SOLID Principles and Design Patterns

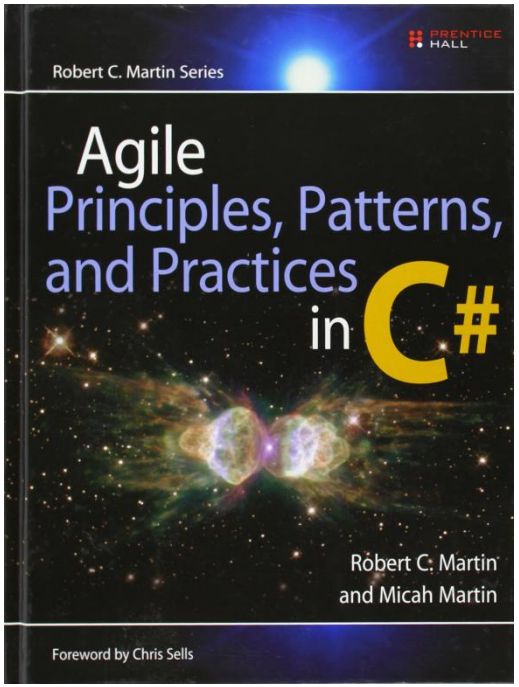
Lecture 7

Objectives

- SOLID Principles
  - Single Responsibility Principle (SRP)
  - Open Closed Principle (OCP)
  - Liskov Substitution Principle (LSP)
  - Interface Segregation Principle (ISP)
  - Dependency Inversion Principle (DIP)
- Design Patterns
  - Creational Design Patterns: Builder, Singleton
  - Structural Design Patterns: Adapter, Composite
  - Behavioural Design Patterns: Observer, Visitor

History of SOLID

- Robert C. Martin is the main person behind these ideas (some individual ideas predate him though).
- First appeared as a news group posting in 1995.
- Full treatment given in Martin and Martin, *Agile Principles, Patterns, and Practices in C#*, Prentice Hall, 2006. (The PPP book)
- Lots of online learning material (http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod)



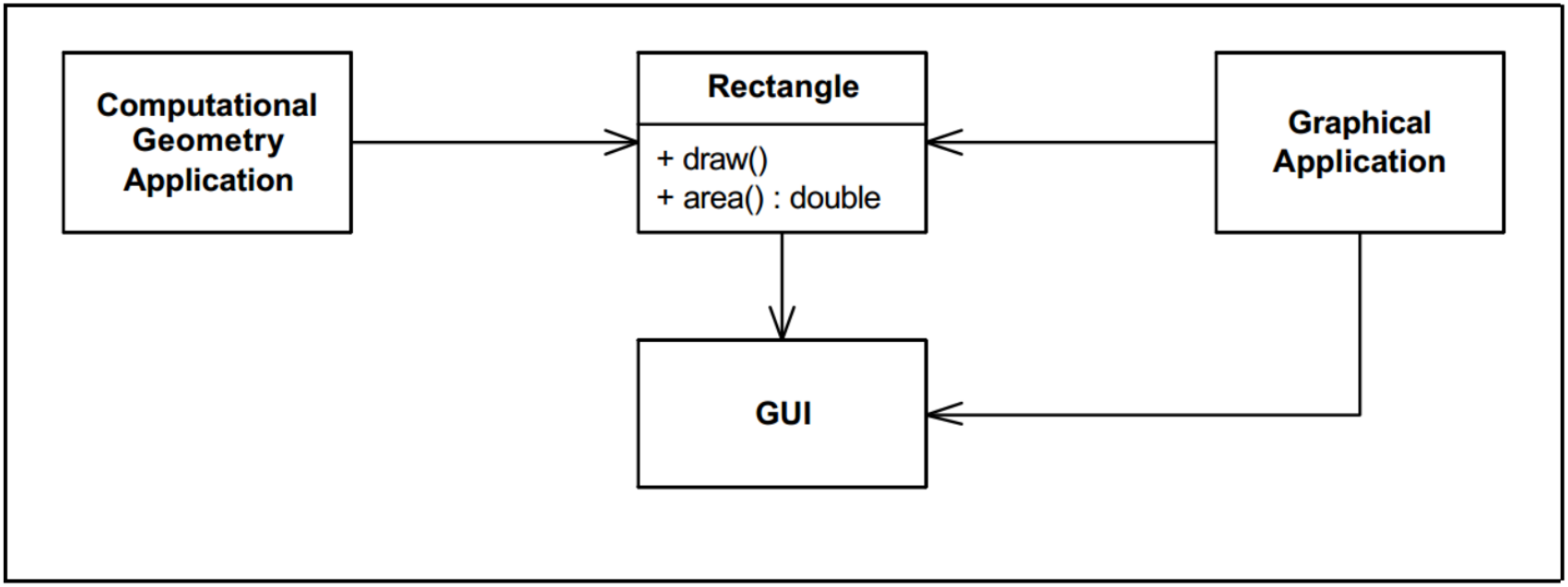
Benefits of SOLID

- Provides a principled way to manage dependency.
- Serves as a solid foundation for OO development upon which more complicated design patterns can be built upon and incorporated naturally.
- Results in code that are flexible, robust, and reusable.

First Pass at Understanding SOLID

- Single Responsibility Principle:
  - A class should have one, and only one, reason to change
- Open Closed Principle:
  - You should be able to extend a class's behaviour, without modifying it
- Liskov Substitution Principle:
  - Derived classes must be substitutable for their base classes
- Interface Segregation Principle:
  - Make fine grained interfaces that are client specific
- Dependency Inversion Principle:
  - Depend on abstractions, not on concretions

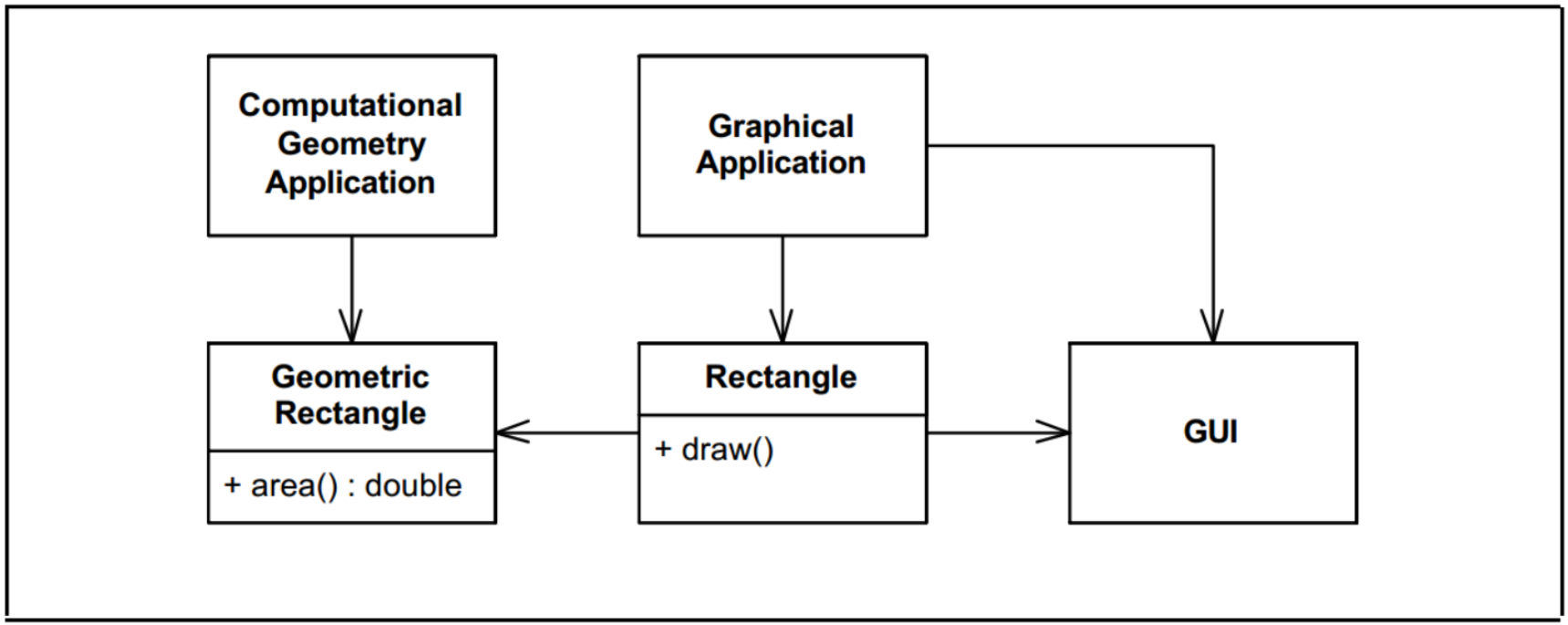
Single Responsibility Principle



Single Responsibility Principle

- *Rectangle* class with draw() and area() methods
- *Computation Geometry App* now depends on *GUI* via *Rectangle*
- Any changes to *Rectangle* due to *Graphical App* necessitates rebuild of *Computational Geometry App*

Single Responsibility Principle



Single Responsibility Principle

- Solution: Take purely computational part of the *Rectangle* class and create new class *GeometricRectangle* class
- All changes regarding graphical display can then be localized into the *Rectangle* class

Single Responsibility Principle

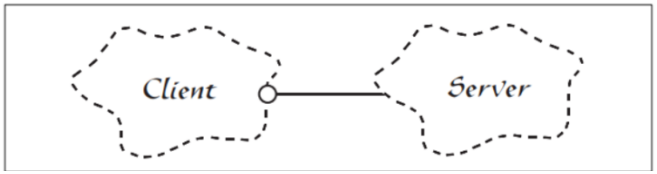
- Another Example
- Modem: dial(), hangup(), send(), recv()
- However, there are two separate kinds of functions that can change for different reasons:
  - Connection-related
  - Data communication-related
- These two should be separated.
- Recall that “Responsibility” == “a reason to change”.

Open-Closed Principle

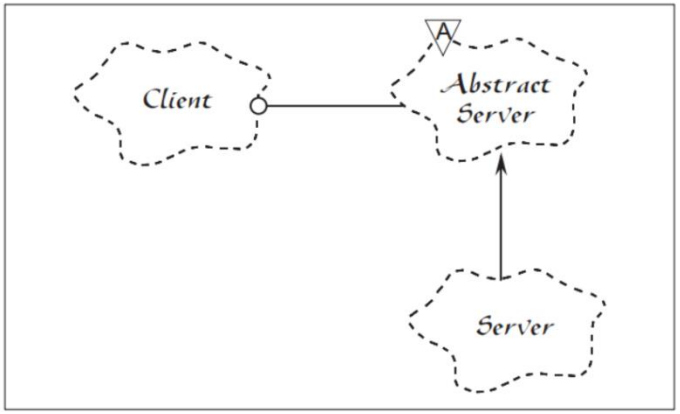
- All systems change during their life cycles.
- Software entities should be open for extension, but closed for modification.
- Goal: avoid a “cascade of changes to dependent modules”.
- When requirements change, you extend the behavior, not changing old code

Open-Closed Principle

- Bad Design: need to change client code when new kinds of server needed

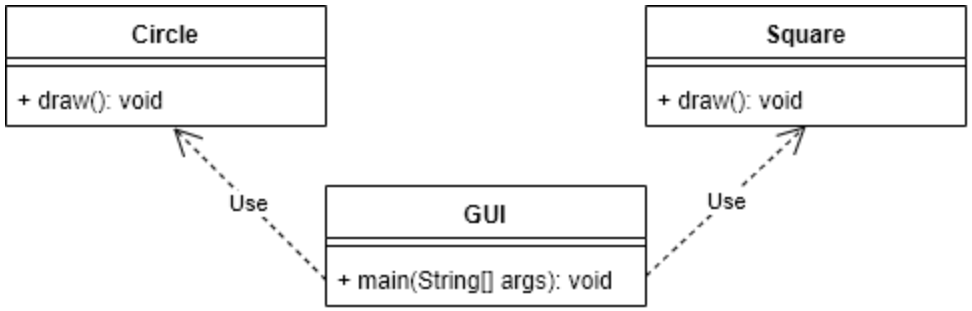


- Good design: can extend to new types of servers without modifying client code



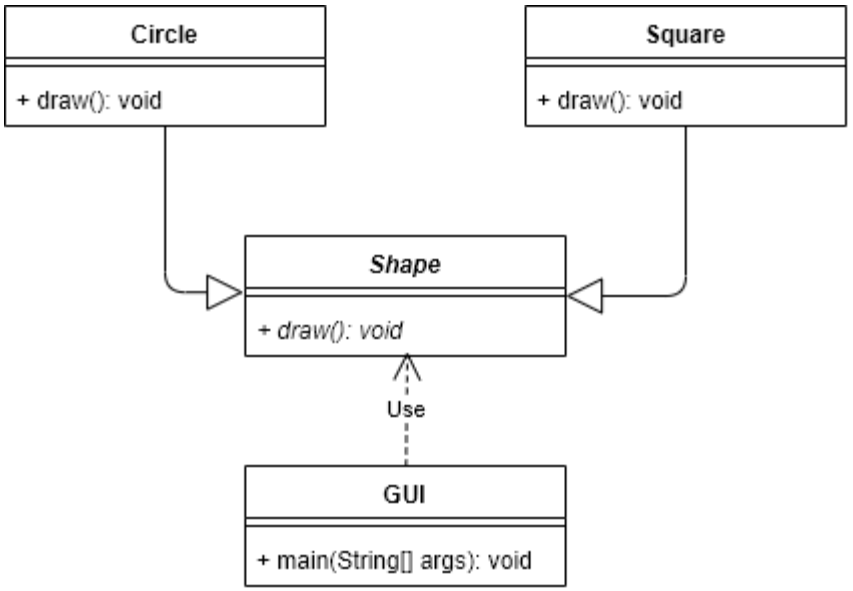
Open-Closed Principle

- Example: There are two classes *Circle* and *Square* to be drawn inside application
- *GUI* uses both of them to draw on computer screen
- *GUI* is not closed because any new shape will require modification



Open-Closed Principle

- Solution: *GUI* has to interact with an abstract *Shape* class for drawing different shapes.



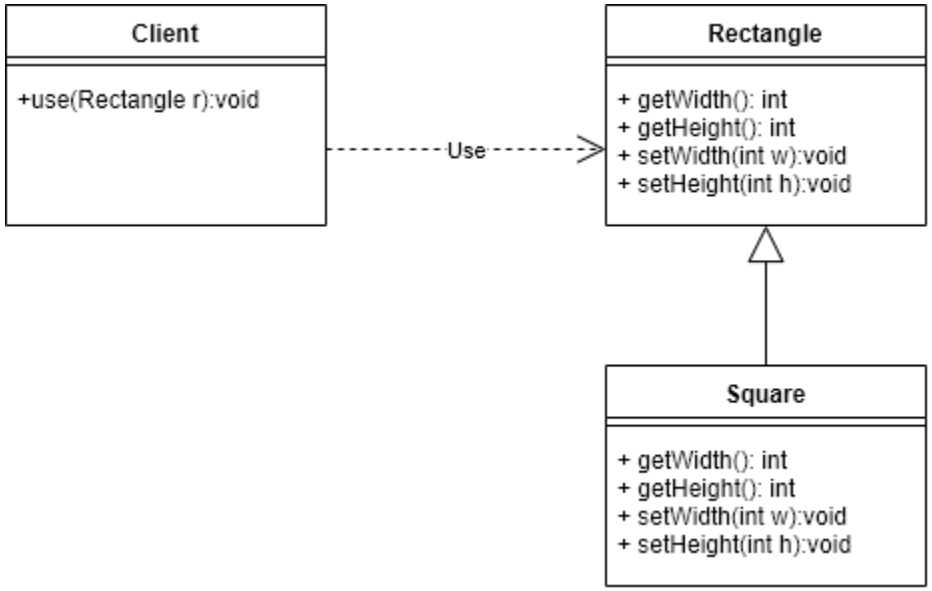
Liskov Substitution Principle

- "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it." (original idea due to Barbara Liskov)
- Violation means the user class's need to know ALL implementation details of the derived classes of the base class
- Violation of Liskov Substitution Principle leads to violation of Open-Closed Principle

Liskov Substitution Principle

- Example: *Square* class inherits from *Rectangle* class, but *setWidth()* and *setHeight* methods of *Rectangle* are not good fit for *Square* class
- When *Square* class is used where *Rectange* class is called for, behaviour can be unpredictable, depending on implementation
- Want either *setWidth()* or *setHeight()* to set both width and height in the Square class
- LSP is violated when adding a derived class requires modifications of the base class.

Liskov Substitution Principle

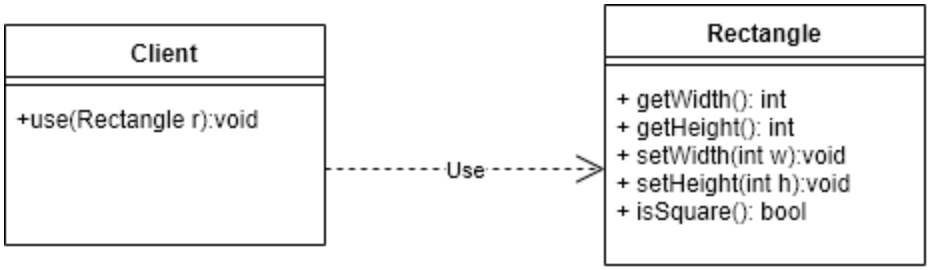


Liskov Substitution Principle

- Cannot assess validity of a class by just looking inside a class. We must see how it is used.
- *Square* is a *Rectangle,* but they behave differently, seen from the outside.
- For LSP to hold, ALL derived classes should conform to the behavior that the clients expect of the base classes.

Liskov Substitution Principle

- Solution:
  - Remove *Square* class because it modifies the behavior of its base class, thus violating LSP.
  - Inside *Rectangle* class create *isSquare()* method.

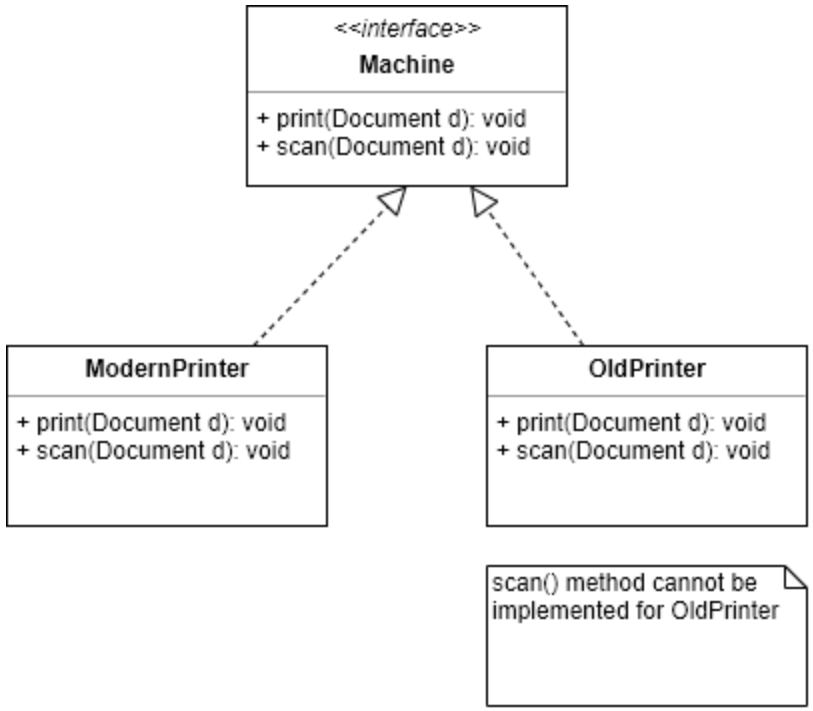


Interface Segregation Principle

- Clients should not be forced to depend upon interfaces that they do not use
- Avoid fat interfaces
- Fat interfaces: interfaces of a class that can be broken down into groups that serve different set of clients
- Clients depending on a subset of interfaces need to change when other clients using a different subset changes.

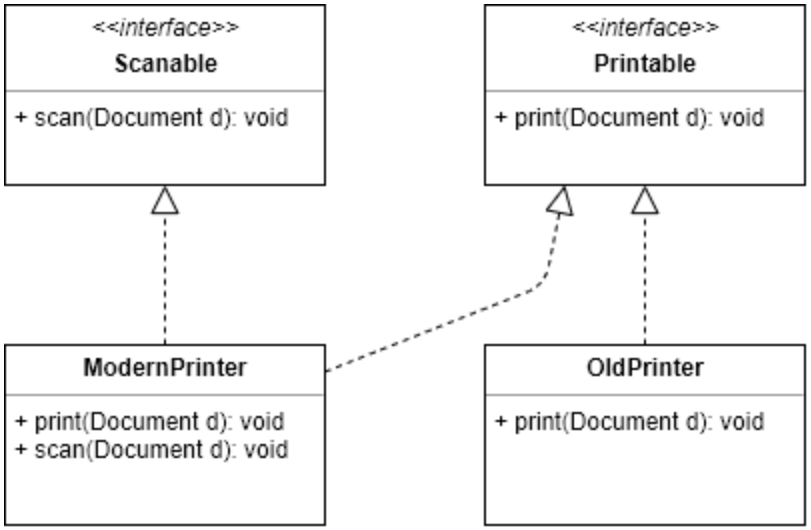
Interface Segregation Principle

- Example: *Machine* interface contains methods that cannot be implemented by some derived classes.



Interface Segregation Principle

- Solution: *Machine* interface should be broken down into smaller interfaces.



Dependency Inversion Principle

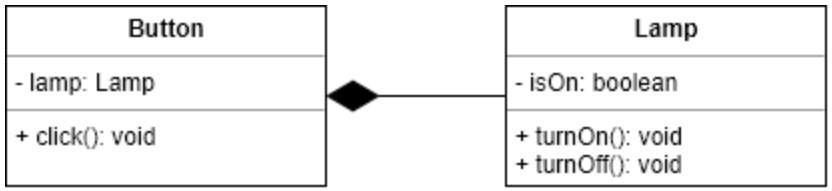
- **A)** High level modules should not depend upon low level modules. Both should depend upon abstractions.
- **B)** Abstractions should not depend upon details. Details should depend upon abstractions.
- “Inversion”, because standard structured programming approaches make the higher level depend on lower level.

Dependency Inversion Principle

- Bad Design:
  - Hard to change (rigidity)
  - Unexpected parts break when changing code (fragility)
  - Hard to reuse (immobility)
- Causes of bad design:
  - Interdependence of the modules
  - Things can break in areas with NO conceptual relationship to the changed part
  - Dependent on unnecessary detail

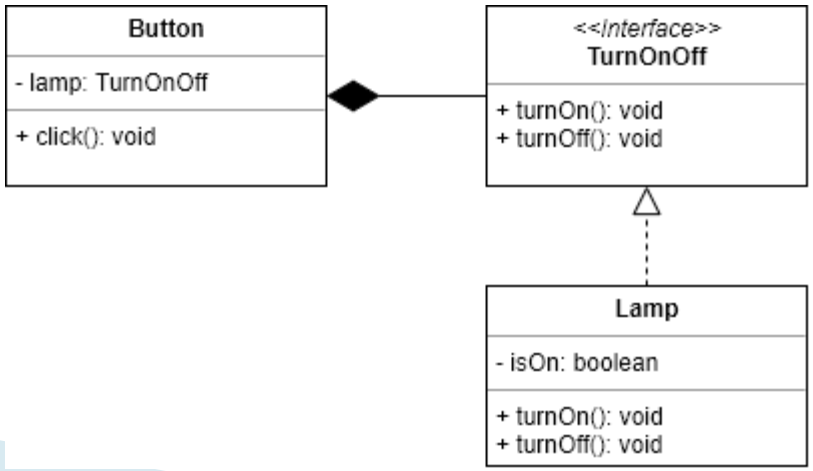
Dependency Inversion Principle

- Example: *Button* class is composed of a concrete *Lamp* class which makes it tightly coupled to this particular implementation of *Lamp.*
- *Button* (high-level module) depends on *Lamp (low-level module)*



Dependency Inversion Principle

- Solution: *Button* class is composed of an abstract *TurnOnOff()* inteface which makes it loosely coupled to *Lamp class.*
- *Button* (high-level module) depends on abstraction
- *Lamp* (low-level module) implements abstraction



Intro to Design Patterns

- “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” – Christopher Alexander
- Design patterns capture the best practices of experienced object-oriented software developers.
- Design patterns are solutions to general software development problems.

Attributes Design Patterns

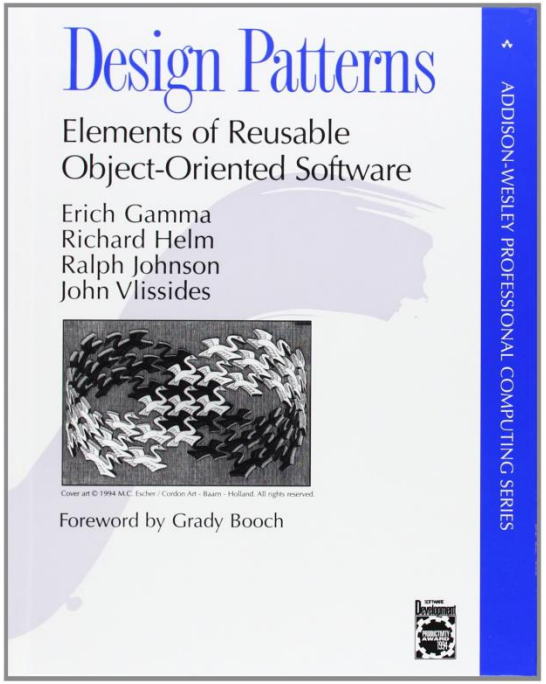
- **Pattern Name and Classification**: A descriptive and unique name that helps in identifying and referring to the pattern.
- **Intent**: A description of the goal behind the pattern and the reason for using it.
- **Also Known As**: Other names for the pattern.
- **Motivation (Forces)**: A scenario consisting of a problem and a context in which this pattern can be used.
- **Applicability**: Situations in which this pattern is usable; the context for the pattern.
- **Structure**: A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.

Attributes Design Patterns

- **Participants**: A listing of the classes and objects used in the pattern and their roles in the design.
- **Collaboration**: A description of how classes and objects used in the pattern interact with each other.
- **Consequences**: A description of the results, side effects, and trade offs caused by using the pattern.
- **Implementation**: A description of an implementation of the pattern; the solution part of the pattern.
- **Sample Code**: An illustration of how the pattern can be used in a programming language.
- **Known Uses**: Examples of real usages of the pattern.
- **Related Patterns**: Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

Design Patterns by Gang of Four

- Erich Gamma, Richard Helm, Ralph Johnson and John Vlisides in their Design Patterns book define 23 design patterns divided into three types:
- *Creational patterns* are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.
- *Structural patterns* help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.
- *Behavioral patterns* help you define the communication between objects in your system and how the flow is controlled in a complex program.



Creational Patterns

- **Abstract Factory:**
  - Factory for building related objects
- **Builder:**
  - Factory for building complex objects incrementally
- **Factory Method:**
  - Method in a derived class creates associates
- **Prototype:**
  - Factory for cloning new instances from a prototype
- **Singleton:**
  - Factory for a singular (sole) instance

Structural Patterns

- **Adapter:**
  - Translator adapts a server interface for a client
- **Bridge**:
  - Abstraction for binding one of many implementations
- **Composite**:
  - Structure for building recursive aggregations
- **Decorator**:
  - Decorator extends an object transparently
- **Facade**:
  - Simplifies the interface for a subsystem
- **Flyweight**:
  - Many fine-grained objects shared efficiently.
- **Proxy**:
  - One object approximates another

Behavioural Patterns

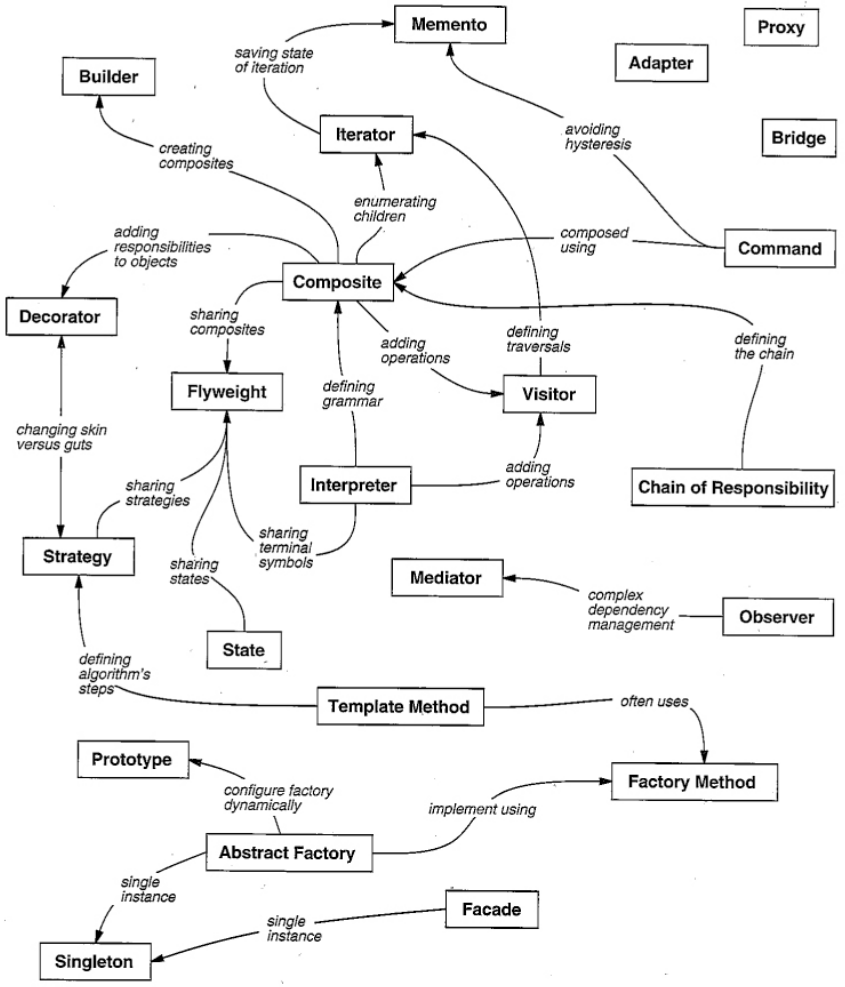
- **Chain of Responsibility**:
  - Request delegated to the responsible service provider
- **Command**:
  - Request or Action is first-class object, hence re-storable
- **Iterator**:
  - Aggregate and access elements sequentially
- **Interpreter**:
  - Language interpreter for a small grammar
- **Mediator**:
  - Coordinates interactions between its associates
- **Memento**:
  - Snapshot captures and restores object states privately

Behavioural Patterns

- **Observer**:
  - Dependents update automatically when subject changes
- **State**:
  - Object whose behavior depends on its state
- **Strategy**:
  - Abstraction for selecting one of many algorithms
- **Template Method**:
  - Algorithm with some steps supplied by a derived class
- **Visitor**:
  - Operations applied to elements of a heterogeneous object structure

23 Design Patterns

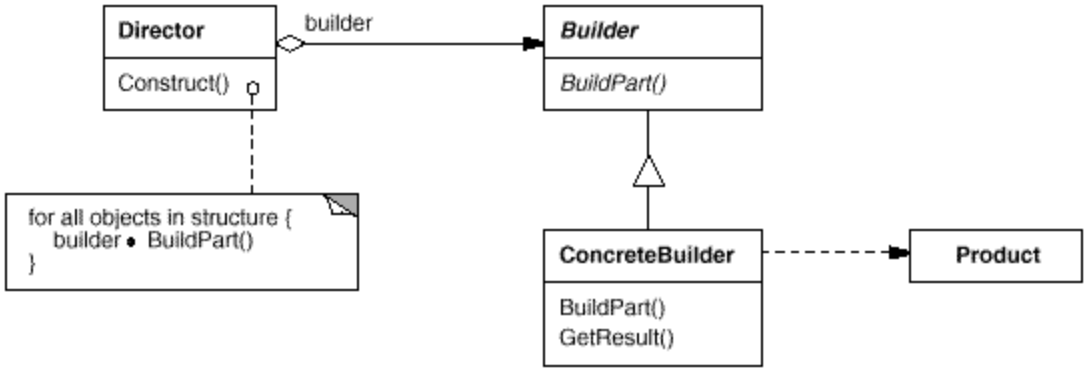
- Consider how design patterns solve design problems.
- Scan intent sections.
- Study how patterns interrelate.
- Study patterns of like purpose.
- Examine a cause of redesign.
- Consider what should be variable in your design.



Design Pattern Relationships

The Builder Pattern

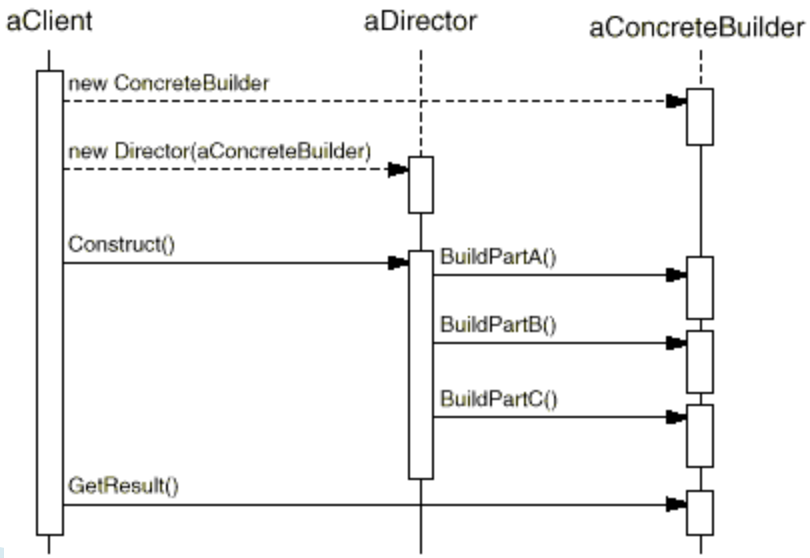
- **The Builder Pattern** separates the construction of a complex object from its representation so that the same construction process can create different representations.



- **Builder** – specifies an abstract interface for creating parts of a **Product** object.
- **ConcreteBuilder** – constructs and assembles parts of the product by implementing the Builder interface. Also, it defines and keeps track of the representation it creates and provides an interface for retrieving the product .
- **Director** – constructs an object using the Builder interface.
- **Product** – represents the complex object under construction.

The Builder Pattern

- The client creates the Director object and configures it with the desired Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.



The Builder Pattern

- Use the Builder pattern when:
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled.
  - The construction process must allow different representations for the object that is constructed.

The Builder Pattern

- Consequences:
  - A Builder lets you vary the internal representation of the product it builds. It also hides the details of how the product is assembled.
  - Each specific builder is independent of the others and of the rest of the program. This improves modularity and makes the addition of other builders relatively simple.
  - Because each builder constructs the final product step-by-step, depending on the data, you have more control over each final product that a Builder constructs.
  - A Builder pattern is somewhat like an Abstract Factory pattern in that both return classes made up of a number of methods and objects. The main difference is that while the Abstract Factory returns a family of related classes, the Builder constructs a complex object step by step depending on the data presented to it.

Builder Example – Pizza Builder

```
/** "Product" */
class Pizza {
  private String dough = "";
  private String sauce = "";
  private String topping = "";
  public void setDough(String dough) {
    this.dough = dough; }
  public void setSauce(String sauce) {
    this.sauce = sauce; }
  public void setTopping(String topping) {
    this.topping = topping; }
}
```

Builder Example – Pizza Builder

```
/** "Abstract Builder" */
 abstract class PizzaBuilder {
    protected Pizza pizza;
    public Pizza getPizza() {
      return pizza; }
    public void createNewPizzaProduct() {
      pizza = new Pizza(); }
    public abstract void buildDough();
    public abstract void buildSauce();
    public abstract void buildTopping();
 }
```

Builder Example – Pizza Builder

```
/** "ConcreteBuilder" */
 class HawaiianPizzaBuilder extends PizzaBuilder {
    public void buildDough()    {
      pizza.setDough("cross"); }
    public void buildSauce()    {
      pizza.setSauce("mild"); }
    public void buildTopping() {
      pizza.setTopping("ham+pineapple"); }
 }
 /** "ConcreteBuilder" */
 class SpicyPizzaBuilder extends PizzaBuilder {
    public void buildDough()    {
      pizza.setDough("pan baked"); }
    public void buildSauce()    {
      pizza.setSauce("hot"); }
    public void buildTopping() {
      pizza.setTopping("pepperoni+salami"); }
 }
```

Builder Example – Pizza Builder

```
/** "Director" */
class Waiter {
   private PizzaBuilder pizzaBuilder;
   public void setPizzaBuilder(PizzaBuilder pb) {
     pizzaBuilder = pb; }
   public Pizza getPizza() {
     return pizzaBuilder.getPizza(); }
   public void constructPizza() {
      pizzaBuilder.createNewPizzaProduct();
      pizzaBuilder.buildDough();
      pizzaBuilder.buildSauce();
      pizzaBuilder.buildTopping();
   }
}
```

Builder Example – Pizza Builder

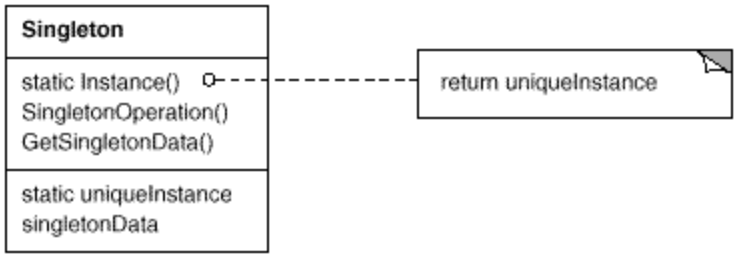
```
 /** A customer ordering a pizza. */
class BuilderExample {
   public static void main(String[] args) {
        Waiter waiter = new Waiter();
        PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
        PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

        waiter.setPizzaBuilder( hawaiianPizzaBuilder );
        waiter.constructPizza();

        Pizza pizza = waiter.getPizza();
   }
}
```

The Singleton Pattern

- Sometimes it is appropriate to have exactly one instance of a class:
  - window managers,
  - print spoolers,
  - filesystems.
- Typically, those types of objects known as singletons, are accessed by disparate objects throughout a software system, and therefore require a global point of access.
- The Singleton pattern addresses all the concerns above. With the Singleton design pattern you can:
  - Ensure that only one instance of a class is created.
  - Provide a global point of access to the object.
  - Allow multiple instances in the future without affecting a singleton class' clients.

The Singleton Pattern

- The Singleton pattern ensures a class has only one instance, and provides a global point of access to it.
- The class itself is responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance.
- Singletons maintain a static reference to the sole singleton instance and return a reference to that instance from a static instance() method.



Singleton Example

```
public class ClassicSingleton {
        private static ClassicSingleton instance = null;

        protected ClassicSingleton() {
              // exists only to defeat instantiation.
        }
        public static ClassicSingleton getInstance() {
              if(instance == null) {
              instance = new ClassicSingleton();
              }
              return instance;
        }
}
```

The *ClassicSingleton* class maintains a static reference to the lone singleton instance and returns that reference from the static getInstance() method.

The Singleton Pattern

- The *ClassicSingleton* class employs a technique known as lazy instantiation to create the singleton; as a result, the singleton instance is not created until the *getInstance()* method is called for the first time. This technique ensures that singleton instances are created only when needed.
- The *ClassicSingleton* class implements a protected constructor so clients cannot instantiate *ClassicSingleton* instances; however, the following code is perfectly legal:

```
public class SingletonInstantiator {
    public SingletonInstantiator() {
      ClassicSingleton instance = ClassicSingleton.getInstance();

      ClassicSingleton anotherInstance = new ClassicSingleton();
        ...
    }
}
```

The Singleton Pattern

How can a class that does not extend *ClassicSingleton* create a *ClassicSingleton* instance if the *ClassicSingleton* constructor is protected?

- Protected constructors can be called by subclasses and *by other classes in the same package*. Hence, because *ClassicSingleton* and *SingletonInstantiator* are in the same package (the default package), *SingletonInstantiator()* methods can create *ClassicSingleton* instances.

Solutions:

1. We can make the ***ClassicSingleton* constructor private** so that only ClassicSingleton's methods call it; however, that means *ClassicSingleton* cannot be subclassed. Also, it's a good idea to declare the singleton class **final**, which makes that intention explicit and allows the compiler to apply performance optimizations.
2. We can put your singleton class in an explicit package, so classes in other packages (including the default package) cannot instantiate singleton instances.

The Singleton Pattern

The ClassicSingleton class is not thread-safe.

If two threads – we will call them Thread 1 and Thread 2, call *ClassicSingleton.getInstance()* at the same time, two *ClassicSingleton* instances can be created if Thread 1 is preempted just after it enters the if block and control is subsequently given to Thread 2.

Solution: **Synchronization**

```
public class ClassicSingleton {
    private static ClassicSingleton instance = null;
    private static Object syncObject; // to synchronize a block
    protected ClassicSingleton() {
       /*exists only to defeat instantiation*/ };
      public static ClassicSingleton getInstance() {
              synchronized(syncObject) {
       if (instance == null) instance = new ClassicSingleton();}
         return instance;}
}
```

The Singleton Pattern

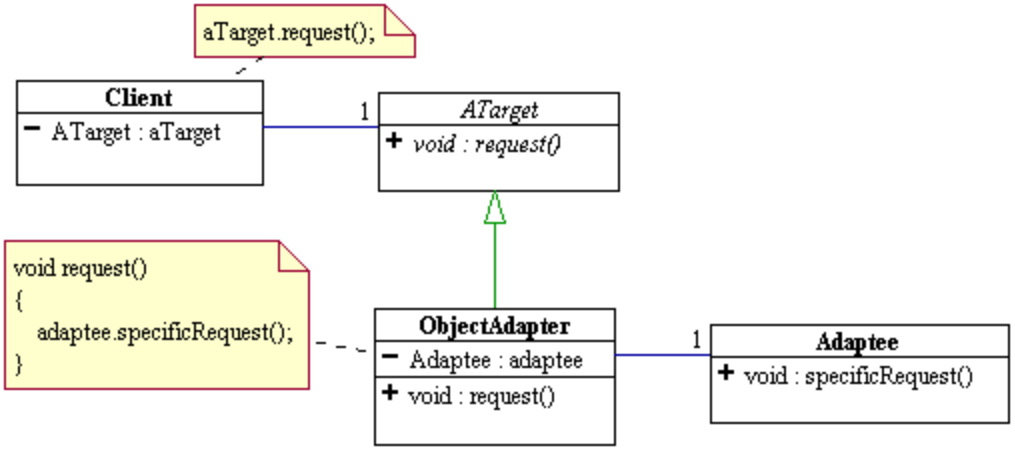
- Consequences:
  - It can be difficult to subclass a Singleton, since this can only work if the base Singleton class has not yet been instantiated.
  - We can easily change a Singleton to allow a small number of instances where this is allowable and meaningful.
  - We can use the same approach to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.
  - The Singleton pattern permits refinement of operations and representation. The Singleton class may be subclassed, and it is easy to configure an application with an instance of this extended class. You can configure the application with an instance of the class you need at run-time.

The Adapter Pattern

- Adapters are used to enable objects with different interfaces to communicate with each other.
- The Adapter pattern is used to convert the programming interface of one class into that of another. We use adapters whenever we want unrelated classes to work together in a single program.
- Adapters come in two flavors, **object adapters and class adapters**.
- The concept of an adapter is thus pretty simple; we write a class that has the desired interface and then make it communicate with the class that has a different interface.
- Adapters in Java can be implemented in two ways: by inheritance, and by object composition.

The Adapter Pattern – Object Adapter

- Object adapters use a compositional technique to adapt one interface to another.
- The adapter inherits the target interface that the client expects to see, while it holds an instance of the adaptee.
- When the client calls the *request()* method on its target object (the adapter), the request is translated into the corresponding specific request on the adaptee.
- Object adapters enable the client and the adaptee to be completely decoupled from each other. Only the adapter knows about both of them.



Object Adapter Example

```
/**
* The SquarePeg class.
* This is the Target class.
*/
public class SquarePeg {
  public void insert(String str) {
    System.out.println("SquarePeg insert(): " + str);}
}

/**
* The RoundPeg class.
* This is the Adaptee class.
*/
public class RoundPeg {
  public void insertIntoHole(String msg) {
   System.out.println("RoundPeg insertIntoHole(): " + msg);}
}
```

If a client only understands the **SquarePeg** interface for inserting pegs using the **insert()** method, how can it insert round pegs, which are pegs, but that are inserted differently, using the **insertIntoHole()** method?

Object Adapter Example

Solution:

Design a **RoundToSquarePeg** adapter that enables to **insertIntoHole()** a **RoundPeg** object connected to the adapter to be inserted as a **SquarePeg**, using **insert()**.

```
/**
* The RoundToSquarePegAdapter class.
* This is the Adapter class.
* It adapts a RoundPeg to a SquarePeg.
* Its interface is that of a SquarePeg.
*/
public class RoundToSquarePegAdapter extends SquarePeg {
  private RoundPeg roundPeg;
  public RoundToSquarePegAdapter(RoundPeg peg) {
    //the roundPeg is plugged into the adapter
    this.roundPeg = peg;}
  public void insert(String str) {
    //the roundPeg can now be inserted in the same manner as a squarePeg!
    roundPeg.insertIntoHole(str);}
}
```

Object Adapter Example

Example:

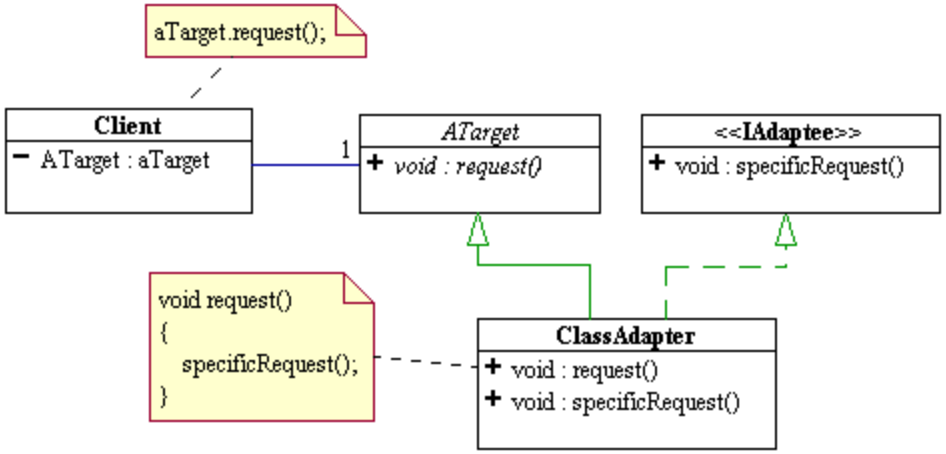
```
// Test program for Pegs.
public class TestPegs {
  public static void main(String args[]) {

    // Create some pegs.
    RoundPeg roundPeg = new RoundPeg();
    SquarePeg squarePeg = new SquarePeg();

    // Do an insert using the square peg.
    squarePeg.insert("Inserting square peg...");

    // Now we'd like to do an insert using the round peg.
    // But this client only understands the insert()
    // method of pegs, not a insertIntoHole() method.
    // The solution: create an adapter that adapts
    // a square peg to a round peg!

    RoundToSquarePegAdapter adapter = new RoundToSquarePegAdapter(roundPeg);
    adapter.insert("Inserting round peg...");}
}
```

Execution trace:

```
SquarePeg insert(): Inserting square peg...
RoundPeg insertIntoHole(): Inserting round peg...
```

The Adapter Pattern – Class Adapter

- Class adapters use multiple inheritance to achieve their goals.
- As in the object adapter, the class adapter inherits the interface of the client's target. However, it also inherits the interface of the adaptee as well.
- Since Java does not support true multiple inheritance, this means that **one of the interfaces must be inherited from a Java Interface type**.
- Both of the target or adaptee interfaces could be Java Interfaces.
- The request to the target is simply rerouted to the specific request that was inherited from the adaptee interface.



Class Adapter Example

Here are the interfaces for round and square pegs:

```
/**
*The IRoundPeg interface.
*/
public interface IRoundPeg {
public void insertIntoHole(String msg);
}

/**
*The ISquarePeg interface.
*/
public interface ISquarePeg {
public void insert(String str);
}
```

Class Adapter Example

Here are the new **RoundPeg** and **SquarePeg** classes. These are essentially the same as before except they now implement the appropriate interface.

```
// The RoundPeg class.
public class RoundPeg implements IRoundPeg {
  public void insertIntoHole(String msg) {
    System.out.println("RoundPeg insertIntoHole(): " + msg);}
}

// The SquarePeg class.
public class SquarePeg implements ISquarePeg {
  public void insert(String str) {
    System.out.println("SquarePeg insert(): " + str);}
}
```

Class Adapter Example

And here is the new **PegAdapter** class:

```
/**
* The PegAdapter class.
* This is the two-way adapter class.
*/
public class PegAdapter implements ISquarePeg, IRoundPeg {
  private RoundPeg roundPeg;
  private SquarePeg squarePeg;

  public PegAdapter(RoundPeg peg) {
    this.roundPeg = peg;}
  public PegAdapter(SquarePeg peg) {
    this.squarePeg = peg;}

  public void insert(String str) {
    roundPeg.insertIntoHole(str);}
  public void insertIntoHole(String msg){
    squarePeg.insert(msg);}
}
```

Class Adapter Example

A client that uses the two-way adapter:

```
// Test program for Pegs.
public class TestPegs {
  public static void main(String args[]) {

    // Create some pegs.
    RoundPeg roundPeg = new RoundPeg();
    SquarePeg squarePeg = new SquarePeg();

    // Do an insert using the square peg.
    squarePeg.insert("Inserting square peg...");

    // Create a two-way adapter and do an insert with it.
    ISquarePeg roundToSquare = new PegAdapter(roundPeg);
    roundToSquare.insert("Inserting round peg...");

    // Do an insert using the round peg.
    roundPeg.insertIntoHole("Inserting round peg...");

    // Create a two-way adapter and do an insert with it.
    IRoundPeg squareToRound = new PegAdapter(squarePeg);
    squareToRound.insertIntoHole("Inserting square peg...");}
}
```

Class Adapter Example

Client program output:

```
SquarePeg insert(): Inserting square peg...
RoundPeg insertIntoHole(): Inserting round peg...
RoundPeg insertIntoHole(): Inserting round peg...
SquarePeg insert(): Inserting square peg...
```

Adapter Pattern – Consequences

Class and object adapters have different trade-offs.

A class adapter:

- **adapts Adaptee to Target by committing to a concrete Adapter class**;
- **lets Adapter override some of Adaptee's behavior**, since Adapter is a subclass of Adaptee;
- **introduces only one object, and no additional indirection is needed to get to the adaptee.**

An object adapter

- **lets a single Adapter work with many Adaptees** - that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
- **makes it harder to override Adaptee behavior**. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

The Composite Pattern

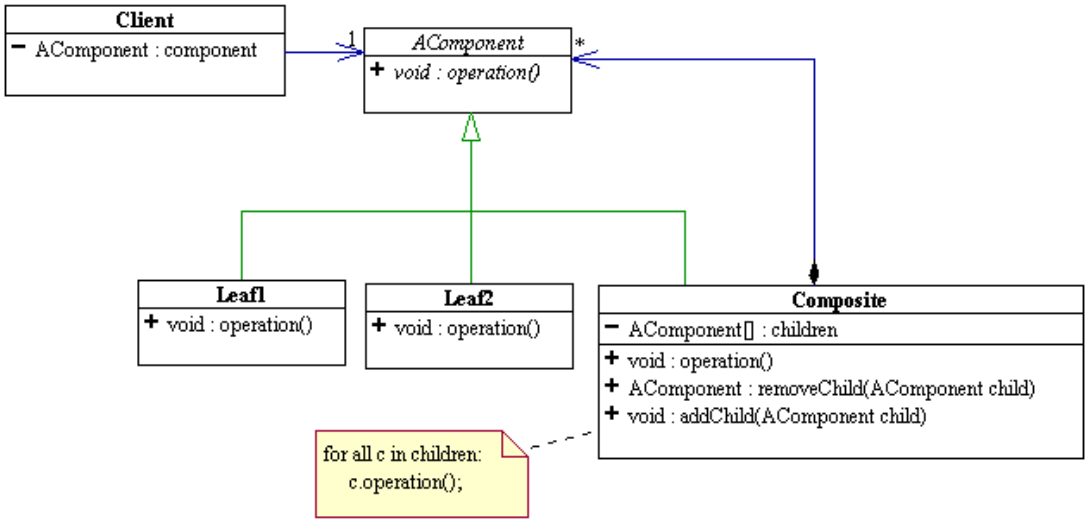
- The **Composite** Design pattern allows a client object to treat both single components and collections of components identically.

- Composite patterns are often used to represent **recursive** data structures. The recursive nature of the Composite structure naturally gives way to recursive code to process that structure.

- Use the Composite pattern when:
  - You want to represent part-whole hierarchies of objects.
  - You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

The Composite Pattern

- The Client uses an abstract component, *AComponent*, for some abstract task, *operation()*.
- At run-time, the Client holds a reference to a concrete component such as Leaf1 or Leaf2.
- When the operation task is requested by the *Client*, the specific concrete behavior with the particular concrete component will be performed.



Composite Pattern Example

```java
/** "Component" */
interface Graphic {
    //Prints the graphic.
    public void print();
}

/** "Composite" */
class CompositeGraphic implements Graphic {
    //Collection of child graphics.
    private final ArrayList<Graphic> childGraphics = new ArrayList<>();

    //Adds the graphic to the composition.
    public void add(Graphic graphic) {
        childGraphics.add(graphic);
    }

    //Prints the graphic.
    @Override
    public void print() {
        for (Graphic graphic : childGraphics) {
            graphic.print();  //Delegation
        }
    }
}

/** "Leaf" */
class Ellipse implements Graphic {
    //Prints the graphic.
    @Override
    public void print() {
        System.out.println("Ellipse");
    }
}
```

Composite Pattern Example

```
/** Client */
public class CompositeDemo {
    public static void main(String[] args) {
        //Initialize four ellipses
        Ellipse ellipse1 = new Ellipse();
        Ellipse ellipse2 = new Ellipse();
        Ellipse ellipse3 = new Ellipse();
        Ellipse ellipse4 = new Ellipse();

        //Creates two composites containing the ellipses
        CompositeGraphic graphic2 = new CompositeGraphic();
        graphic2.add(ellipse1);
        graphic2.add(ellipse2);
        graphic2.add(ellipse3);

        CompositeGraphic graphic3 = new CompositeGraphic();
        graphic3.add(ellipse4);

        //Create another graphics that contains two graphics
        CompositeGraphic graphic1 = new CompositeGraphic();
        graphic1.add(graphic2);
        graphic1.add(graphic3);

        //Prints the complete graphic (Four times the string "Ellipse").
        graphic1.print();
    }
}
```

Composite Pattern – Consequences

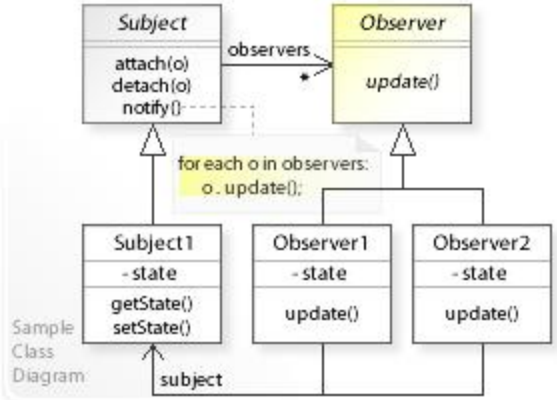
- The Composite pattern allows you to define a class hierarchy of simple objects and more complex composite objects so that they appear to be the same to the client program.
- Because of this simplicity, the client can be that much simpler, since nodes and leaves are handled in the same way.
- The Composite pattern also makes it easy for you to add new kinds of components to your collection, as long as they support a similar programming interface.
- The composite is essentially a singly-linked tree, in which any of the objects may themselves be additional composites.

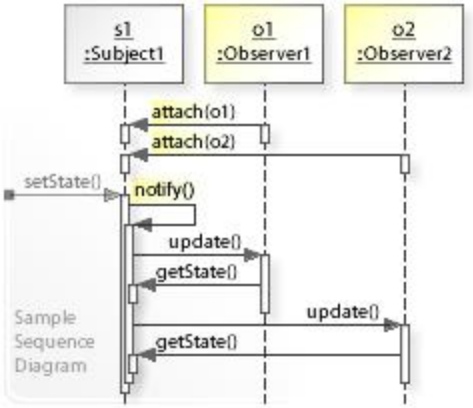
The Observer Pattern

- The cases when certain objects need to be informed about the changes occured in other objects are frequent. To have a good design means to decouple as much as possible and to reduce the dependencies. The Observer Design Pattern can be used whenever a subject has to be observed by one or more observers.
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- This pattern is a cornerstone of the **Model-View-Controller** (MVC) architectural design, where the Model implements the mechanics of the program, and the Views are implemented as Observers that are as much uncoupled as possible to the Model components.

The Observer Pattern

- The participants of the Observer pattern:
- **Subject**– interface or abstract class defining the operations for attaching and de-attaching observers to the client. Also known as, **Observable**.
- **ConcreteSubject** – concrete Observable class. It maintain the state of the observed object and when a change in its state occurs it notifies the attached **Observers**.
- **Observer** – interface or abstract class defining the operations to be used to notify the Observer object.
- **ConcreteObserverA, ConcreteObserverB** – concrete **Observer** implementations





Observer Example

```
class EventSource {
    public interface Observer {
        void update(String event);
    }

    private final List<Observer> observers = new ArrayList<>();

    private void notifyObservers(String event) {
        observers.forEach(observer -> observer.update(event));
    }

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void scanSystemIn() {
        Scanner scanner = new Scanner(System.in);
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            notifyObservers(line);
        }
    }
}
```

Observer Example

```
public class ObserverDemo {
    public static void main(String[] args) {
        System.out.println("Enter Text: ");
        EventSource eventSource = new EventSource();

        eventSource.addObserver(event -> {
            System.out.println("Received response: " + event);
        });

        eventSource.scanSystemIn();
    }
}
```