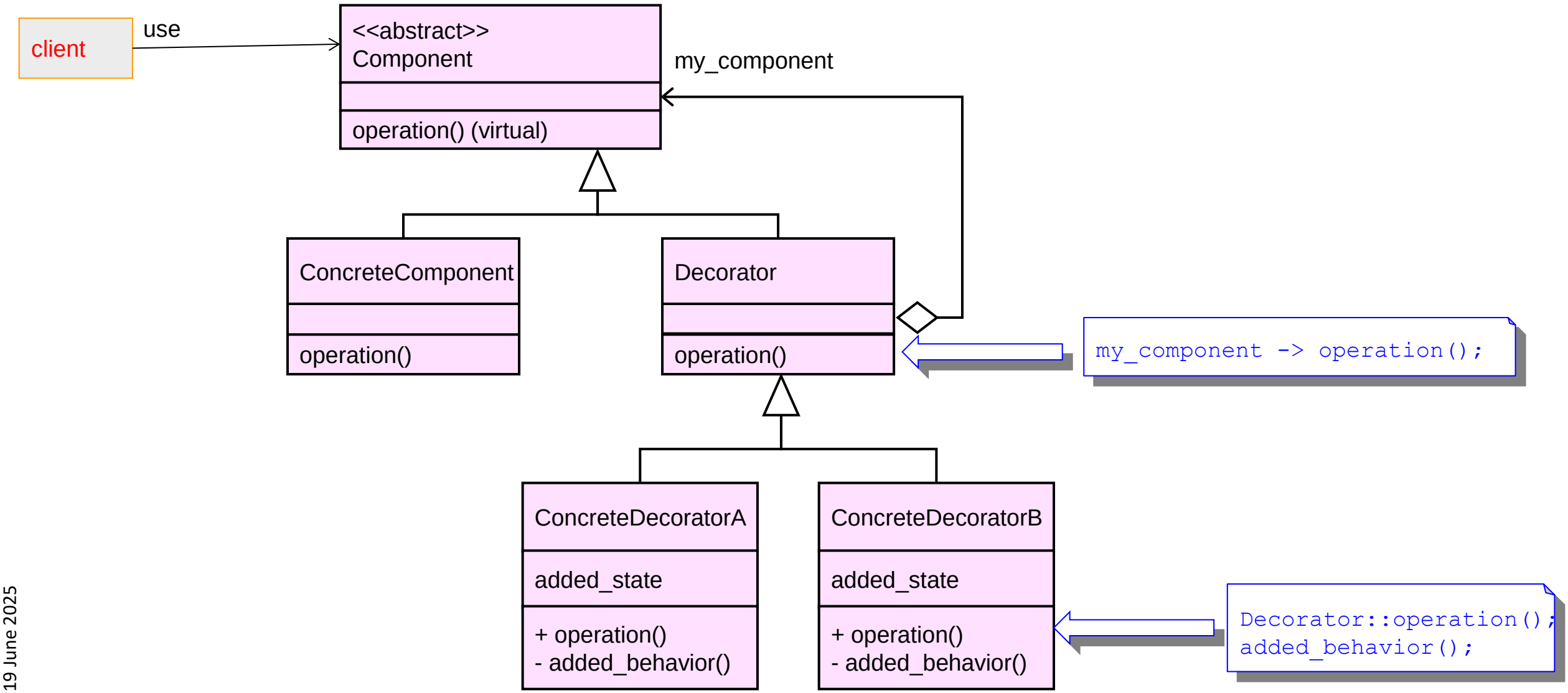




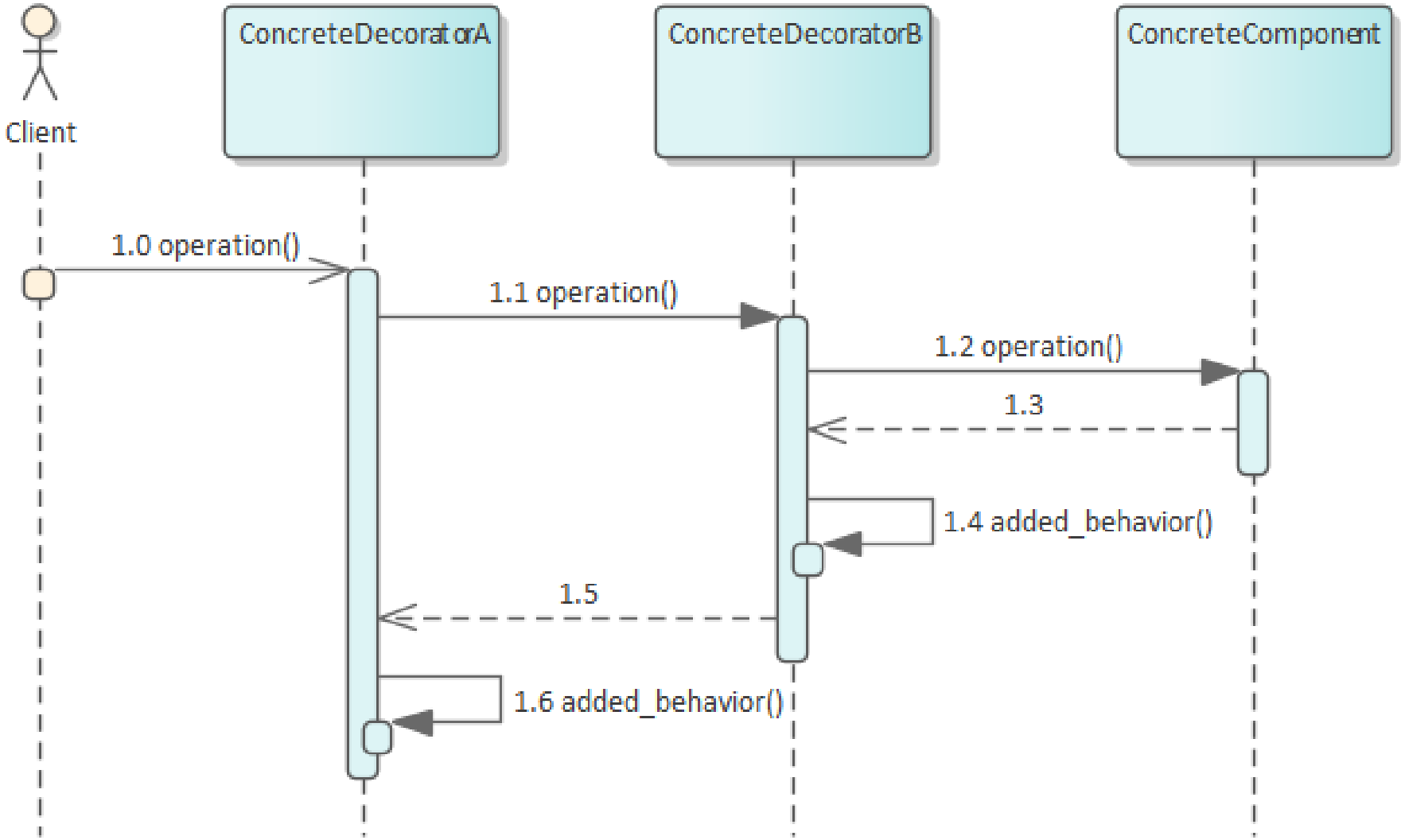
Decorator Pattern: Example



Decorator Pattern: Generic Structure

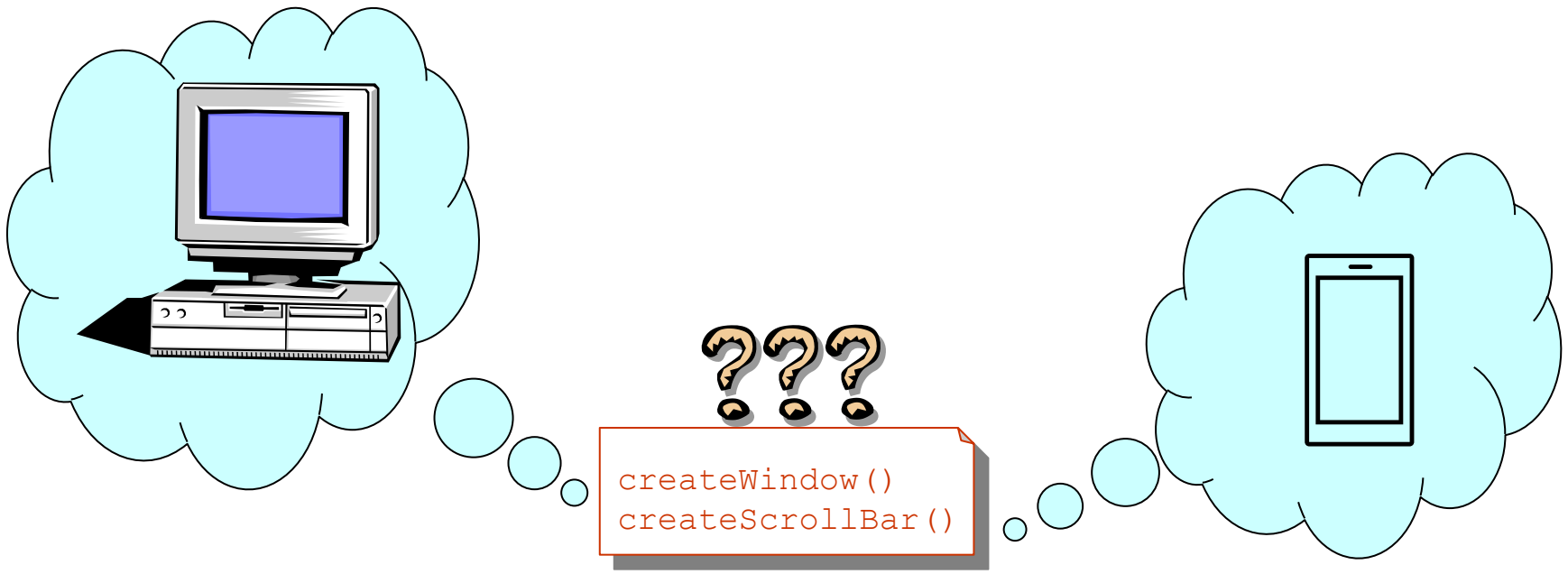


Decorator Pattern: Behavior

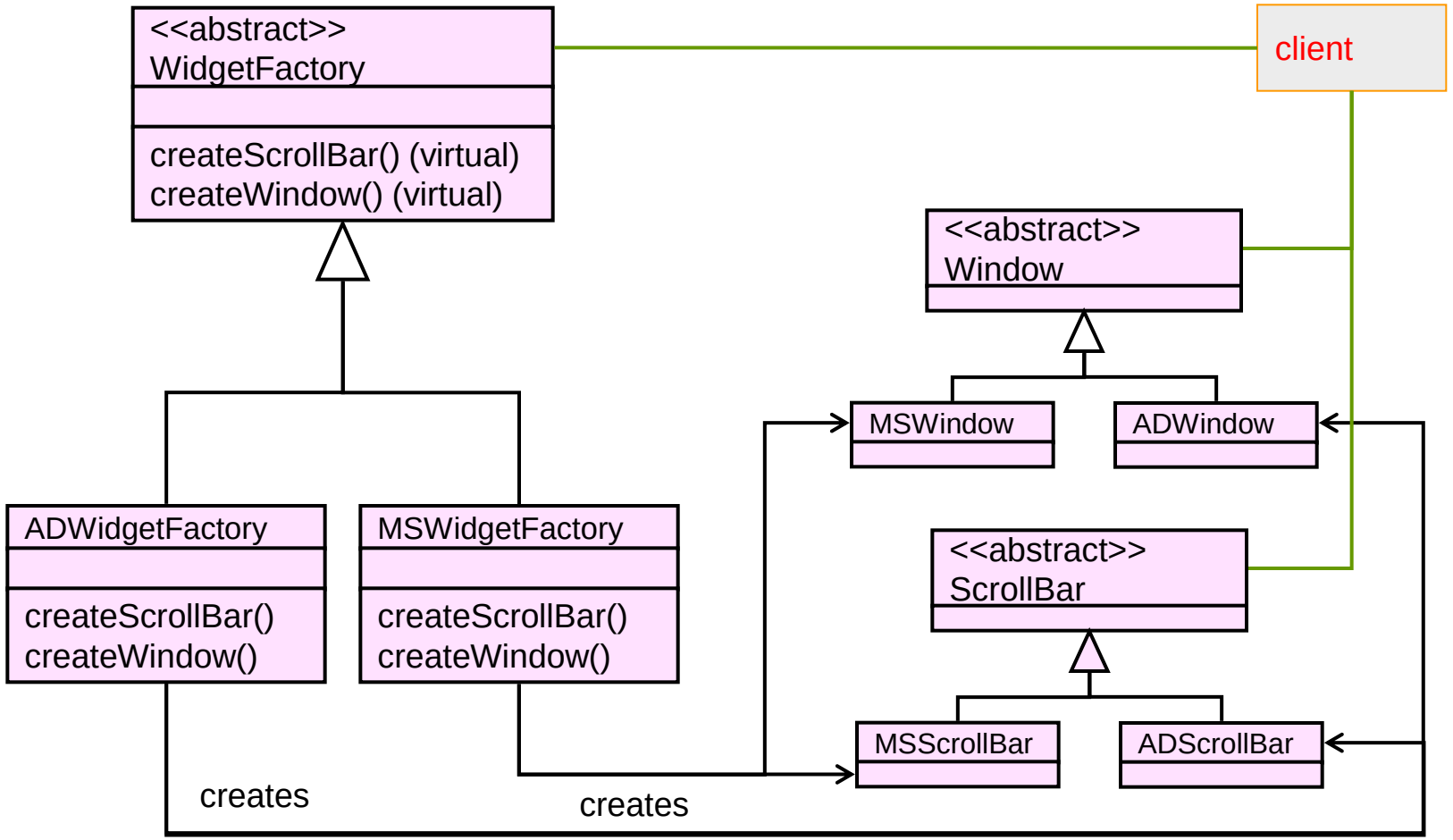


Abstract Factory Pattern (Creational)

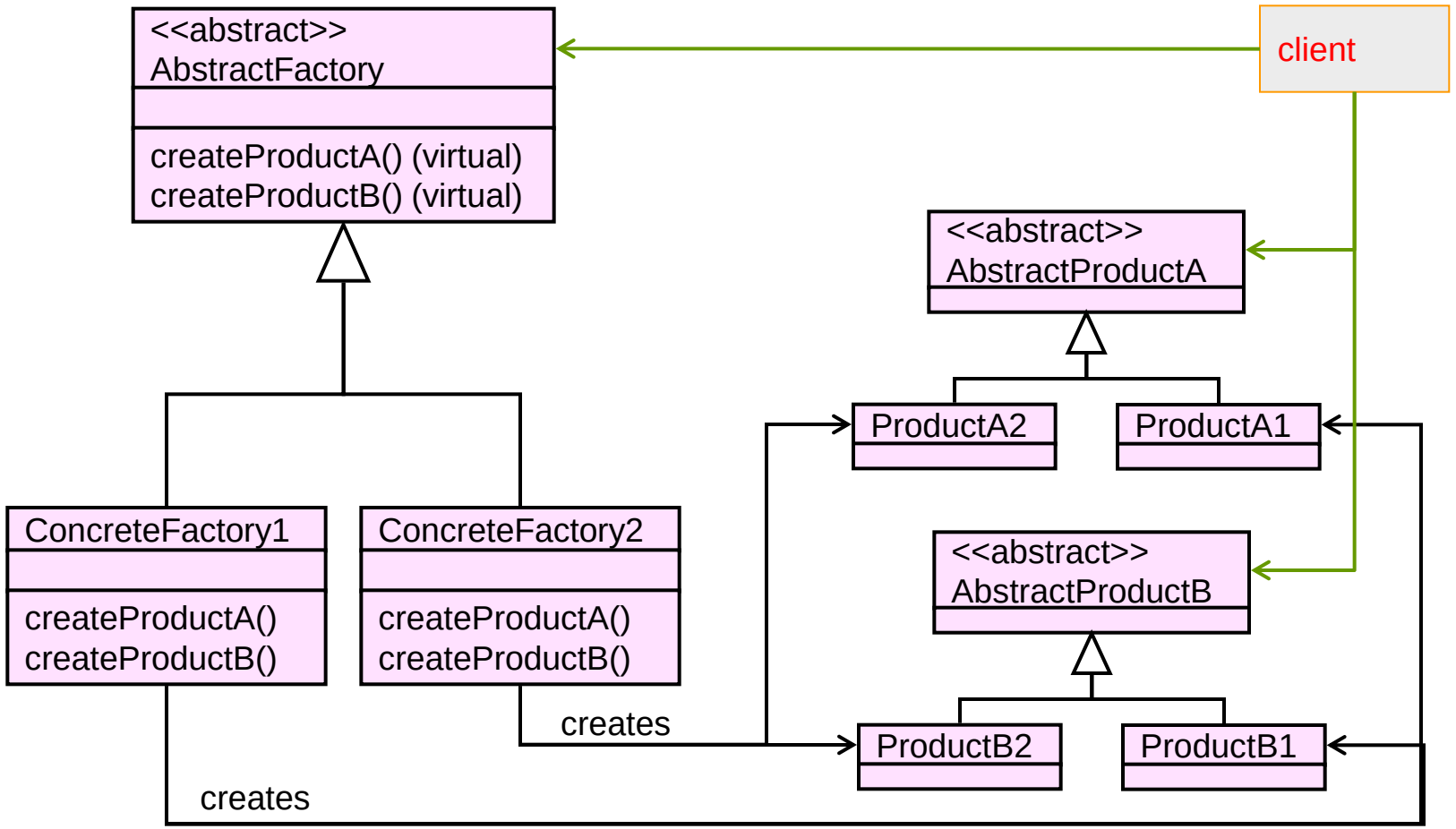
- Problem: Want to create and use objects in changing environments without knowing implementation details
  - User doesn't know or care about implementation
- Example: GUI for Windows and Android



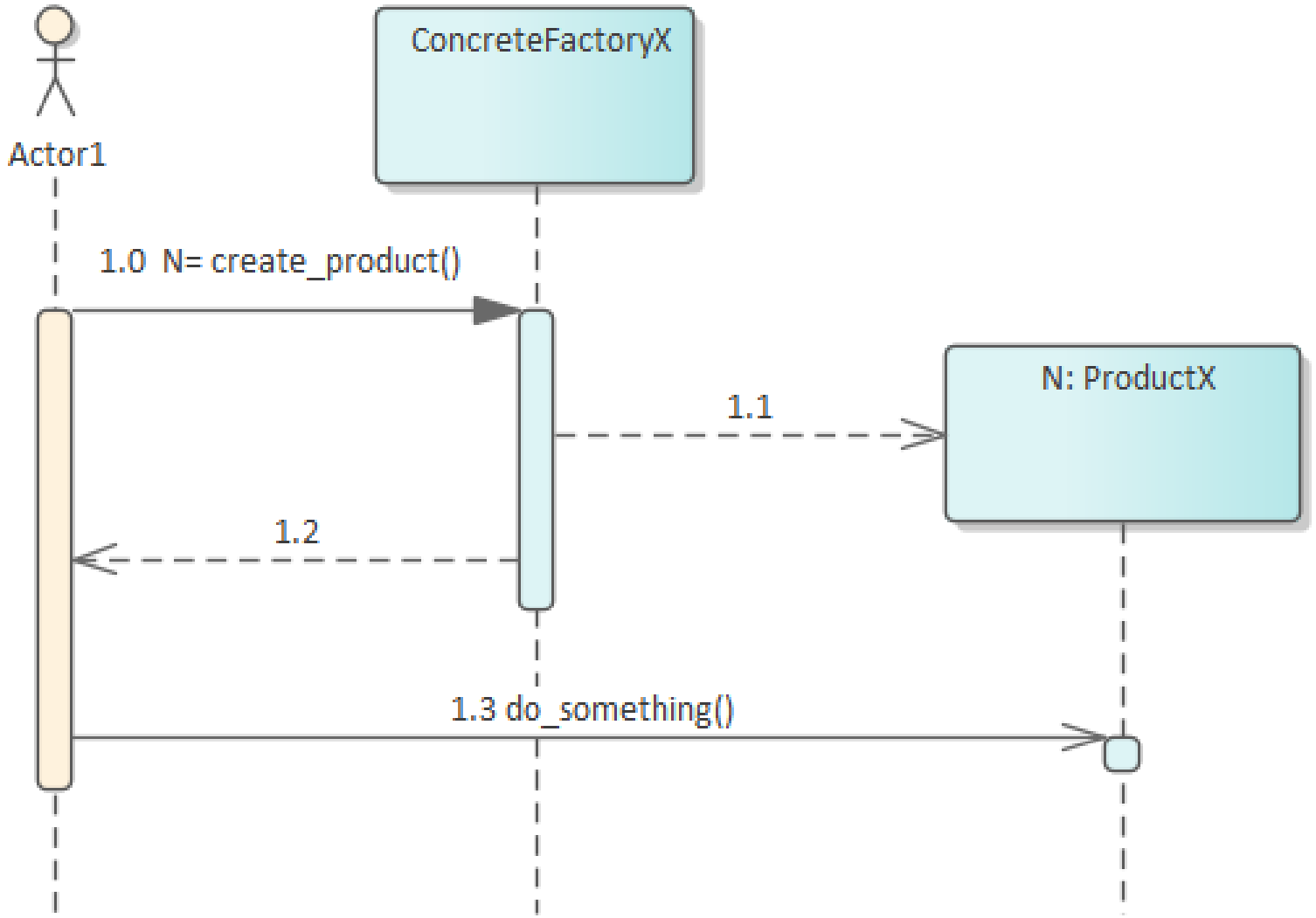
Abstract Factory Pattern: Example



Abstract Factory Pattern: Generic Structure

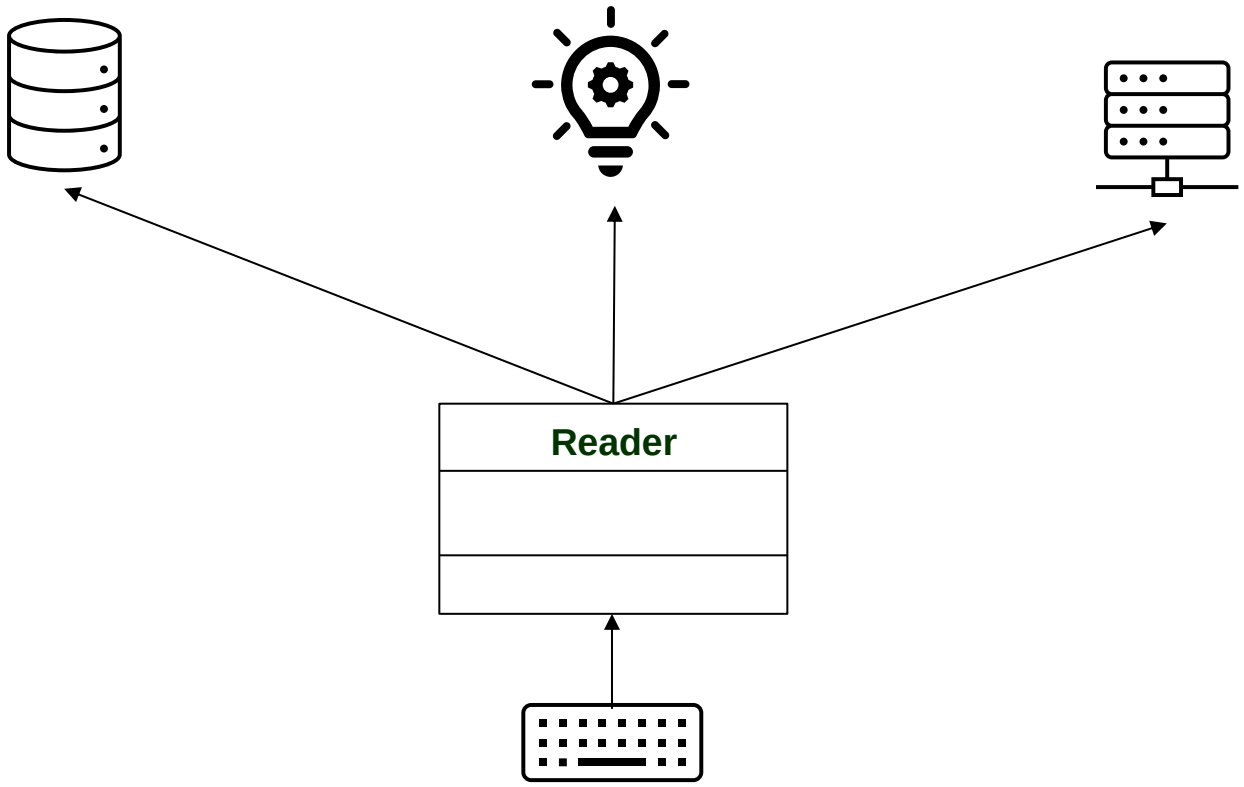


Abstract Factory Pattern: Behavior



Observer Pattern (Behavioral)

- Problem: Want to display or process data in multiple ways
- Example: Want to enable one object to inform others of events



Observer Pattern: Generic Structure

Subject

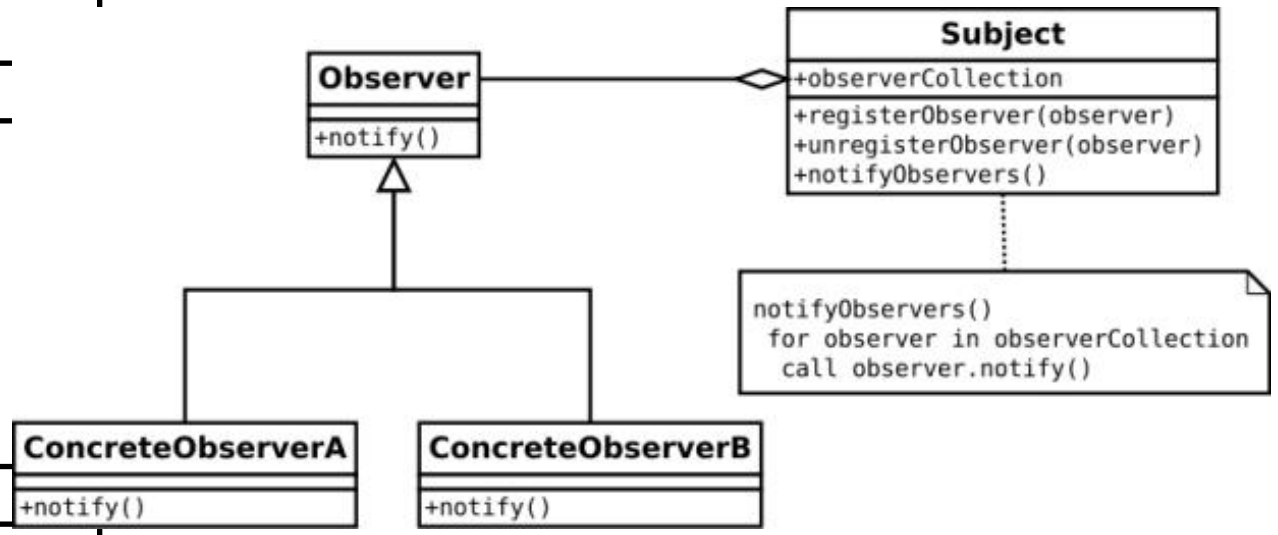
- Holds the data
- Keeps list of observers
- Informs observers when data change

Observer

- Interface / Virtual class

Concrete Observer

- Watches for one purpose
- Responds to the changes appropriately



Strategy Pattern (Behavioral)

- Problem: Want to perform an action different ways, select at run time
- Example: Give directions in a car visually or orally

Strategy Pattern: Generic Structure

Context

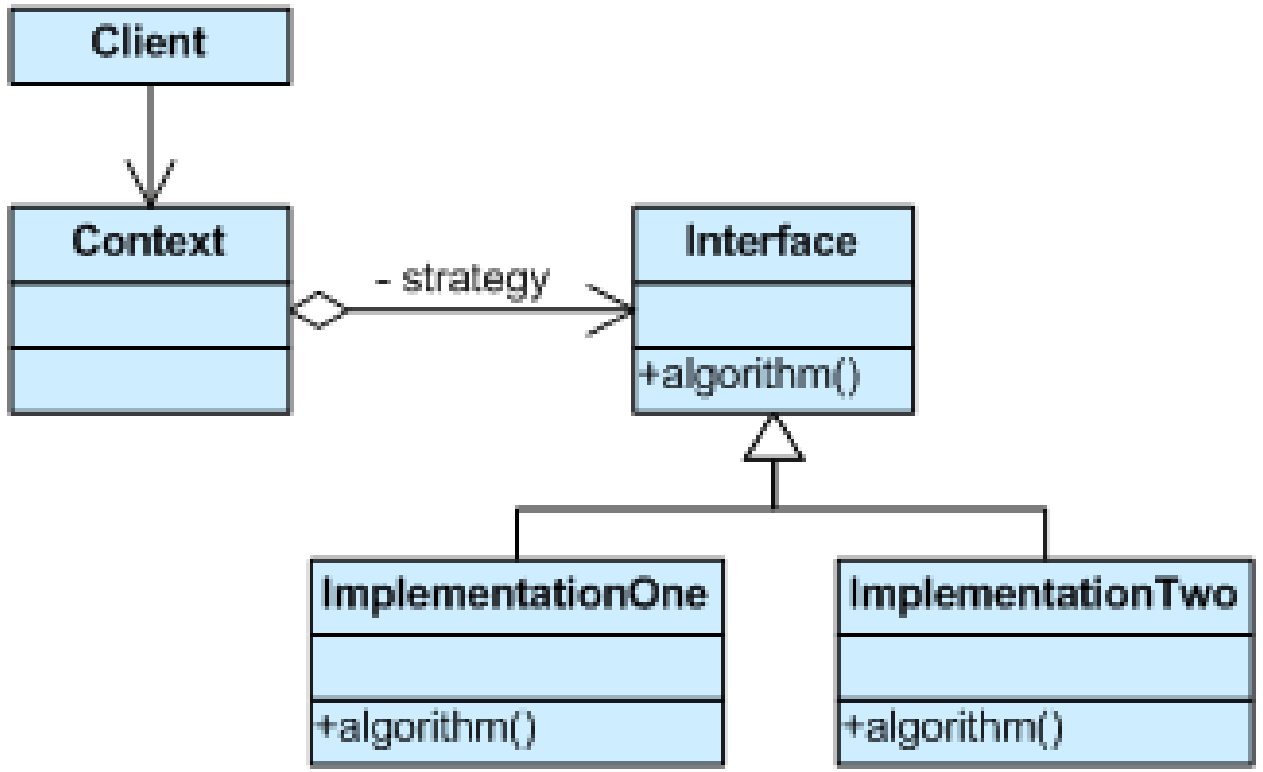
- The service that can do things more than one way
- Doesn't want details of implementation

Interface

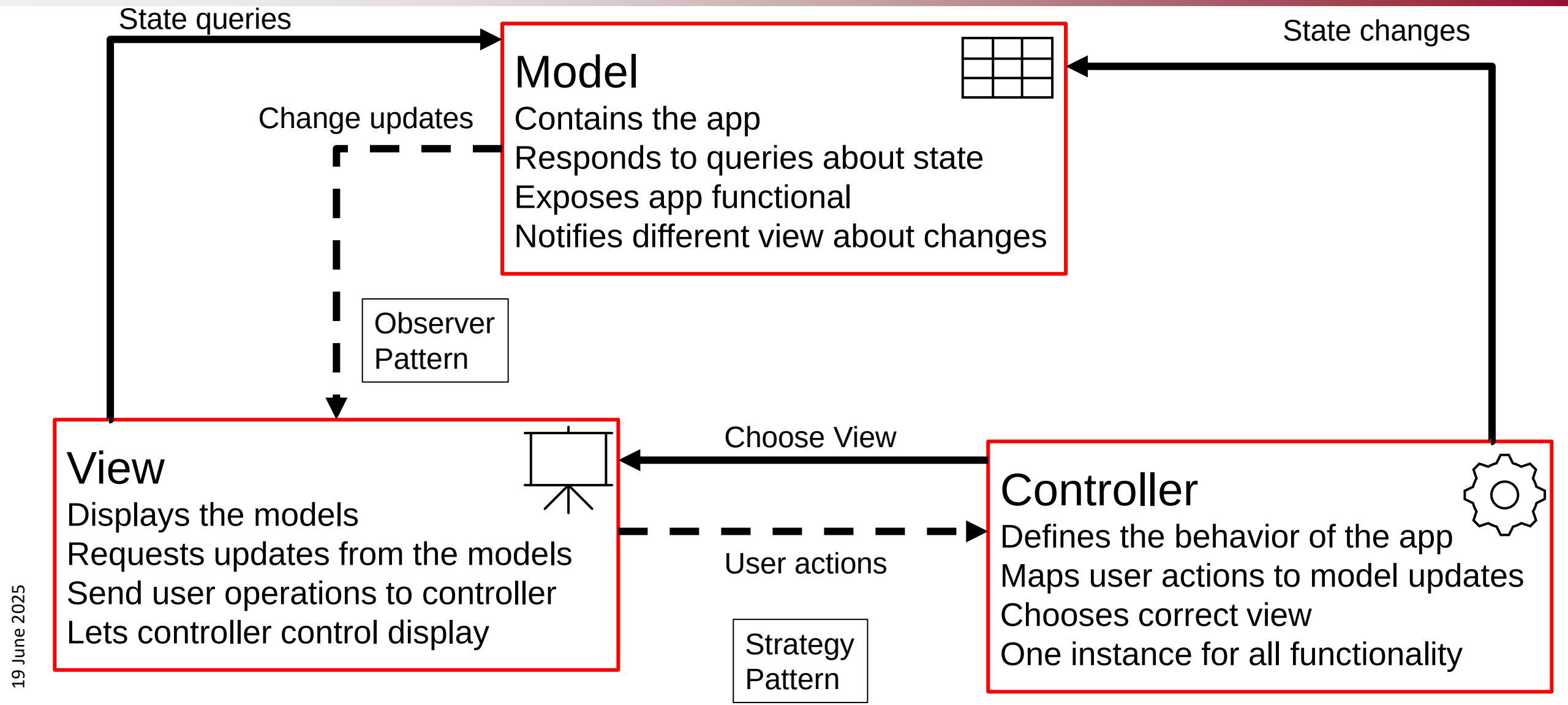
- Algorithm's generic interface

Implementation One, Two

- Different ways to do the algorithm

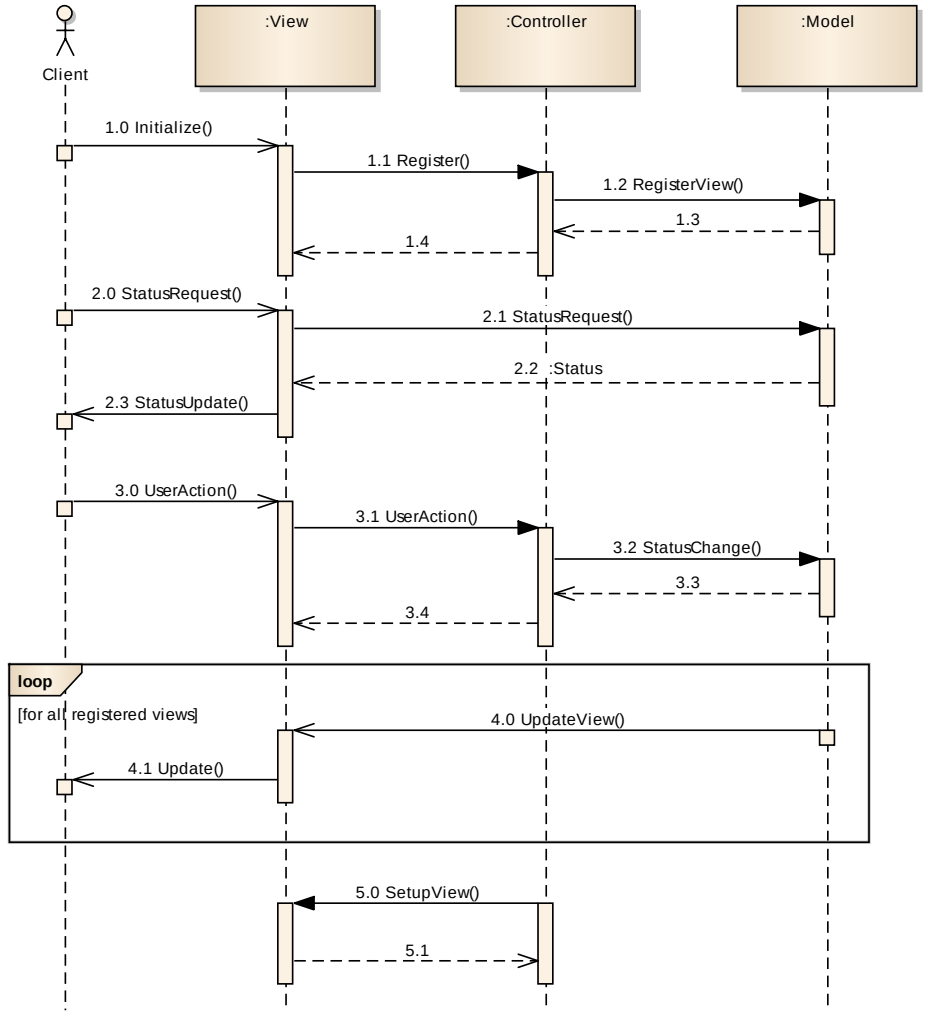


Model View Controller



MVC Conceptual Operation

1. Start and register the view (Observer)
2. Query the model (Observer)
3. User action to update model (Strategy)
4. Update all views (Strategy)
5. Choose view (Strategy)



Anti-Patterns

- God Object
  - A single class that has the entire app logic
  - Effectively procedural programming inside OOP
- Sequential Coupling
  - Class that requires its methods be called in a specific order
  - E.g. init(), begin(), start()
- Poltergeist
  - Short lived object that helps move data between objects
- Singletonitis
  - Overuse of the singleton pattern
  - Singleton class is one with only one instance – e.g. Controller, Database

So Far

- Software Design Patterns
- SOLID

SOLID – 5 Fundamentals for Flexible, Maintainable OOP

- **S**ingle Responsibility Principle (SRP)
- **O**pen Close Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

Single Responsibility Principle (SRP)

Every class has exactly one responsibility

- Responsibility = reason to change
- Every class therefore has only one reason to change

Example: Class representing an Email message

- Two reasons to change:
  - Change in the email message format (e.g. text, HTML)
  - Change in the email protocol (e.g., IMAP, POP3)

Implementing Email Messages – Double Responsibility

```
interface IEmail {
      public void setSender(String sender);
      public void setReceiver(String receive
      public void setContent(String content);
}

class Email implements IEmail {
      public void setSender(String sender) {// set sender; }
      public void setReceiver(String receiver) {// set receive; }
      public void setContent(String content) {// set content; }
}
```

Responsibility for the message format

Responsibility for the content type

Source: www.oodesign.com

Implementing Email Message – Stable Design

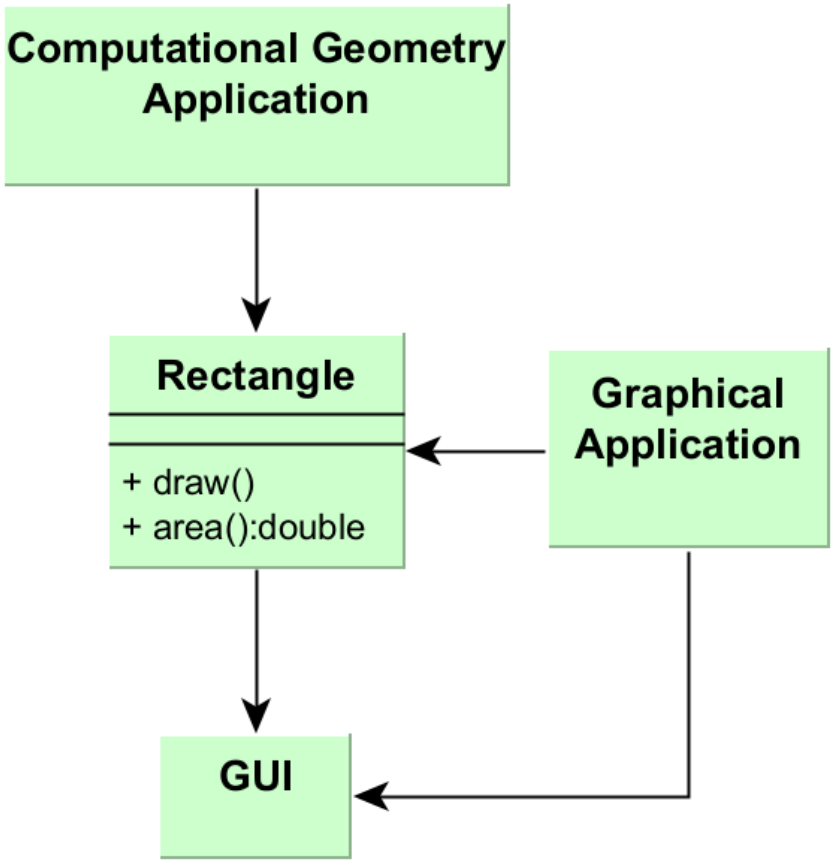
```
interface IEmail {interface IEmail {
    public void setSender(String sender);
    public void setReceiver(String receiver);
    public void setContent(IContent content);
}
interface IContent {
    public String getAsString();
}
class Content implements IContent {
    public String getAsString(); // used for serialization
}

class Email implements IEmail {
    public void setSender(String sender) {// set sender; }
    public void setReceiver(String receiver) {// set receiver; }
    public void setContent(Content content) {// set content; }
}
```

Responsibility for the content type

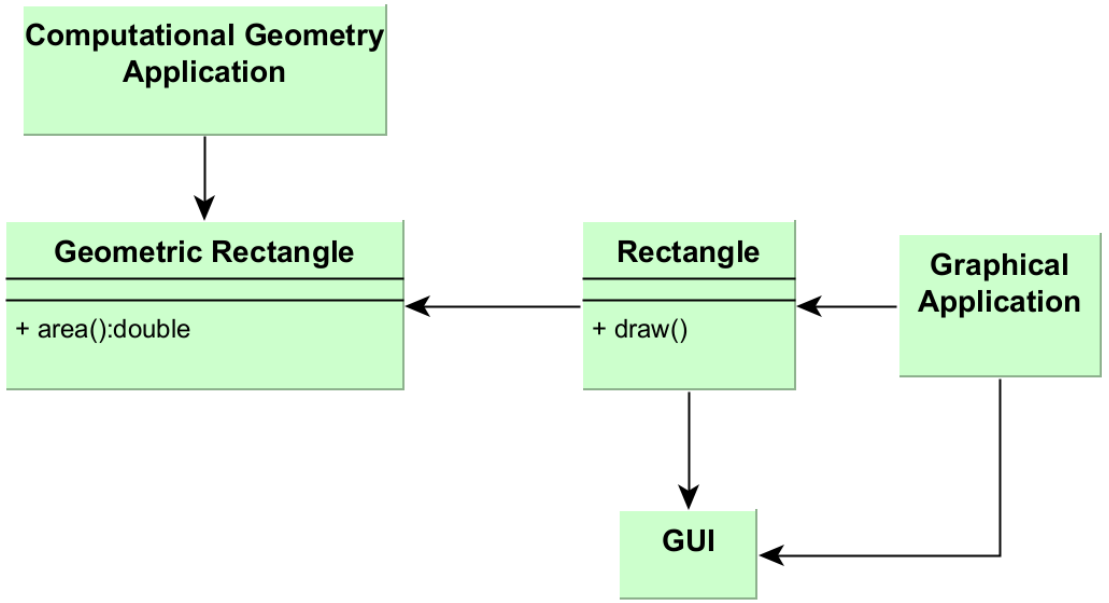
Responsibility for the message format

Another Example: Separating Responsibility for Model and Display

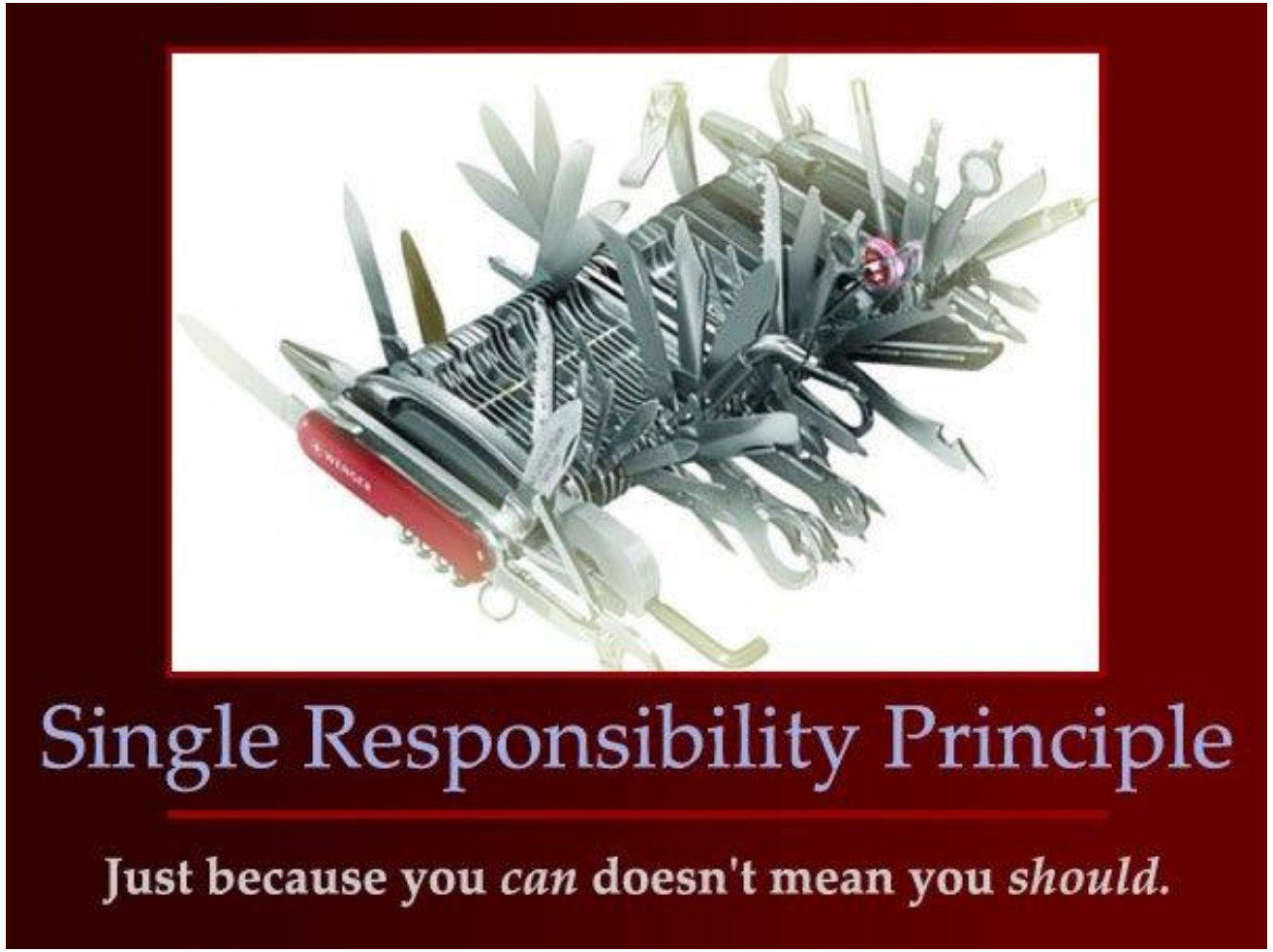
Double Responsibility



Separated Responsibility

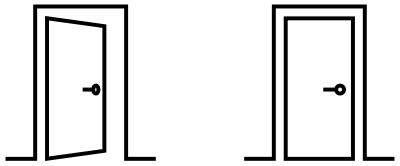


Summary: Single Responsibility



Source: www. themoderndeveloper.com

Open-Close Principle (OCP)



A Class should be

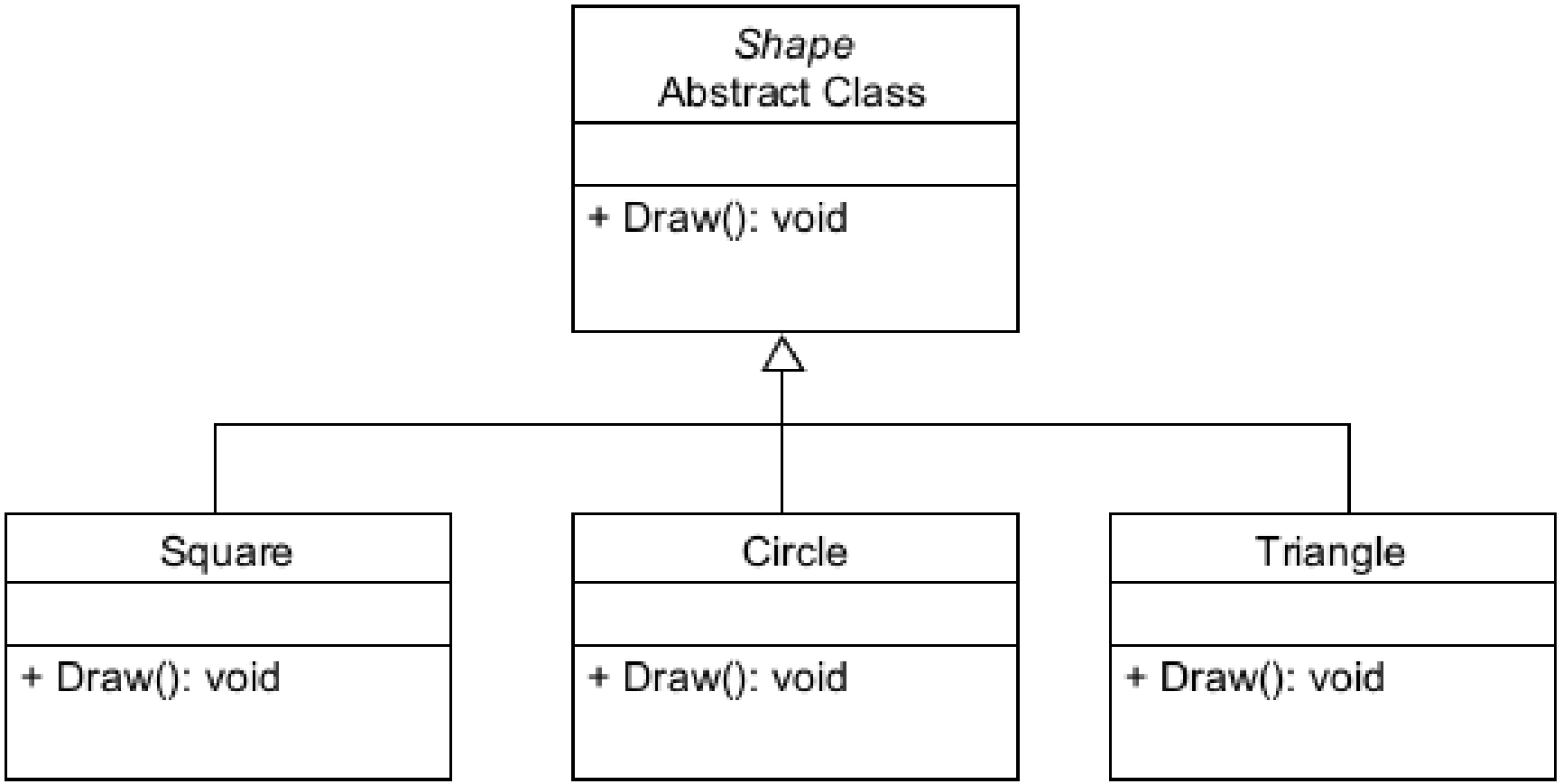
- Open to Extensions
- Closed to Changes

Implication:

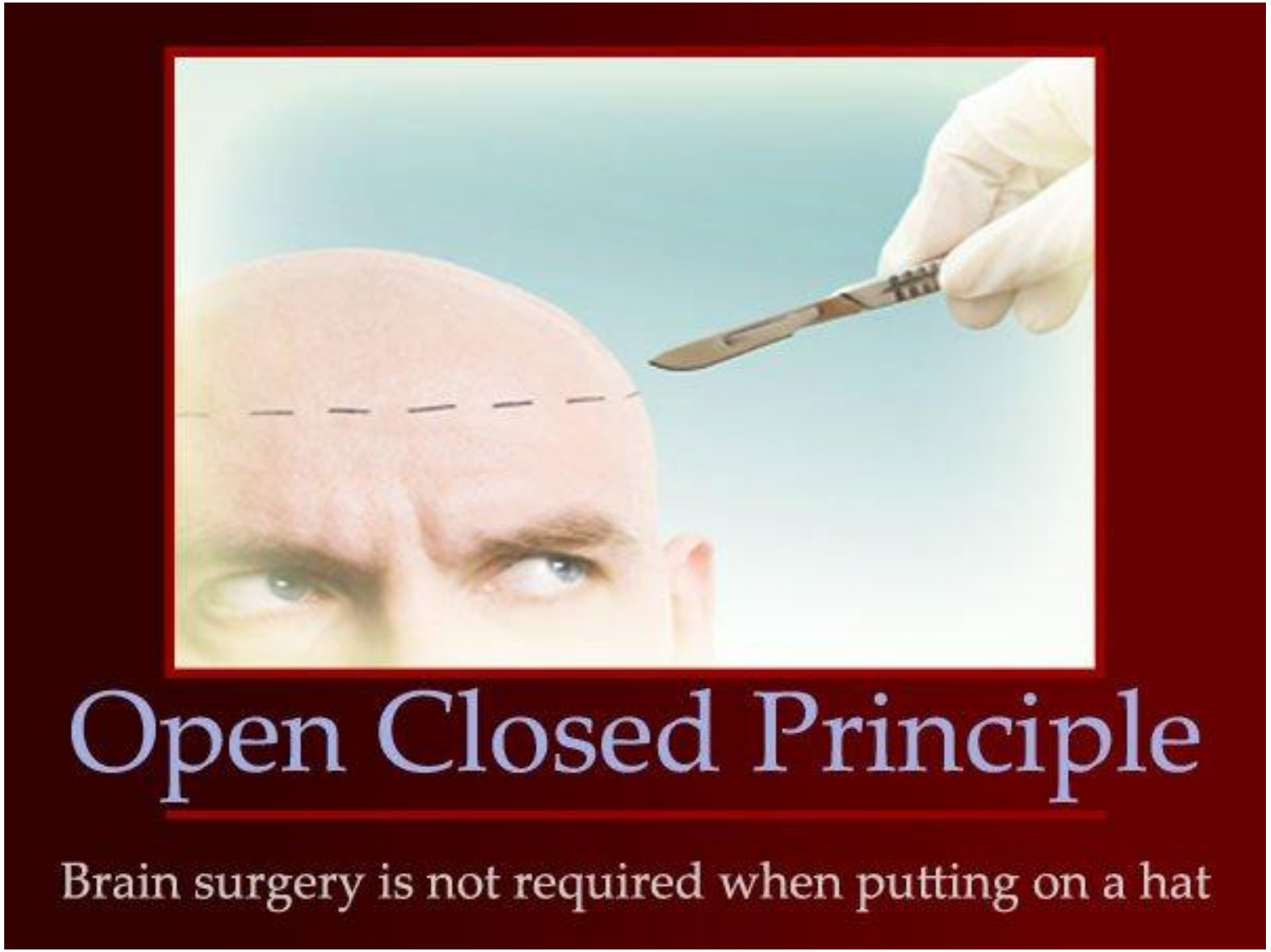
- Changes must be made by adding sub-classes (inheritance) that implement them

Example: Open-Close Principle

- Using Abstract Classes and Concrete Sub-Classes



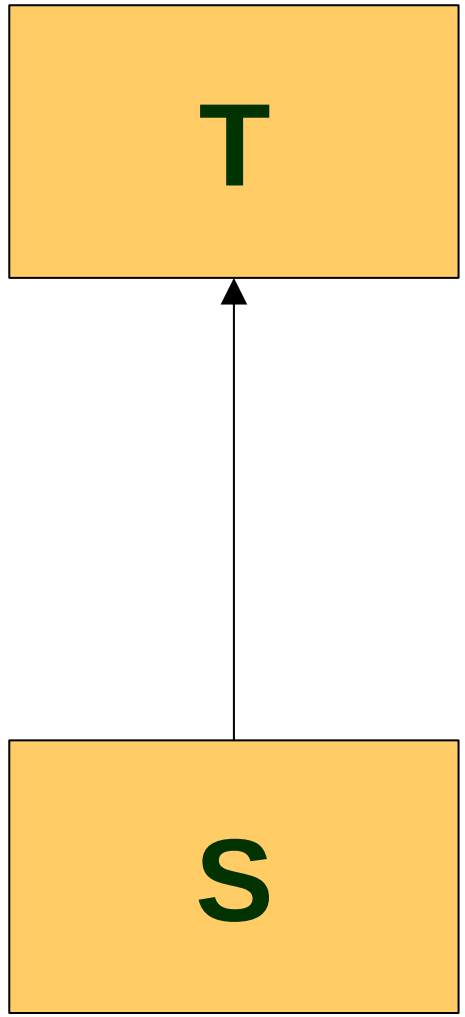
Summary: Open-Close Principle



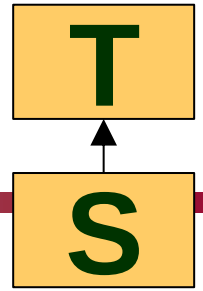
Source: www. themoderndeveloper.com

Liskov's Substitution Principle (LSP)

- If S is a sub-class of T, any instance of T can be replaced with an instance of S without change of behavior



Liskov's Substitution Principle (LSP)



Implications

Sub-class can not modify super-class behavior

Calls to a super-class defined method are handled identically by all sub-classes

Classical Examples

Is Square a sub-class of Rectangle?

Geometrically, yes

Is Circle a sub-class of Ellipse?

Programmatically, no

- Square and Circle require one parameter
- Rectangle and Ellipse require two parameters

Example of Liskov Substitution

```
class Rectangle {
   protected int m_width;
   protected int m_height;
   public void setWidth(int width){
      m_width = width;
   }
   public void setHeight(int height){
      m_height = height;
   }
   public int getArea(){
      return m_width * m_height;
   }
}
```

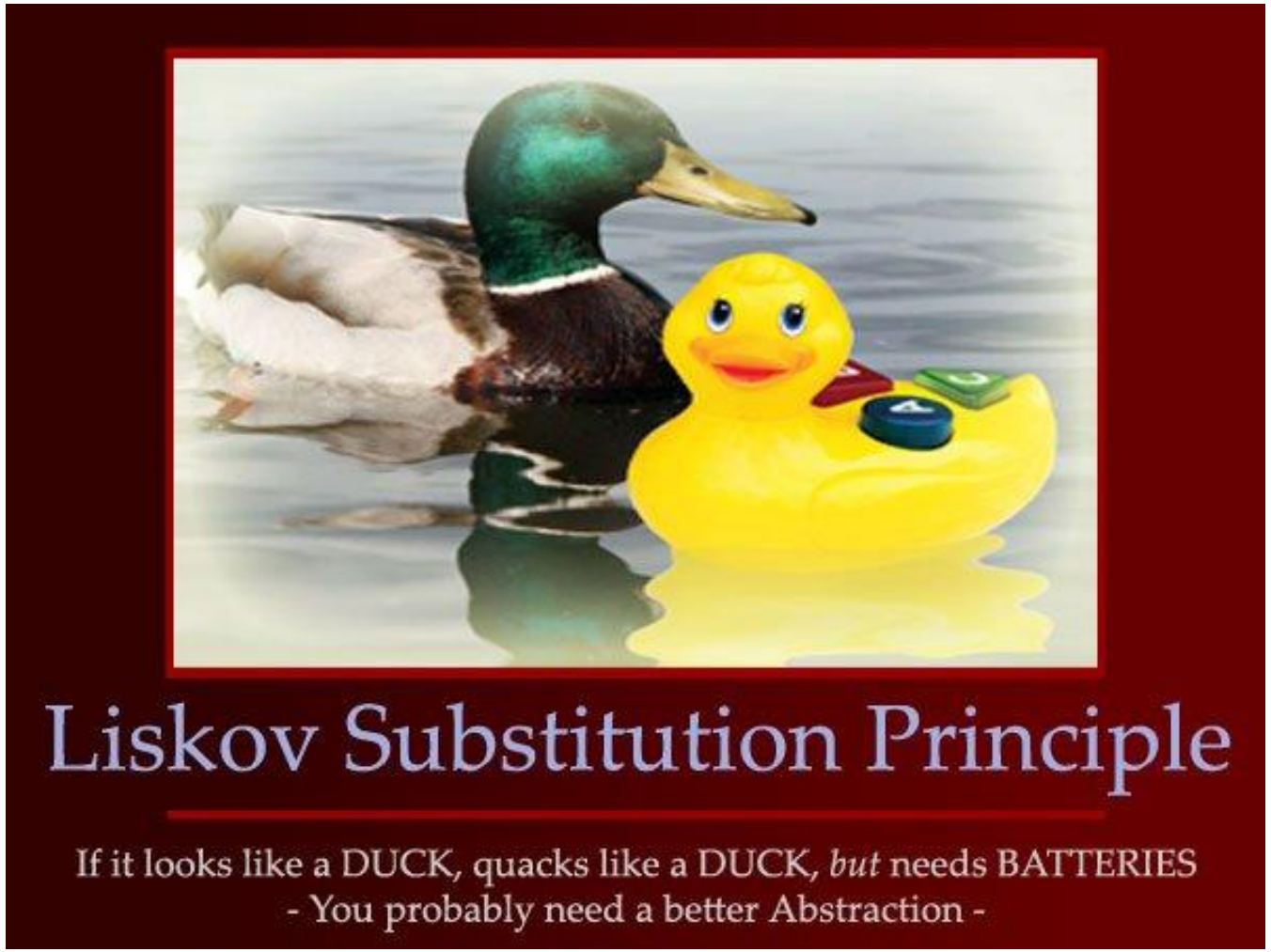
"Trick" since a square has identical height and width

```
quare extends Rectangle {
 public void setWidth(int width){
    m_width = width;
    m_height = width;
 }
  public void setHeight(int height){
     m_width = height;
     m_height = height;
  }
}
```

Example of Liskov Substitution

```
class LspTest {
  private static Rectangle getNewRectangle() {
     /* it can be an object returned
                      by some factory ... */
     return new Square();
}
  public static void main (String args[]) {
     Rectangle r = LspTest.getNewRectangle();
     r.setWidth(5);
     r.setHeight(10)
     System.out.println(r.getArea())
  }
}
```

Summary: Liskov Substitution



Source: www. themoderndeveloper.com

Interface Segregation Principle (ISP)

Don’t force a Customer to be dependent on an interface it doesn’t use

Replace Fat Interfaces with Thin Interfaces

- Tailored to specific Customer types

Example: Interface Segregation Principle (ISP)

A `TimedDoor` is

- A regular door
- A timer

Given:

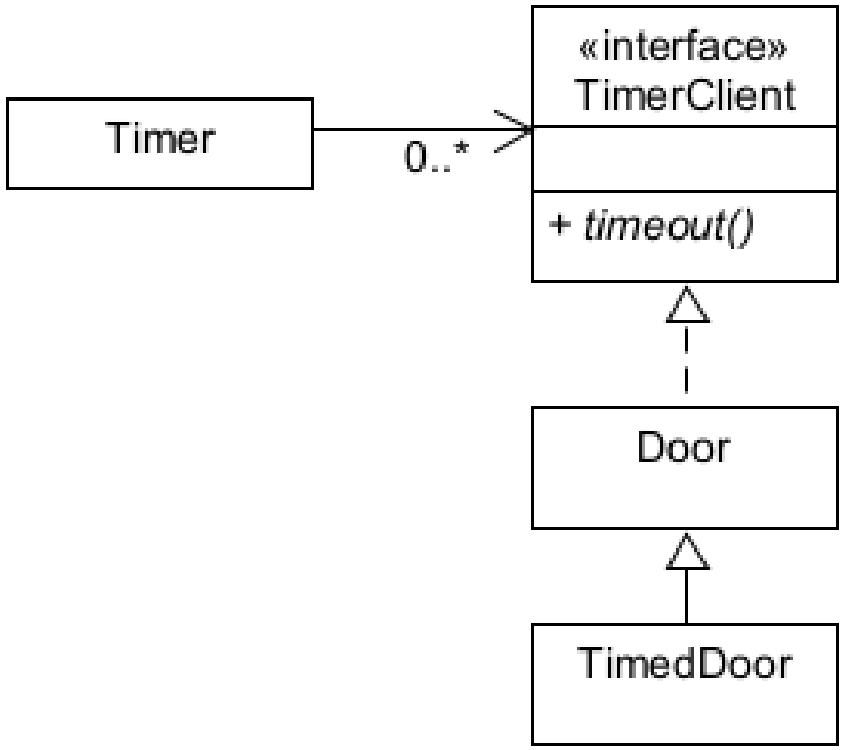
- `Door` class (regular)
- `Timer` class – wakes up `TimerClient`'s waiting for it
- `TimerClient` interface – objects waiting for wakeup

How should `TimedDoor` look?

```
public class Door {
    public void lock() { /* impl*/ }
    public void unlock() { /*impl*/ }
    public boolean isOpen() {/*impl*/}
}
public class Timer {
    public void register(int timeout,
TimerClient client) { /* impl*/ }
}
public interface TimerClient {
    public void timeout();
}
```

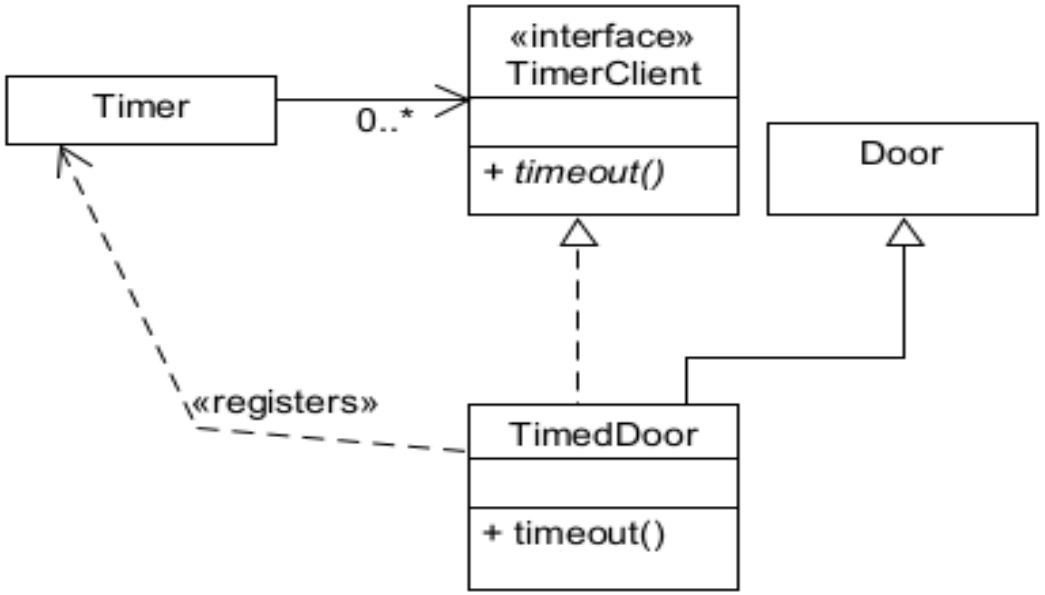
Source: http://www.slideshare.net/JouniSmed/designing-object-oriented-software-lecture-slides-2013

TimedDoor Option 1



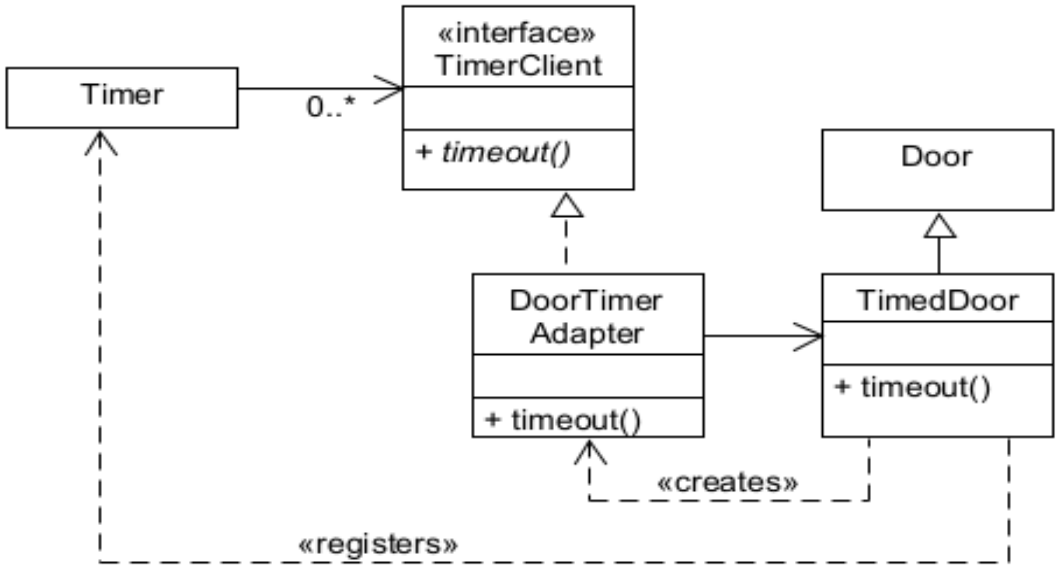
- Put the TimerClient at the time of the hierarchy
- Force every Door to be a TimedDoor

TimedDoor Option 2



- Use multiple inheritance
- Not all languages support this!
  - Java does via interfaces

TimedDoor Option 3



- Adapter Pattern
- `DoorTimerAdapter` is a helper Object

Summary: Interface Segregation Principle (ISP)

Source: www. themoderndeveloper.com

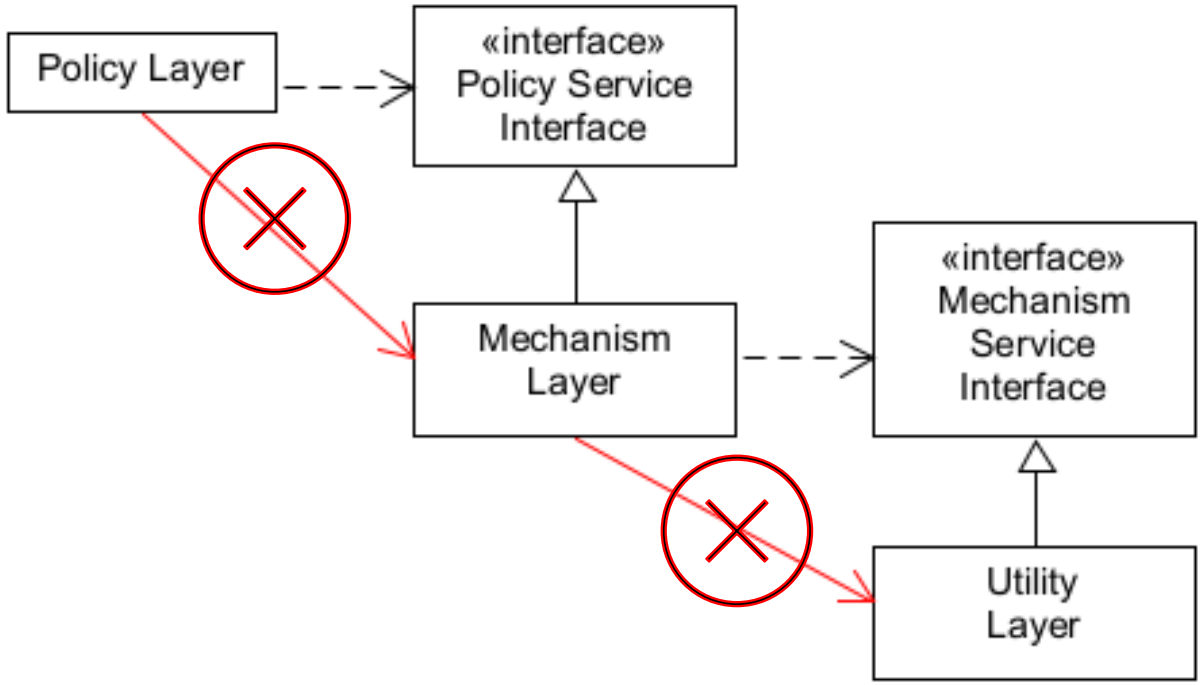
Dependency Inversion Principle (DIP)

High level modules should not be dependent on low level modules

Both should depend on Abstractions

Abstractions should not depend on Details

Details should depend on Abstractions



Example: High level object depends on low level object

We have a `Thermostat` class

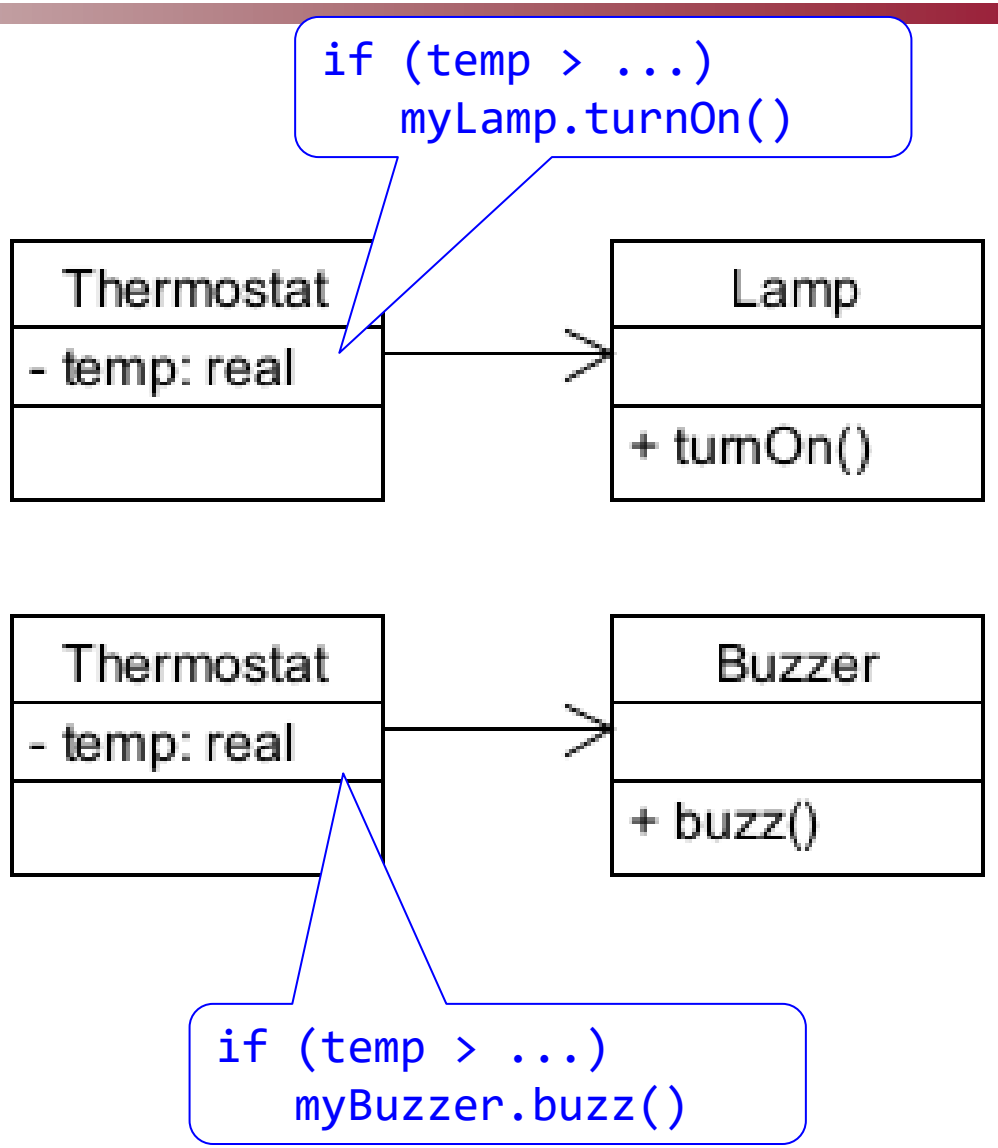
- Contains `Lamp`
- Operates `Lamp`

`Thermostat` contains `Lamp` → Changing `Lamp` for `Buzzer` requires us to change `Thermostat`!

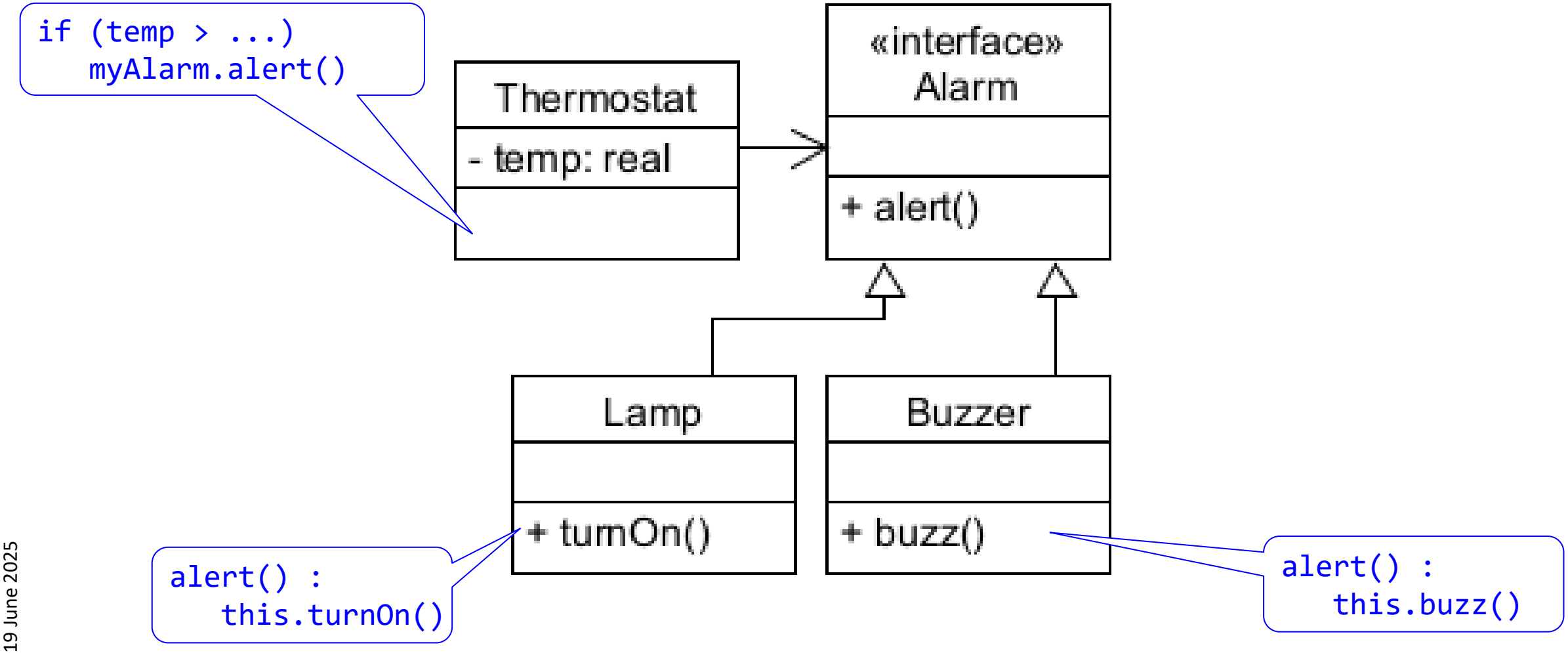
`Thermostat` is high level object

- Could notify in many ways

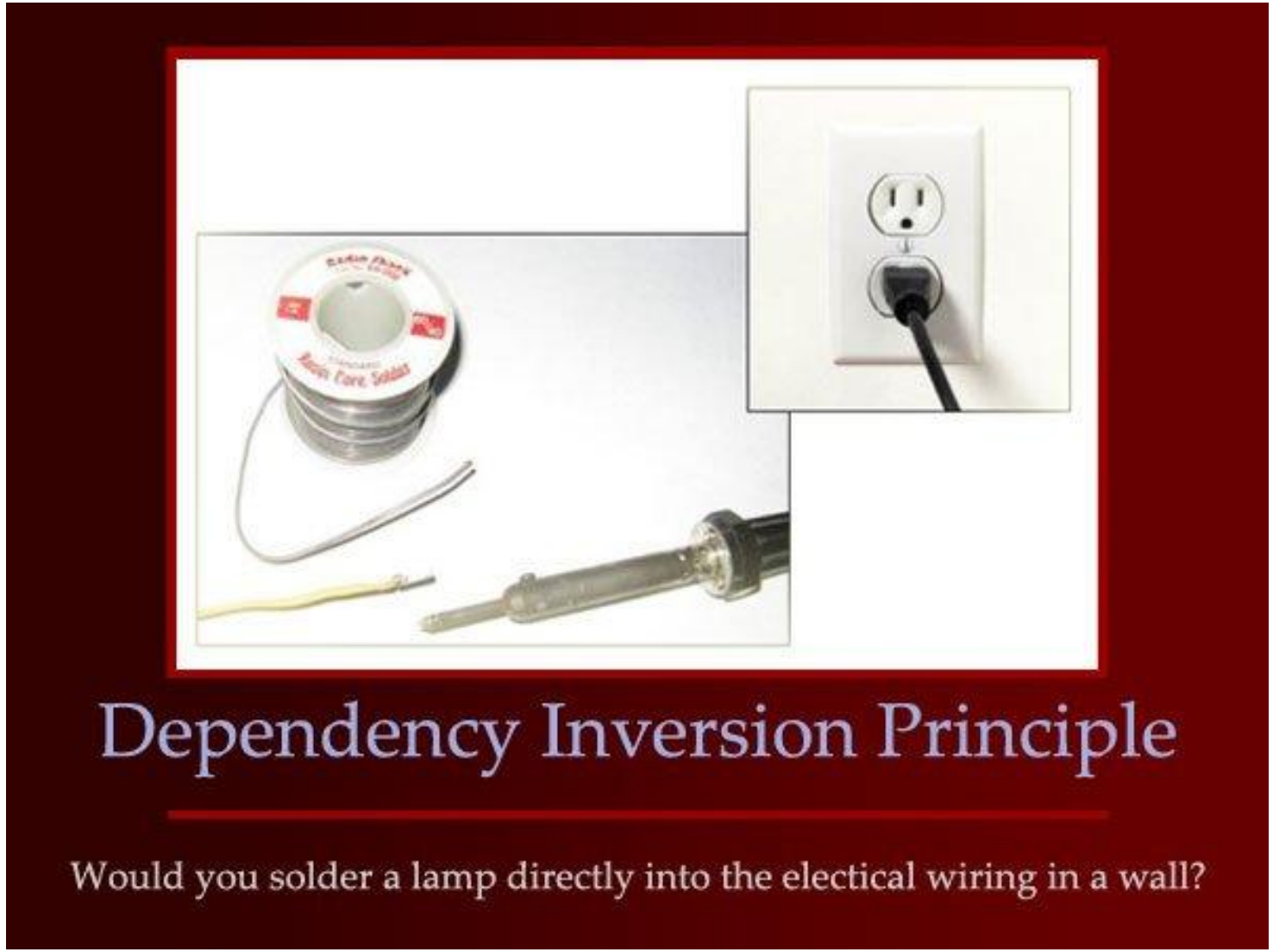
`Lamp` & `Buzzer` are low level objects



Dependency Inversion via Abstraction



Summary: Dependency Inversion Principle (DIP)



Source: www. themoderndeveloper.com

Conclusion

- Software Design Patterns
- SOLID