Design Patterns

1. The Observer Pattern

Register and notify.

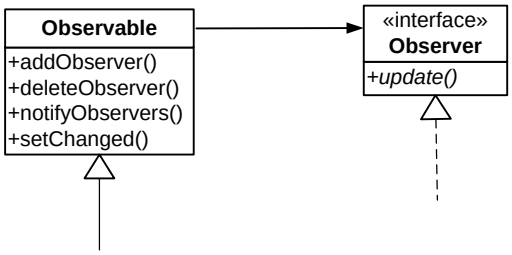
Publishers + Subscribers = Observer Pattern

As a good example, a newspaper subscription service with its publisher and subscribers is a good way to visualize the pattern.

Definition: the Observer Pattern defines a one-to-many dependency between objects so that one object changes state, all of its dependents are notified and updated automatically.

Using Java's build-in Observer Pattern:

Java has build-in support in several of its APIs. The most general is the Observer interface and the Observable class in the java.util package. You can implement either a push or pull style of update to your observers.



public void **notifyObservers**()
public void **notifyObservers**(Object arg)

`void` **`update`**`(Observable o,` Object arg)

Questions:

- The notifyObservers() method is a public method, if every observer calls this method, then what will happen?
- How to use pull?
  A: Use the sub class's public get methods.
-

The dark side of java.util.Observable:

The Observable is a class, not an interface, and worse, it doesn't even implement an interface. This means you have to subclass it, and that limits the reuse potential.

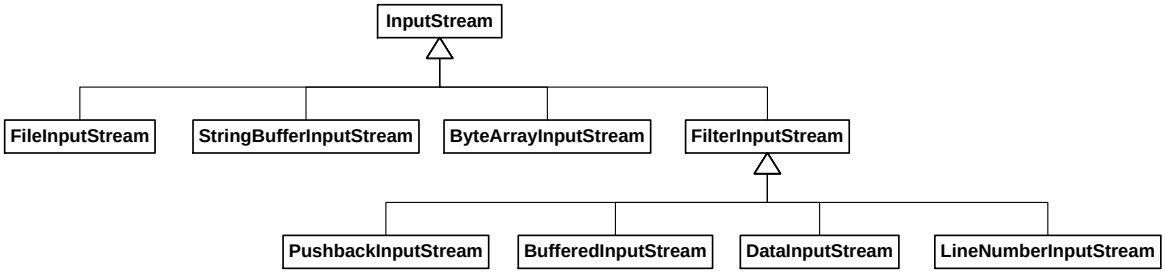
2. The Decorator Pattern

Wrappers.

The decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Each component can be used on its own, or wrapped by a decorator.
Each decorator HAS-A (wraps) component, which means the decorator has an instance variable that holds a reference to a component.
Decorators implement the same interface or abstract class as the component they are going to decorate.

An Java I/O example:



InputStream is the abstract component.
FilterInputStream is an abstract decorator.
The left three are concrete components that will be wrapped with decorators.
The bottoms four are the concrete decorators.

```
InputSteam in = new BufferedInputStream(new FileInputStream("test.txt"));
```

Q: Why do we need an abstract decorator class in the middle? May be it is used to group the decorators into a sub inheritance tree, based on their behaviors, to make the class diagram clear?

3. The factory pattern

4. The singleton pattern

The **Singleton Pattern** ensures a class has only one instance, and provides a global point of access to it.

```
public class Singleton {

	private static Singleton uniqueInstance;

	private Singleton() {};

	public static synchronized Singleton getInstance() {

		if (uniqueInstance == null ){
			uniqueInstance = new Singleton();
		}
		return uniqueInstance;
	}
}
```

The only time that synchronization is relevant is the first time through this method. After the first time though, synchronization is totally unneeded overhead.

```
public class Singleton {

	private static Singleton uniqueInstance = new Singleton();

	private Singleton() {};

	public static Singleton getInstance() {

		return uniqueInstance;
	}
}
```

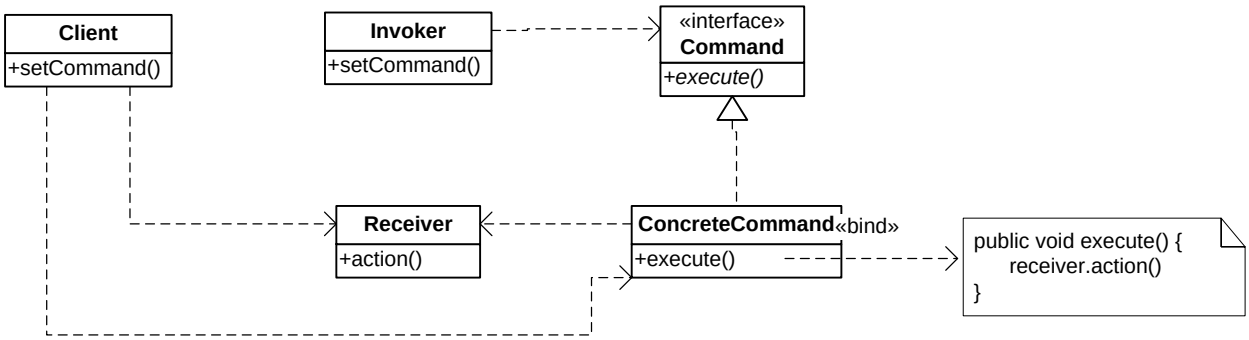
The JVM guarantees that the instance will be created before any thread accesses the static uniqueInstance variable.

```
public class Singleton {
	private volatile static Singleton uniqueInstance;
	private Singleton() {};

	public static Singleton getInstance() {
		if (uniqueInstance == null ){
			synchronized (Singleton.class){
				if (uniqueInstance == null ){
					uniqueInstance = new Singleton();
				}
			}
		}
		return uniqueInstance;
	}
}
```

The first check could be a thread turn over; meanwhile the volatile keeps the variable to be unique.

5. The Command Pattern

The command pattern encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.



- Command declares an interface for all commands.
- The ConcreteCommand defines a binding between an action and a receiver.
- The Invoker holds a command and at some point asks the command to carry out a request by calling its execute method.
- The client is responsible for creating a ConcreteCommand and setting its receiver.

Note:

- Using state to implement Undo. Maintain a set of states in the ConcreteCommand.
- Using macro command. Keep a set of commands inside a concrete command.

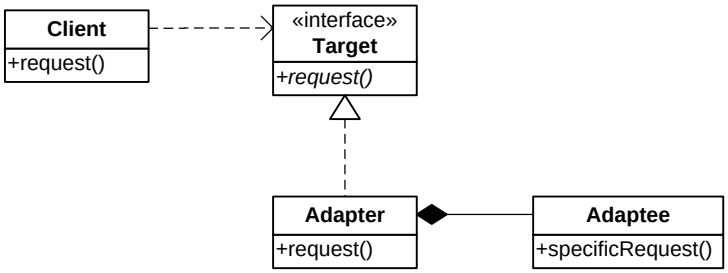
Used in applications:

1. Queuing requests
   Thread remove a command from the queue, calls its execute() method.

2. Logging requests.
   By using logging, we can save all the operations since the last check point.

6. The Adapter and Façade patterns

The **Adapter Pattern** converts the interface of a class into another interface the clients expect.



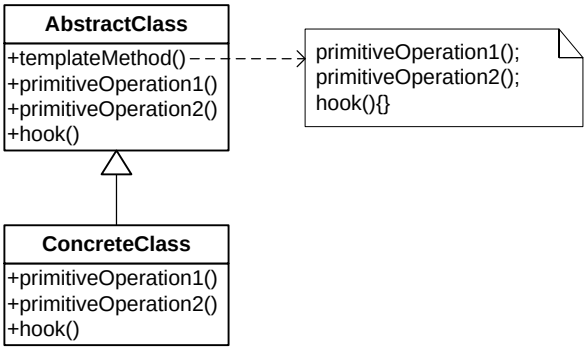
- Client sees only the target interface.
- The Adapter implements the target interface, and is composed with an Adaptee.
- All requests get delegated to the Adaptee.

Note: This pattern is similar to the decorator pattern. The difference is that the Adapter delegates the request to the Adaptee, while the decorator adds more behaviors to the class it decorates.

The **Façade Pattern** provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

7. The Template Method Pattern

The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



The abstract class contains the template method. The template method makes use of the primitive operations to implement the algorithm.

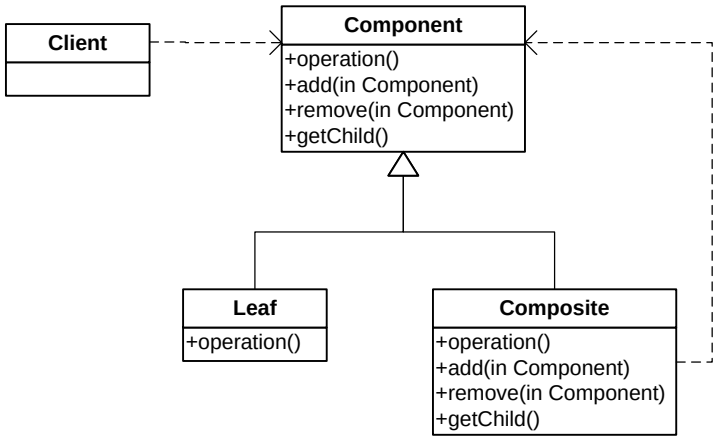
There may be many concrete classes, each implementing the full set of operations required by the template method.

A hook is a method declared in the abstract method, given an empty or default implementation. This gives subclasses the ability to "hook into" the algorithm at various points. The subclass is also free to ignore the hook. An example of using hook is AOP, wrap an optional begin and end operations around the actually operation.

```
abstract class AbstractClass {

	final void templateMethod(){
		primitiveOperation1();
		primitiveOperation2();
		concreteOperation();
		hook();
	}

	abstract void primitiveOperation1();
	abstract void primitiveOperation2();
	final void concreteOperation(){//implementation here};
	void hook(){}

}
```

8. The Iterator and Composite Patterns

The **Composite Pattern** allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of object uniformly.



- The client uses the Component interface to manipulate the objects in the composition.
- The Component defines an interface for all objects in the composition.
- A leaf defined the behavior for the elements in the composition, by implementing the operations.
- The composite defines the behavior of components having children, also implement the leaf operations.

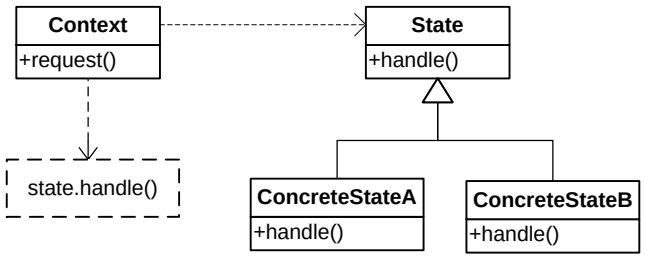
Examples: Swing Manu and ManuItem

To iterate through a composite component, we need a composite Iterator. The CompositeIterator implement the Iterator interface, and overwrite the methods. The constructor takes an Iterator and pushes it into a stack. In the methods (next(), hasNext()), it pops up an Iterator from the stack and calling each Iterator's methods.

9. The State Pattern

A state machine.

The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.



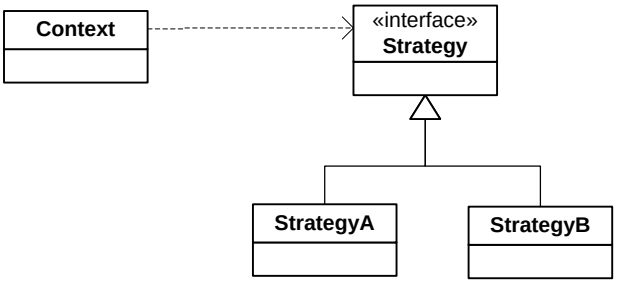
- The Context (the machine) is the class that can have a number of internal states and maintains a current state. Whenever the request (actions) is made of the Context it is delegated to the state to handle, and the current state may get changed as well.
- The State interface defines a common interface for all concrete states.
- Concrete states handle request from the Context. Each concrete state provides its own implementation for a request. Each concrete state holds a reference of the Context, and calling setState method for each action.
- When the Context changes its states, the individual concrete state's behavior will change as well.

Note: The disadvantage is that we create dependencies between states.

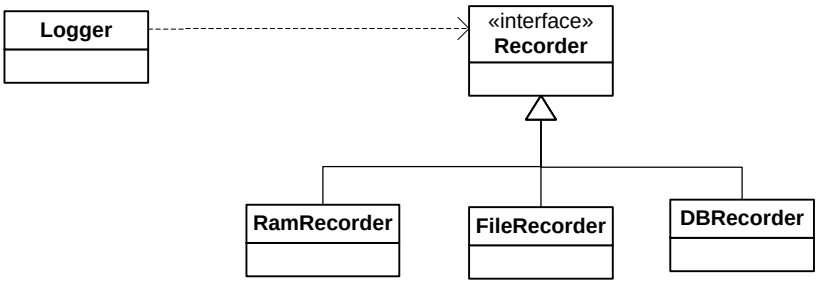
10. The Strategy Pattern

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

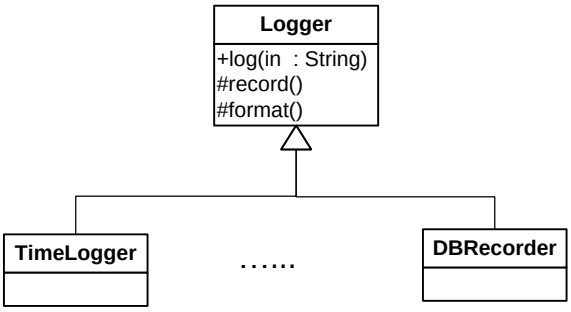
The canonical form of the Strategy pattern is shown below. One algorithm (the *context*) is shielded from the other (the *strategy*) by an interface. The context is unaware of how the strategy is implemented, or of how many different implementations there are. The context typically holds a pointer or reference to the strategy object with which it was constructed.



An logger example:



Strategy + Template Method => Bridge Pattern



Decorator pattern can also apply to the above diagram.

11. The Proxy Pattern

12. The Delegate Pattern

The delegation pattern is a design pattern where an object, instead of performing one of its stated tasks, delegates that task to an associated helper object. It passes the buck, so to speak (technically, an Inversion of Responsibility). The helper object is called the delegate.

```
class RealPrinter { // the "delegate"
     void print() {
       System.out.print("something");
     }
 }

 class Printer { // the "delegator"
     RealPrinter p = new RealPrinter(); // create the delegate
     void print() {
       p.print(); // delegation
     }
 }

 public class Main {
     // to the outside world it looks like Printer actually prints.
     public static void main(String[] args) {
         Printer printer = new Printer();
         printer.print();
     }
 }
```

13. The Visitor Pattern

Patterns have been Used

1. Template method pattern

   Dao, mapResult method in the BaseDao class.
   Adapters for the Swing listener classes. Applet's init, start, stop, destroy methods.

2. Command pattern

   JUnit, Log4j

3.