

Design Patterns

- **More design patterns (GoF)**
 - **Structural:** Adapter, Bridge, Façade
 - **Creational:** Abstract Factory, Singleton
 - **Behavioral:** Observer, Iterator, State, Visitor

Design Patterns

- **Design patterns have become very popular in the last decade or so**
- **Major source: GoF book 1995**
 - "Design Patterns: Elements of Reusable Object-Oriented Software"
 - Gamma, Helm, Johnson, Vlissides (gang of 4)
- **Patterns describe well-known solutions to common design problems**
 - Used in Java libraries, especially in the GUI libraries

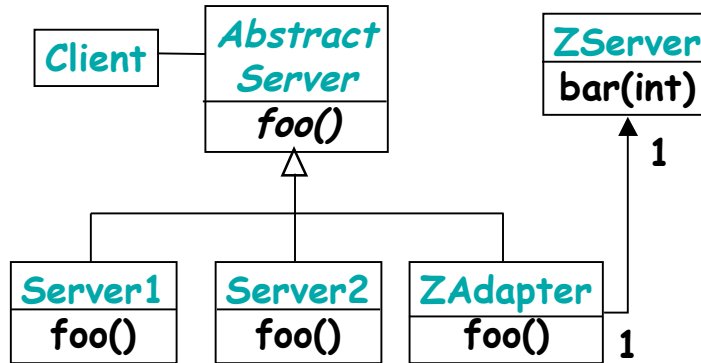
Design Patterns (LAR Ch26; GoF)

- **Structural**
 - Concerned with how classes and objects are composed to make larger structures (Adapter, Bridge, Composite, Façade)
- **Creational**
 - Abstract the instantiation process to make a system independent of how its objects are created & represented (Abstract Factory, Singleton)
- **Behavioral**
 - Describe patterns of communication and interaction between objects (algorithms and responsibility assignment) (Observer, State, Strategy, Mediator)

Adapter Pattern: Interface Matcher

- **Problem: incompatible interfaces**
- **Solution: create a wrapper that maps one interface to another**
 - Key point: neither interface has to change and they execute in decoupled manner
 - Think of how you use a power plug adaptor when you travel to Europe
- **Example:**
 - Client written against some interface
 - Server with the right functionality but with the wrong interface

Example



- Option 1: Change the client, **bad**
- Option 2: Change Zserver, **too hard**
- Option 3: Create an adapter to wrap Zserver to obtain necessary functionality, **good soln**

Design Patterns-10, CS431 F06, BG Ryder/A Rountev

5

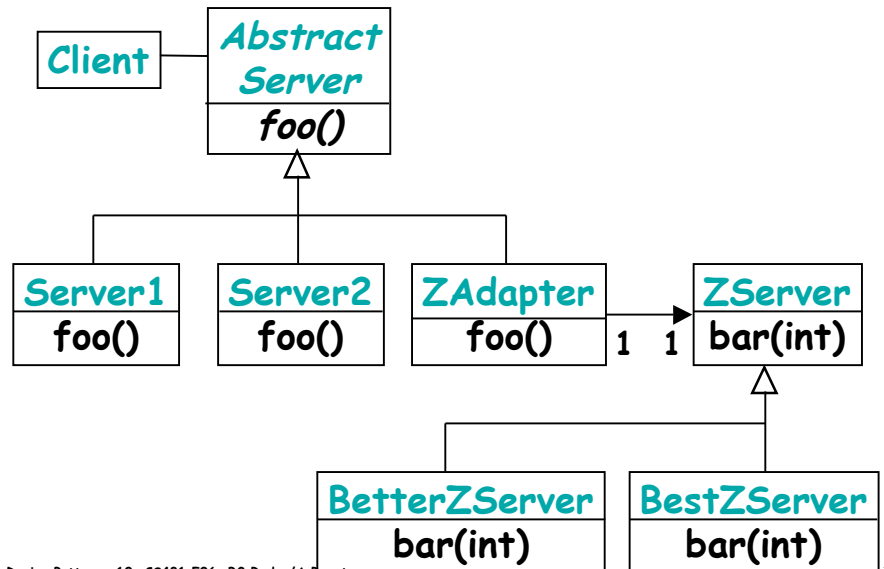
Sample Java Code

```
abstract class AbstractServer { abstract void foo();
}
class ZAdapter extends AbstractServer {
    private ZServer z;
    public ZAdapter() { z = new ZServer(); }
    public void foo() { z.bar(5000); } //wrap call to
    ZServer method
}
...
somewhere in client code:
AbstractServer s = new ZAdapter();
```

Design Patterns-10, CS431 F06, BG Ryder/A Rountev

6

Hierarchy of Adaptees



Design Patterns-10, CS431 F06, BG Ryder/A Rountev

7

Sample Java Code

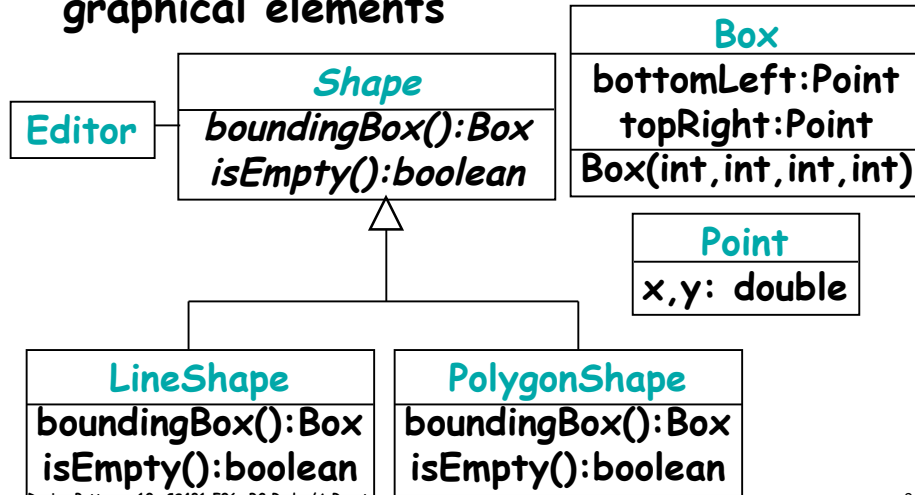
```
abstract class AbstractServer { abstract void foo();
}
class ZAdapter extends AbstractServer {
    private ZServer z;
    public ZAdapter(int perf) {
        if (perf > 10)    z = new BestZServer();
        else if (perf > 3) z = new BetterZServer();
        else             z = new ZServer();
    }
    public void foo() { z.bar(5000); }
}
```

Design Patterns-10, CS431 F06, BG Ryder/A Rountev

8

Another Adapter Example (GoF)

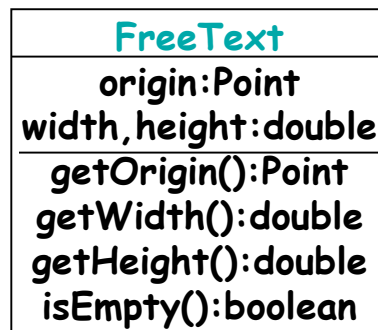
- Drawing editor: diagrams built with graphical elements



9

Adding TextShape

- Complicated, so let's reuse existing code



- Problem: mismatched interfaces
- Solution: create a **TextShape** adapter

Design Patterns-10, CS431 F06, BG Ryder/A Rountev

10

Sample Java Code

```
class TextShape implements Shape {
    private FreeText t;
    public TextShape() { t = new FreeText(); }
    public boolean isEmpty() { return t.isEmpty(); }
    public Box boundingBox() {
        int x1 = toInt(t.getOrigin().getX());
        int y1 = toInt(t.getOrigin().getY());
        int x2 = toInt(x1 + t.getWidth());
        int y2 = toInt(y1 + t.getHeight());
        return new Box(x1,y1,x2,y2); }
    private int toInt(double) { ... } }
```

Design Patterns-10, CS431 F06, BG Ryder/A Rountev

11

Pluggable Adapters

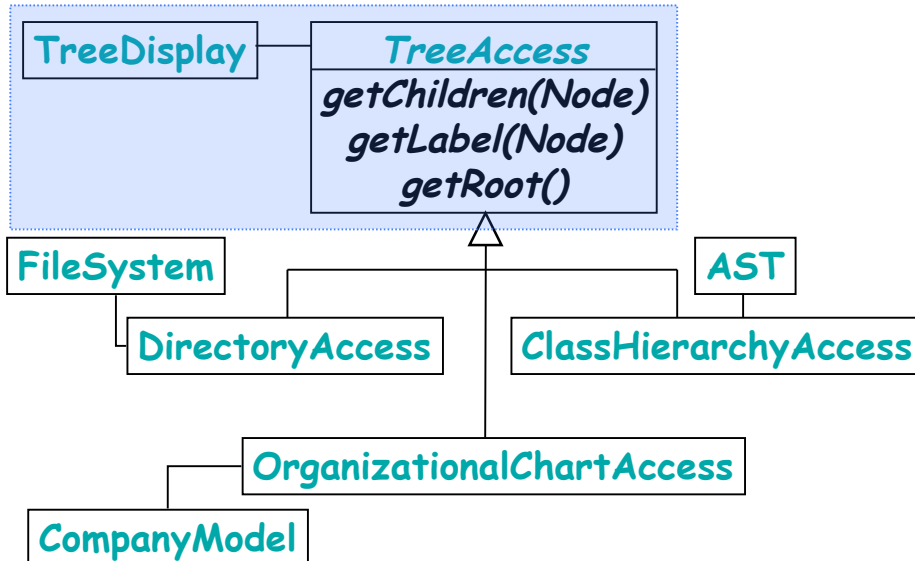
- Preparation for future adaptation
 - Define a narrow interface
- Future users of our code will write adapters that implement the interface



Design Patterns-10, CS431 F06, BG Ryder/A Rountev

12

Example: Display of Trees



Design Patterns-10, CS431 F06, BG Ryder/A Rountev

13

Bridge Pattern

- GoF: "Decouple an abstraction from its implementation so that the two can vary independently"
- Key issue: **dimensions of variability**
- For single dimension: polymorphism based on inheritance or interfaces
 - e.g., different kinds of shapes
- What if there are **several** dimensions?

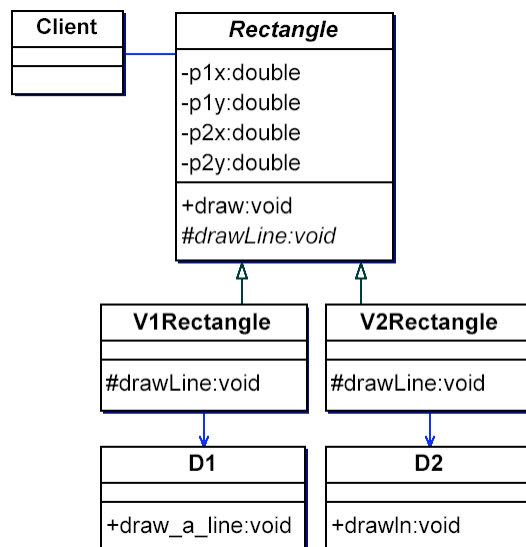
Design Patterns-10, CS431 F06, BG Ryder/A Rountev

14

Example - Why necessary?

- Program that draws rectangles
 - Two drawing classes D1 and D2
 - Each rectangle uses only one of the two
 - When a rectangle is created, we are told which drawing class it will use
- D1 provides
 - draw_a_line(x1,y1,x2,y2) - two vertices
 - draw_a_circle(x,y,r) - center and radius
- D2: drawln(x1,x2,y1,y2), drawcr(x,y,r)

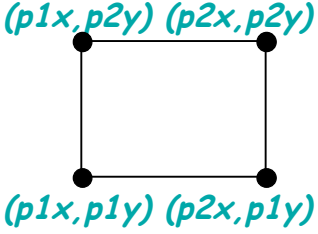
Possible Design



Rectangle Code

```
abstract class Rectangle {
  public void draw() {
    drawLine(p1x,p1y,p2x,p1y);
    drawLine(p1x,p1y,p1x,p2y);
    drawLine(p2x,p2y,p2x,p1y);
    drawLine(p2x,p2y,p1x,p2y);
  }
  protected abstract void
    drawLine(double,double,double,double);
  private double p1x, p1y, p2x, p2y;
  // constructor not shown }

```



The diagram shows a rectangle with four vertices marked by black dots. The top-left vertex is labeled $(p1x, p2y)$, the top-right vertex is labeled $(p2x, p2y)$, the bottom-left vertex is labeled $(p1x, p1y)$, and the bottom-right vertex is labeled $(p2x, p1y)$. Lines connect the vertices to form the rectangle's perimeter.

Subclasses

```
class V1Rectangle extends Rectangle {
  public void drawLine(double x1, double y1,
    double x2, double y2)
  { d1.draw_a_line(x1,y1,x2,y2); }

  public V1Rectangle(double x1, double y1,
    double x2, double y1, D1 d)
  { super(x1,y1,x2,y2); d1 = d; }
  private D1 d1; }

class V2Rectangle extends Rectangle {
  ... d2.drawln(x1,x2,y1,y2) ...}

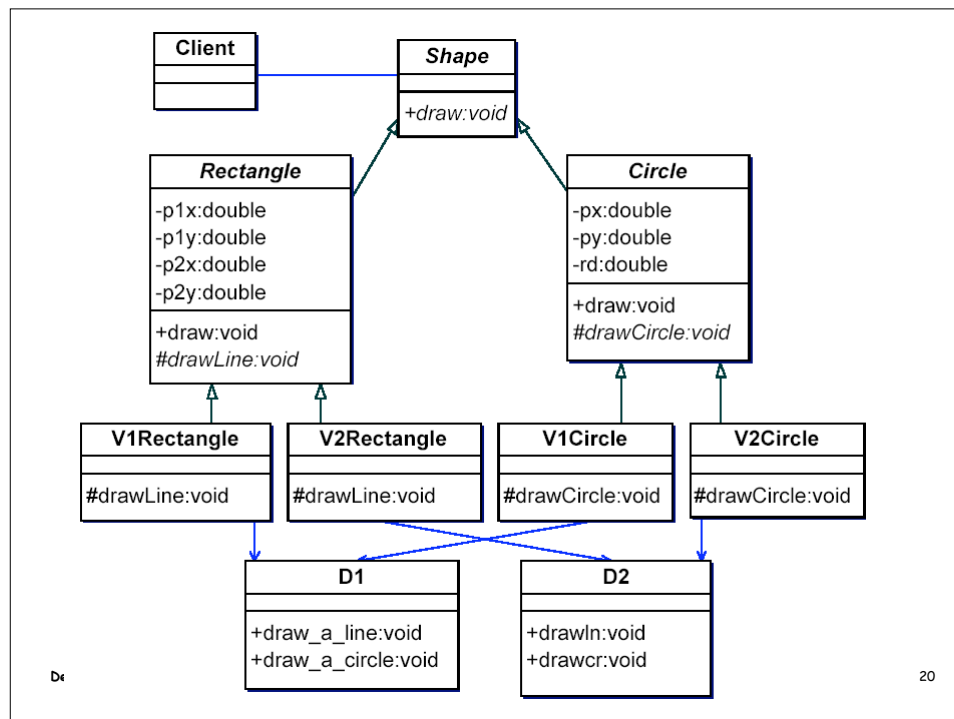
```

Change in Requirements

- The boss wants circles
- Client code should not make a distinction between circles and rectangles w.r.t. drawing
- Solution: superclass **Shape**, subclasses **Rectangle** and **Circle**
- All existing client code is rewritten to use **Shape**

Design Patterns-10, CS431 F06, BG Ryder/A Rountev

19



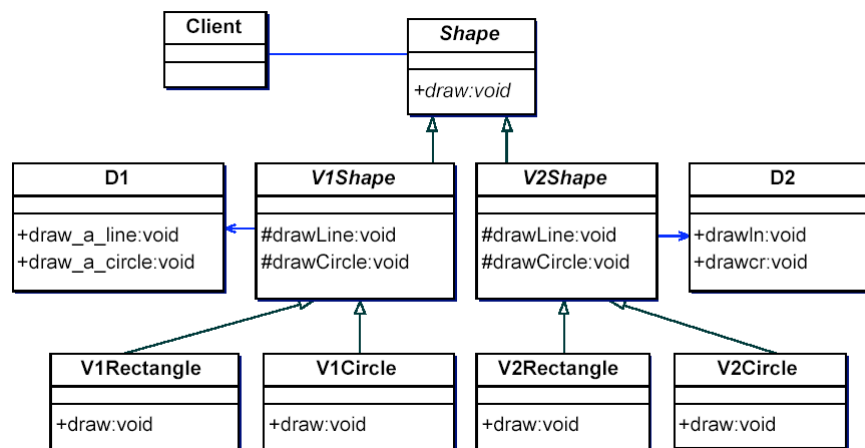
D1

20

Problems

- Too many classes
 - What if there is another drawing class?
And two other kinds of shapes?
 - Total of $3 \times 4 = 12$ classes
- Many classes are coupled to D1 and D2
 - What if the interface of D1 changed?

Possible Alternative Solution



Still problems: # classes, coupling

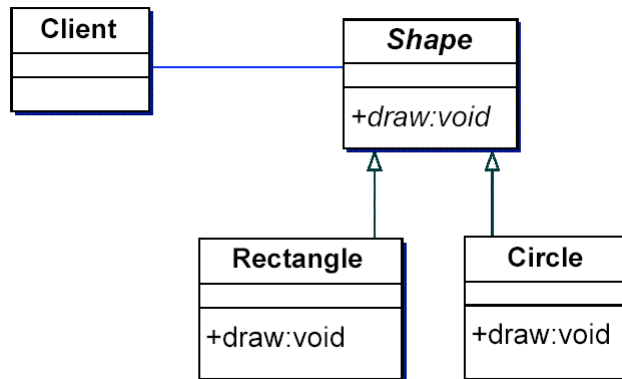
Deeper Problem

- Overuse of inheritance
- Two separate dimensions of variability
 - Kinds of shapes
 - Implementations for aspects of the display
- Common mistake: using inheritance when there are better solutions
- Solution: consider **object composition** instead of inheritance

Bridge Pattern

- Consider the two dimensions separately
- Connect ("bridge") them with composition
- Kinds of shapes
 - **Shape** is an abstract concept
 - Abstract class
 - Subclasses **Rectangle** and **Circle**
 - Each shape is responsible for drawing itself
 - Abstract method **draw** in **Shape**
 - Normal methods **draw** in the subclasses

Dimension 1



Design Patterns-10, CS431 F06, BG Ryder/A Rountev

25

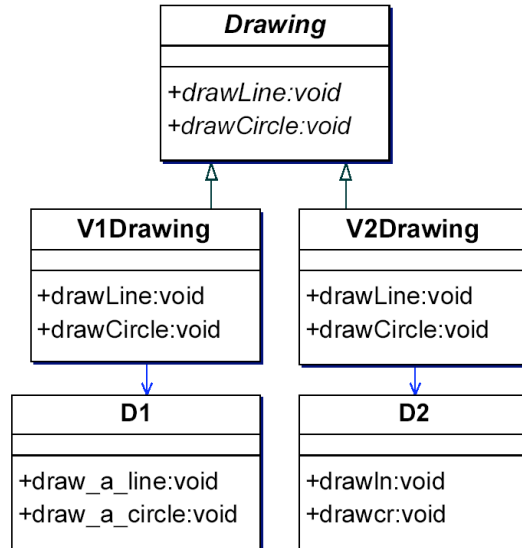
Drawing

- **Kinds of drawing**
 - Abstract class **Drawing**
 - Subclasses **V1Drawing** and **V2Drawing**
- **Each drawing is responsible for knowing how to draw lines and circles**
 - Abstract methods **drawLine()** and **drawCircle()** in **Drawing**
 - Normal methods in the subclasses
- **Association with the appropriate D1/D2**

Design Patterns-10, CS431 F06, BG Ryder/A Rountev

26

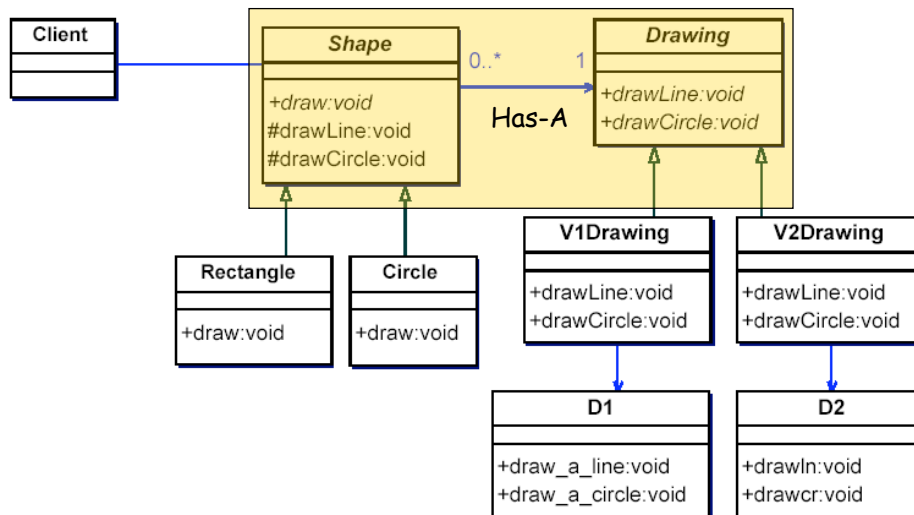
Dimension 2



Design Patterns-10, CS431 F06, BG Ryder/A Rountev

27

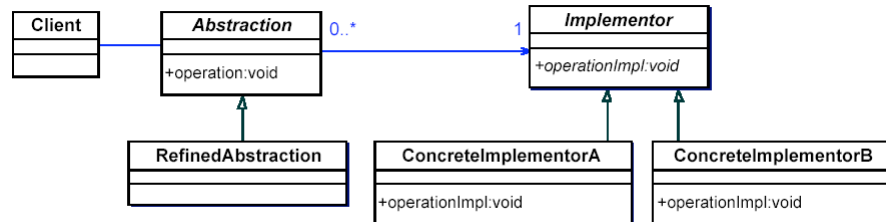
The Bridge



Design Patterns-10, CS431 F06, BG Ryder/A Rountev

28

GoF Formulation



- Decouple an **abstraction** from its **implementation** so that the two can vary independently
- operation calls operationImpl

Design Patterns-10, CS431 F06, BG Ryder/A Rountev

29

Some Options for Implementor Creation

- How, when, and where is the Implementor “hooked up” to the Abstraction? Examples:
- Option 1: Constructors of Abstraction decide, based on input parameters
 - e.g. container: if size < 4 use linked list, otherwise use a hash table
- Option 2: Default implementor, plus changes based on runtime usage
 - If a container grows too much -> switch
- Option 3: Factory pattern (more later)

Design Patterns-10, CS431 F06, BG Ryder/A Rountev

30

Facade Pattern

- A large subsystem, many classes
- Simplified view for the clients
 - High-level interface that is easier to use
 - e.g. we have a 3D drawing library, but we only want 2D, for a subset of the functionality
- Typical reasons
 - Use only a subset of the capabilities
 - Use in a particular "specialized" way
 - Reduce coupling with client code
- Often accessed via Singleton

Observations

- Possibly does some work, but mostly uses the existing classes
- Limits access to underlying classes
 - If necessary, allows **complete** hiding, to reduce coupling, or to track all accesses
- Ease of maintenance
- Facade vs. Adapter -- Differences?
 - Both have pre-existing classes and use indirection
 - Adapter has interface client needs; Façade does not
 - Façade is free to define interface; Adapter is not
 - Key difference is their intent

Summary - Structural Patterns

- How are classes and objects composed to form larger structures?
 - Adapter: two incompatible interfaces
 - Bridge: independence of an abstraction from some implementation aspect
 - Facade: simpler interface
- Other structural patterns
 - Composite, Decorator, Flyweight, Proxy (GoF)
- Onto creational patterns