CS 619 Introduction to OO Design and Development

GoF Patterns (Part 1)

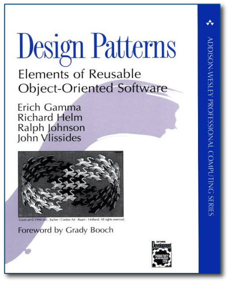
Fall 2012

Review: Design Patterns are NOT

- Designs that can be encoded in classes and reused as is (i.e., linked lists, hash tables)
- Complex domain-specific designs (for an entire application or subsystem)

- They are:
  - "Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."

GoF Design Patterns


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- Each pattern has four essential elements:
  - Pattern name
  - Problem
  - Solution
  - Consequences

Pattern Name

- A handle used to describe:
  - a design problem
  - its solutions
  - its consequences
- Increases design vocabulary
- Makes it possible to design at a higher level of abstraction
- Enhances communication
- *"The Hardest part of programming is coming up with good variable [function, and type] names."*

Problem

- Describes when to apply the pattern
- Explains the problem and its context
- May describe specific design problems and/or object structures
- May contain a list of preconditions that must be met before it makes sense to apply the pattern

Solution

- Describes the elements that make up the
  - design
  - relationships
  - responsibilities
  - collaborations
- Does not describe specific concrete implementation
- Abstract description of design problems and how the pattern solves it

Consequences

- Results and trade-offs of applying the pattern
- Critical for:
  - evaluating design alternatives
  - understanding costs
  - understanding benefits of applying the pattern
- Includes the impacts of a pattern on a system's:
  - flexibility
  - extensibility
  - portability

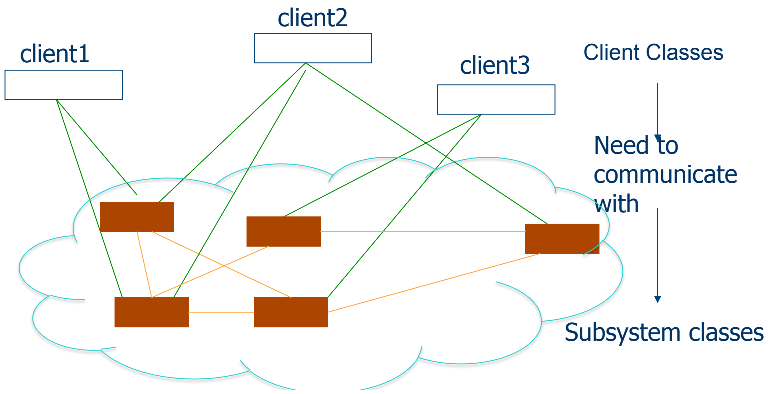
Design Space for GoF Patterns

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

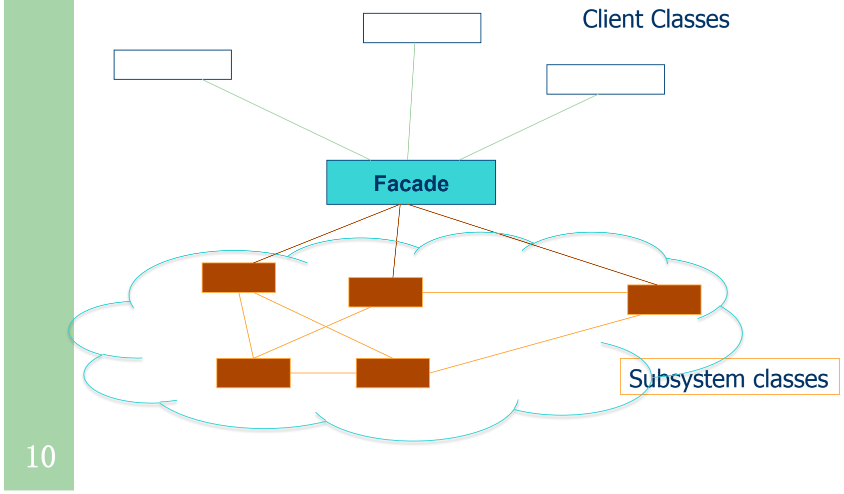
Scope: domain over which a pattern applies
Purpose: reflects what a pattern does

Facade Pattern: Problem

- Provide a unified interface to a set of interfaces in a subsystem.
- Facade Pattern defines a higher-level interface that makes the subsystem easier to use.



Facade Pattern: Solution



Facade Pattern

Why?

- Subsystems often get complex as they evolve.
- Need to provide a simple interface to many, often small, classes. But not necessarily to ALL classes of the subsystem.

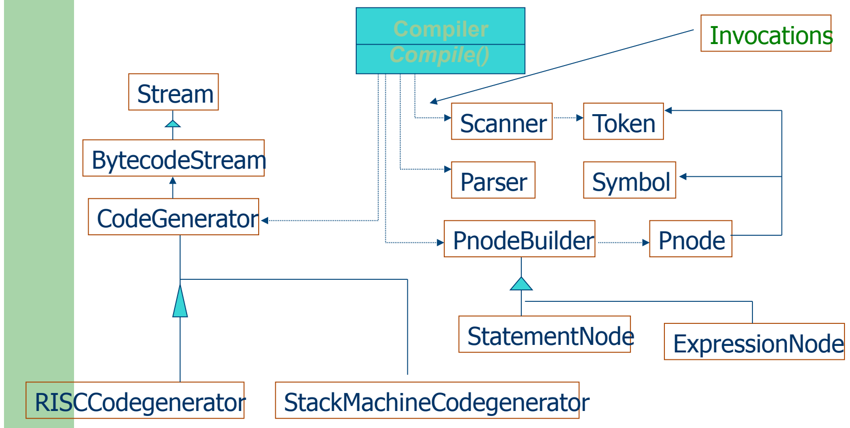
Benefits:

- Facade provides a simple default view good enough for most clients.
- Facade decouples a subsystem from its clients.
- A facade can be a single entry point to each subsystem level. This allows layering.

Facade Pattern: Participants and Communication

- Participants: Facade and subsystem classes
- Clients communicate with subsystem classes by sending requests to facade.
- Facade forwards requests to the appropriate subsystem classes.
- Clients do not have direct access to subsystem classes.

Example: A compiler



Example

```
class Compiler {            // Facade. Offers a simple interface to compile and
  public:                   // Generate code.
            Compiler();
            virtual void Compile (istream&, BytecodeStream&);

}
void Compiler:: Compile (istream& input, BytecodeStream& output) {
            Scanner scanner (input);
            PnodeBuilder builder;
            Parser parser;
            parser.Parse (scanner, builder);
            RISCCodeGenerator generator (output);
            Pnode* parseTree = builder.GetRootNode();
  }         parseTree→Traverse (generator);
```

Façade Pattern in Java API

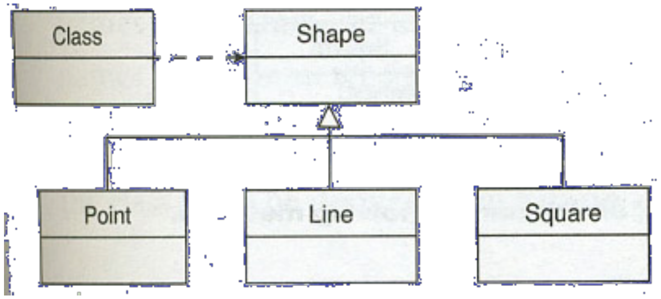
- ExternalContext behaves as a facade for performing cookie, session scope and similar operations.
- Underlying classes it uses are HttpSession, ServletContext, javax.servlet.http.HttpServletRequest and javax.servlet.http.HttpServletResponse.

Common Mistakes:

- Facade layer should not be forced and its always optional. Clients should be allowed to bypass the facade layer and interact with components directly.
- Methods in facade layer should not contain only one or two lines which calls the other components.
- Facade is ‘not’ a layer that imposes security and hides important data and implementation.
- Subsystems are not aware of facade and there should be no reference for facade in subsystems.

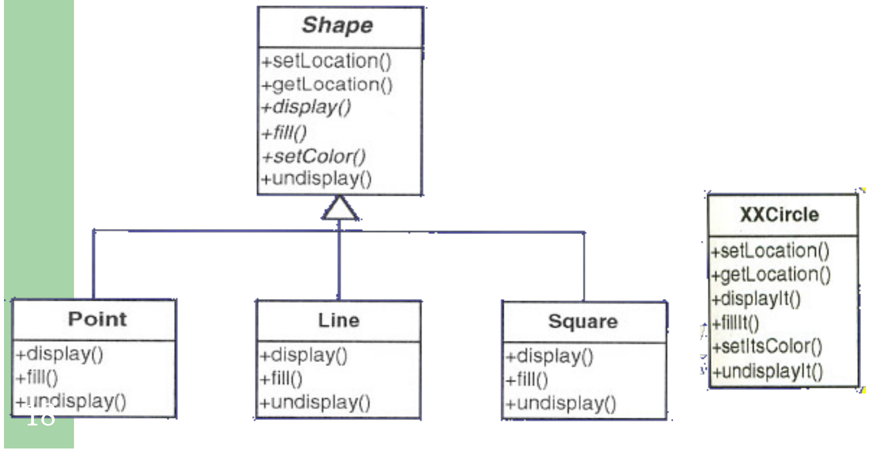
Adapter Pattern: Problem

Example: We need to create a shape class and have the concrete classes of point, line, and square derive from shape as in the following figure:



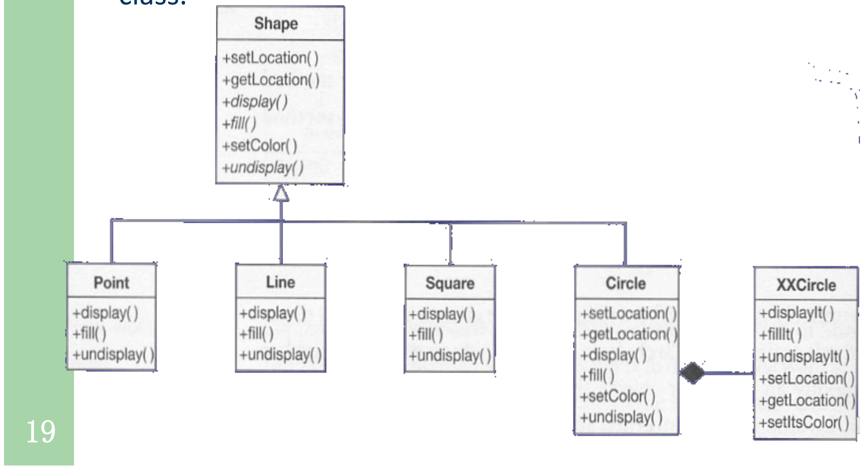
Adapter Pattern

We define a series of behaviors that all Shapes will have in common in the Shape class and then override their behavior in the concrete classes of Point, Line, and Square.



Solution:

Create a Circle object that encapsulates XXCircle by making an XXCircle object an attribute of the Circle class.



Code Fragment:

```
class Circle extends Shape {
   …
   private XXCircle myXXCircle;
   …
   public Circle () {
        myXXCircle = new XXCircle();
   }

   void public display() {
         myXXCircle.displayIt();
   }
   …
}
```

Adapter Pattern

Intent: Match an existing object beyond your control to a particular interface

Problem: A system has the right data and behavior, but the wrong interface.

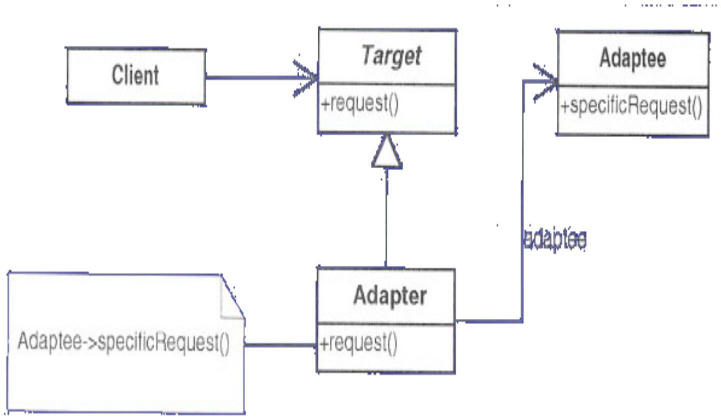
Solution: Provides a wrapper with the desired interface.

P & C: Adapters Target (the class it derives from). Allows the Client to use the Adaptee as if it were a type of Target.

Consequences: Allows for preexisting objects to fit into new class structures.

Implementation: Contain the existing class in another class. Have the containing class match the required interface and call the methods of the contained class.

Adapter Pattern Structure



Adapter v.s. Facade

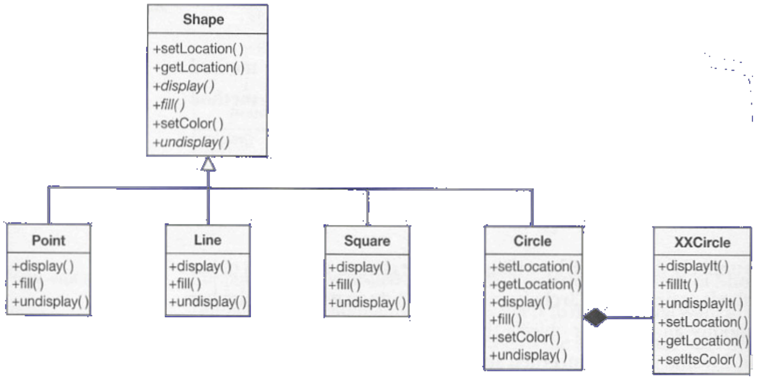
	Façade	Adapter
Are there preexisting classes?	Yes	Yes
Is there an interface we MUST design to?	No	Yes
Does an object need to behave polymorphically?	No	Probably
Is a simpler interface needed?	Yes	No

Encapsulation

Traditional view of encapsulation: data hiding.

Board viewpoint: Encapsulation is any kind of hiding.

You can hide:



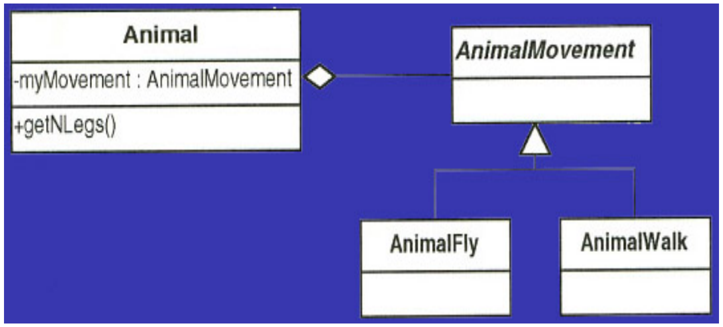
Exercise:

Consider the traditional problem of creating classes that represent animals. Your requirements are:

- Each type of animal can have a different number of legs
- Animal objects can retrieve this information.
- Each type of animal can have a different type of movement. E.g. walking, fyling…
- Animal objects must be able to return how long it will take to move from one place to another given a certain type of terrain.

Find what is varying and encapsulate it

- Works with many variations are present.
- It is better to have a data member that indicates the type of movement the object has.



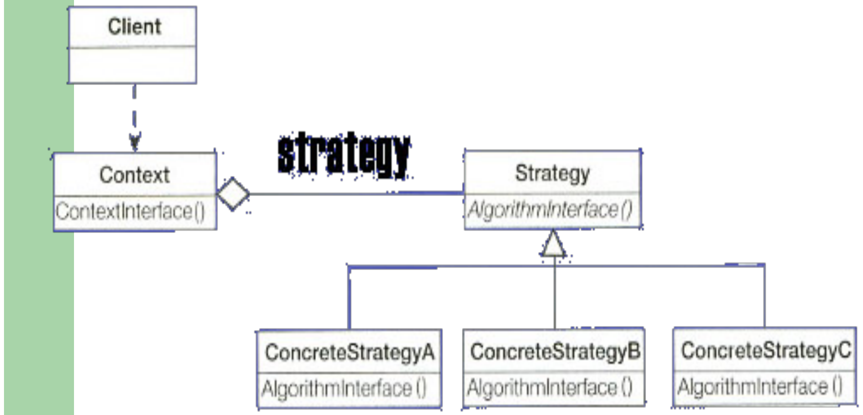
Strategy Pattern

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

The Strategy Pattern is based on a few principles:

- Objects have responsibilities
- Different specific implementations of these responsibilities are manifested through the use of polymorphism
- There is a need to manage several different implementations of the same basic algorithm.

Strategy Pattern



Strategy Pattern

Intent: Enable you to use different business rules or algorithms depending on the context in which they occur.

Problem: The selection of an algorithm that needs to be applied depends on the client making the request or the data being acted on.

Solution: Separate the selection of the algorithm from the implementation of the algorithm.

Strategy Pattern

Participants and collaborators:

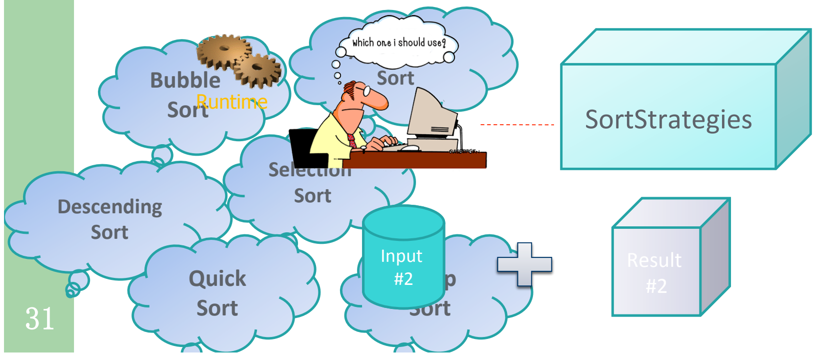
- **Strategy** specifies how the different algorithms are used.
- **ConcreteStrategies** implement these different algorithms.
- **Context** uses a specific **ConcreteStrategies** with a reference of type **Strategy**. **Strategy** and **Context** interact to implement the chosen algorithm. The **Context** forwards request form its client to **Strategy**.

Implementation:

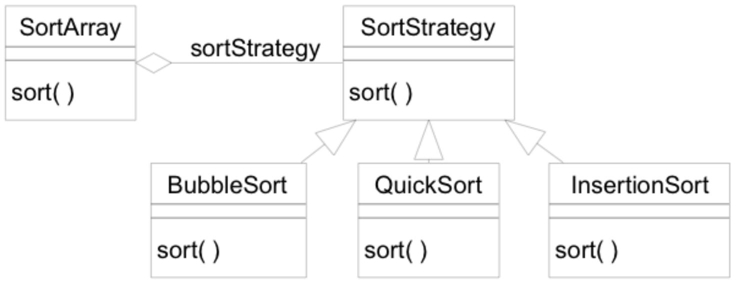
- Have the class that uses the algorithm (Context) contain an abstract class (Strategy) that has an abstract method specifying how to call the algorithm.
- Each derived class implements the algorithm needed.

Example : Sort

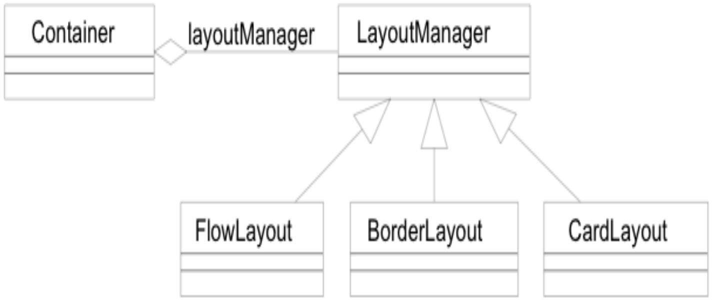
- Assume that, you need to write a sort program that will have an array and at run-time you want to decide which sort algorithm to use.
- Strategy is what we group the many algorithms that do the same things and make it interchangeable at run-time



Class Diagram of Sort Example



Another Example: Class Diagram of Layout



Container
layoutManager
LayoutManager
FlowLayout
BorderLayout
CardLayout