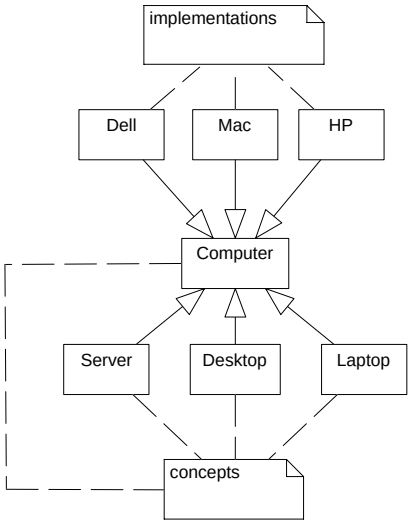
The Bridge Pattern

Use the **Bridge Pattern** to vary not only your implementations, but also your abstractions.

Toni Sellarès
Universitat de Girona

The Bridge Pattern: motivation

Inheritance hierarchies can be hierarchies of abstarctions (concepts) or hierarchies of implementations. For example:



Mixing concepts with implementations can lead to hierarchies that are confusing and difficult to maintain.

It could lead to an exponential number of subclasses and soon we will have an exploding class hierarchy.

The Bridge Pattern

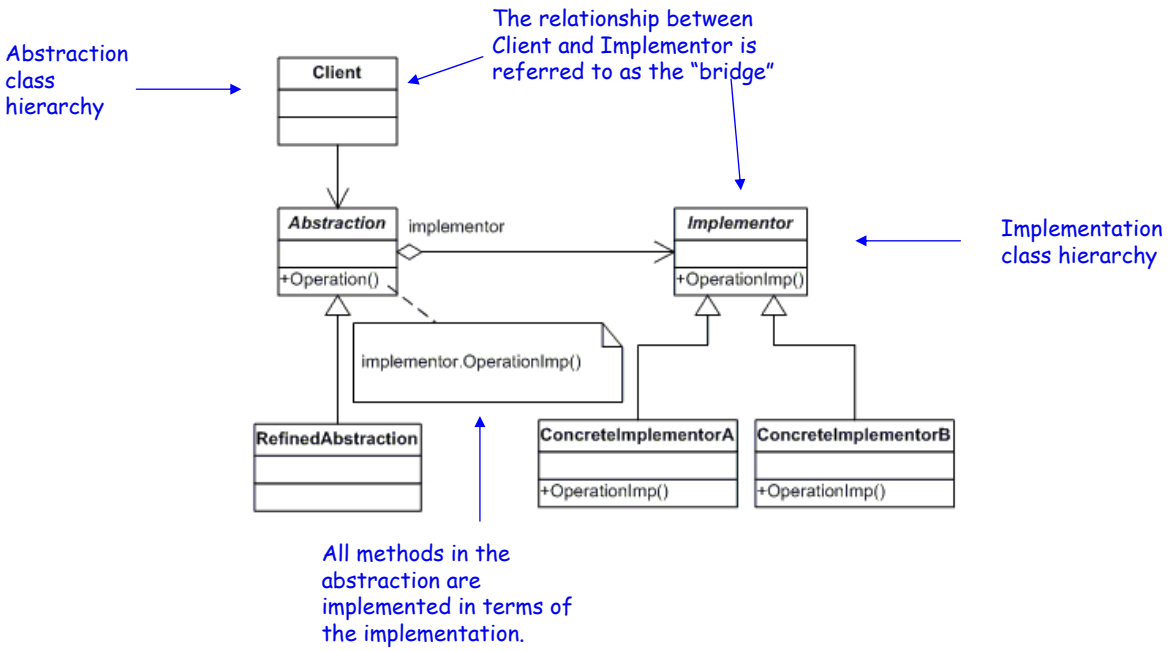
The Bridge pattern is designed to separate a class's interface from its implementation so you can vary or replace the implementation without changing the client code.

The participants in the Bridge pattern are:

- **Abstraction**
  - defines the abstraction's interface
  - maintains a reference to the Implementor
  - forwards requests to the Implementor (collaboration)
- **RefinedAbstraction**
  - extends abstraction interface
- **Implementor**
  - defines interface for implementations
- **ConcreteImplementor**
  - implements Implementor interface, ie defines an implementation

The Bridge Pattern

Use the **Bridge Pattern** to vary not only your implementations, but also your abstractions.



Bridge: Structural Example

```
/**
 * Test driver for the pattern.
 */
public class Test{
          public static void main( String arg[] )
          {
                    Abstraction abstraction = new RefinedAbstraction();
                    Implementor implementor = new ConcreteImplementorB();
                    abstraction.setImplementor( implementor );
                    abstraction.operation();
          }
}
/**
 * Extends the interface defined by Abstraction.
 */
public class RefinedAbstraction extends Abstraction{
          public void operation()
          {
                    super.operation();
          }
}
```

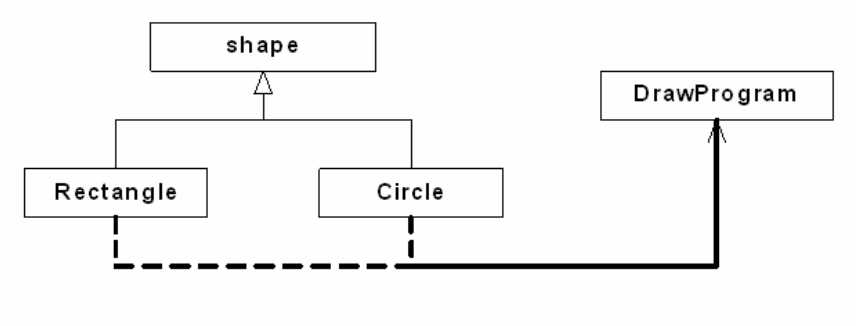
```
/**
 * Defines the interface for implementation classes. This
 * interface doesn't have to correspond exactly to Abstraction's
 * interface; in fact the two interfaces can be quite different.
 * Typically the Implementor interface provides only primitive
 * operations, and Abstraction defines higher-level operations
 * based on these primitives.
 */
public interface Implementor
{
          void operation();
}

/**
 * Implements the Implementor interface and defines its
 * conrete implementation.
 */
public class ConcreteImplementorA implements Implementor
{
          public void operation()
          {
          }
}
```

```
/**
 * Implements the Implementor interface and defines its
 * conrete implementation.
 */
public class ConcreteImplementorB implements Implementor
{
          public void operation()
          {
          }
}


/**
 * Defines the abstraction's interface. Maintains a reference
 * to an object of type Implementor.
 */
public class Abstraction
{
          private Implementor implementor;
          public void setImplementor( Implementor implementor )
          {
                    this.implementor = implementor;
          }
          public Implementor getImplementor()
          {
                    return implementor;
          }
          public void operation()
          {
                    implementor.operation();
          }
}
```

The Bridge Pattern: example 1

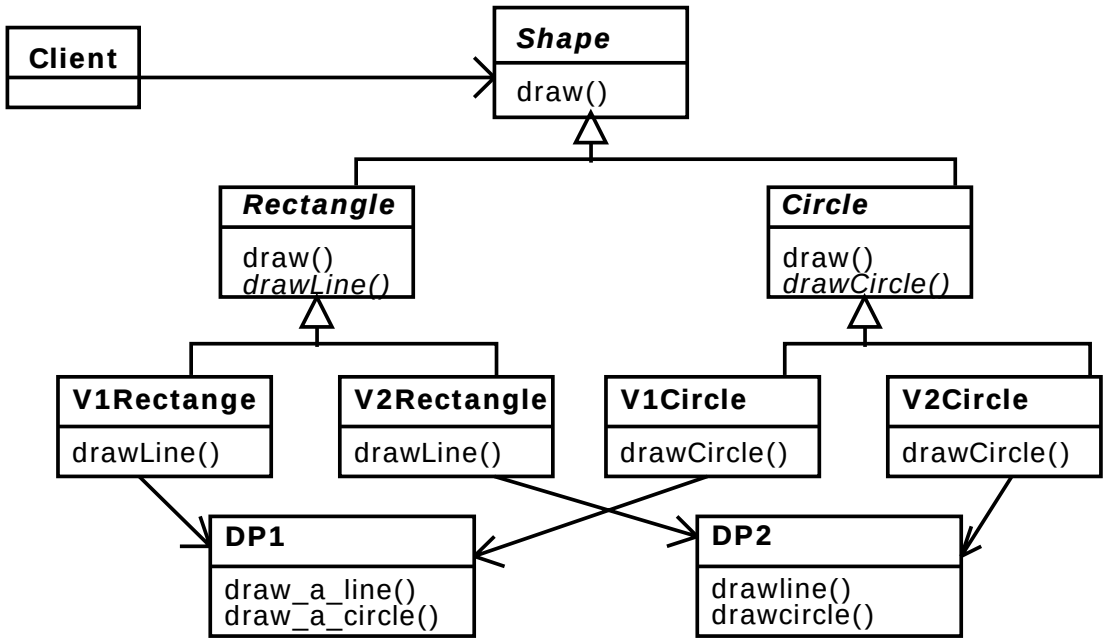


- The shape interface provides an abstract interface for handling shapes.
- The Drawing Program is used by both Rectangle and Circle to implement the Draw() method.
  - Abstraction and implementation are coupled.

Update the design to support 2 Drawing Programs:

  - DP1 and DP2 that have different interfaces.

Solution: derive two classes from each Rectangle and Circle

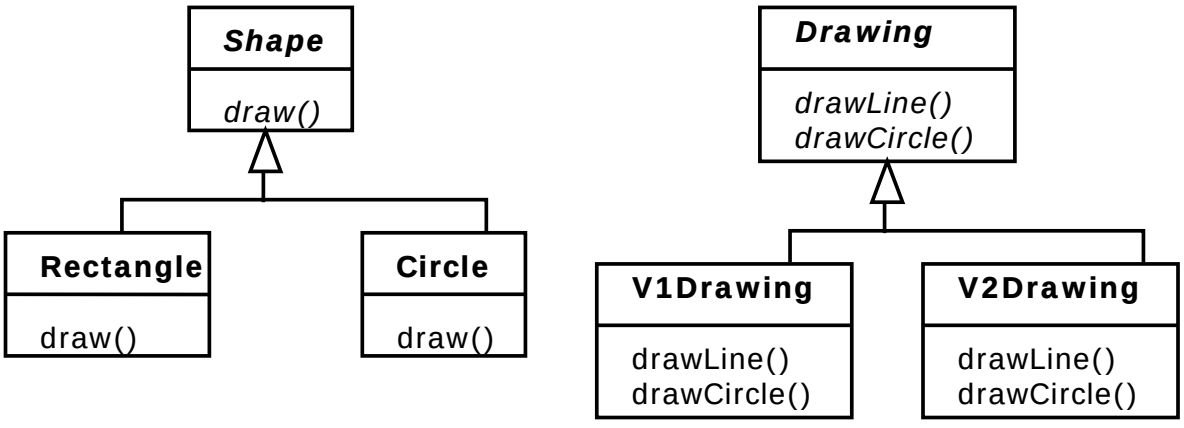


The class explosion problem arises because the abstraction (kinds of Shapes) and the implementation (drawing programs) are tightly coupled: each type of shape must know what type of drawing program it is using.

Update design to be scalable by separating abstraction from implementation**.**

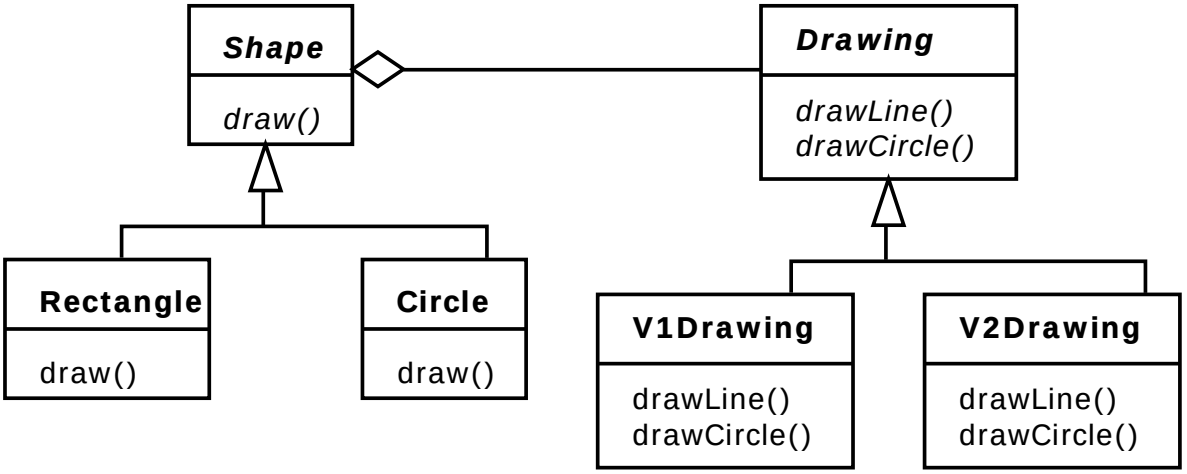
OO Principle: find what varies and encapsulate it

- The abstract classes **Shape** and **Drawing** encapsulate the specific variations.

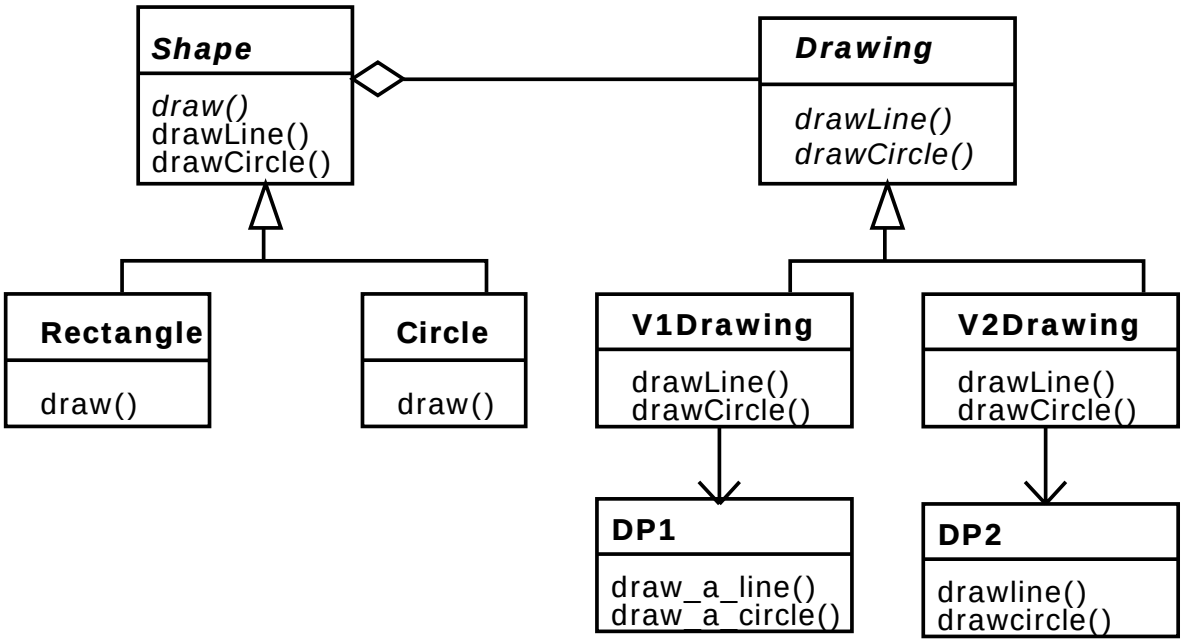


OO Principle: use composition instead of inheritance.

- Shape uses Drawing.



Addition of further dependencies

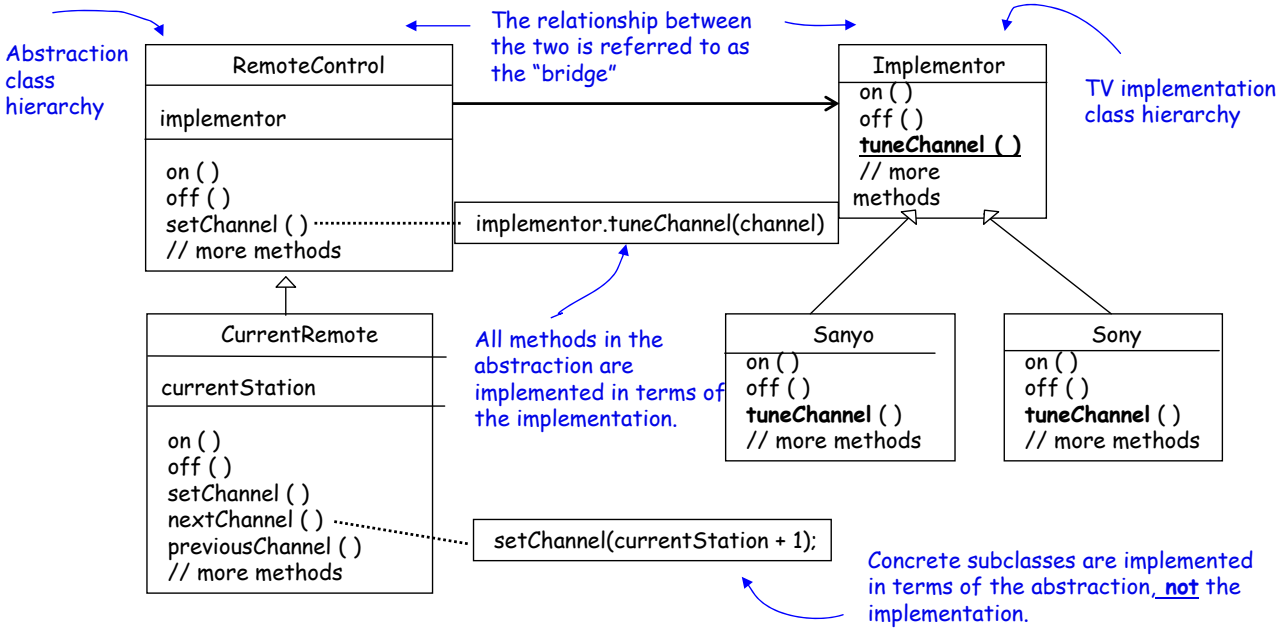


- Shape contains an abstract interface to Drawing.
- The derivatives of Shape no longer dependent on the drawing program.
- Drawing interface uses adaptors to wrap Drawing DP1 and DP2.

- Note that the solution presented integrates the Adapter pattern with the Bridge pattern. This is caused by the fact that the interfaces of DP1 and DP2 have to be adapted to the interface needed.

- While it is very common to see the Adapter pattern incorporated into the Bridge pattern, the Adapter pattern is not part of the Bridge pattern.

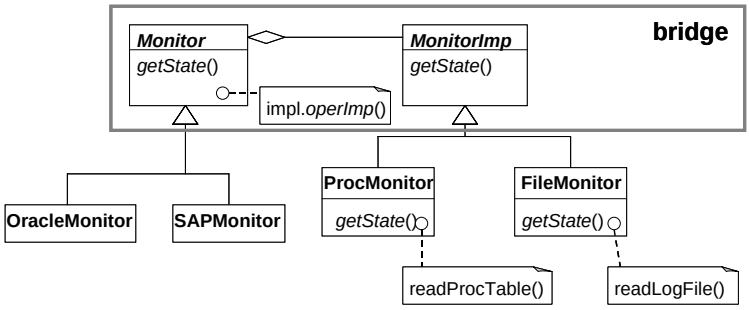
The Bridge Pattern: example 2



The Bridge Pattern: example 3

Often monitors for specific system components are based on non object-oriented techniques, e.g. parsing and filtering log files or accessing internal runtime information by operating system calls.

Here we depict two concrete monitor implementations for accessing the process table of the operating system (ProcMonitor) and for accessing status logs (FileMonitor) and Via the bridge and the abstract monitor class specific monitors, e.g. for the Oracle DB software can use the available implementations.



The Bridge Pattern

Use Bridge pattern when:

- Desire to avoid a permanent binding between an abstraction and implementation
- When abstractions and implementations should be extensible through subclassing.
- When clients should be protected from Implementation changes.
- When the implementation should be completely hidden from the client.
- If there is a proliferation of classes.

Adapter vs. Bridge

- **Adapter**
  - Adapts **existing** classes to an expected interface.
  - The interface and classes exist; the new thing is the adapter, a go–between class.
- **Bridge**
  - Creates an abstract Interface **to be** implemented by multiple classes.
  - keeps the implementations separate from the interface.
  - The new thing is the interface.

Adapter vs. Bridge

- The adapter pattern is geared towards making **unrelated** components work together.
  - Applied to systems after they're designed (**reengineering**, interface engineering).
- A bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.
  - **Ab initio engineering** of an "extensible system".