DESIGN PATTERNS

A Design Pattern is a pattern that captures a solution to a recurring design problem, it is neither a pattern for implementation problems nor a ready-made solution that has to be applied. It is a description of communicating objects and classes that are customized to solve a general design problem in a particular context. It is a kind of blueprint which consists of different parts, and all these parts together make the pattern. When we talk about the pattern, we therefore mean all of these parts together (not only the class diagram, etc.).

Each design pattern describes a problem which occurs over and over again 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 doing it the same twice.

The description of a design pattern includes a name, a description of the design problem or context it addresses, a solution template and a discussion of the consequences of applying the pattern:

1) Pattern name: increase of design vocabulary.
2) Problem description: when to apply it, in what context to use it.
3) Solution description: the elements that make up the design, their relationships, responsibilities and collaborations -> generic!
4) Consequences: results and trade-offs of applying the pattern.

Do not apply too many patterns, look at trade-offs, most patterns make systems more complex (but address a certain need). Do good modeling and see whether patterns can help, and where.

Design patterns are:

a) Smart: they are elegant solutions that a novice would not think of
b) Generic: they are independent on specific system type, language (although slightly biased towards C++)
c) Well-proven: design patterns have been successfully tested in several systems
d) Simple: can be combined for more complex solutions
e) Well-known: they can be used to communicate complex ideas more easily

Design patterns do not describe recognizable flaws in software design, they address them. Design patterns convey know-how and experience, using them may contribute to making a software design more flexible. Sometimes using design patterns may lead to unnecessary structures and clutter in the code.

A design antipattern is an abstract description of a common design flaw (difetto di design).

Singleton

Definition: the Singleton Pattern ensures that only one object is instantiated for a class, and provides a global point of access to it.

Use cases: When you need to ensure that only one instance of a class is created, and a global point of access to it is provided. The global point of access to the instance can be useful when you want to ensure that other parts of the system can access the shared resource without having to pass the instance around as a parameter.

Disadvantages: a) it can introduce tight coupling between the code that uses the singleton and the singleton itself, which can make it difficult to change the implementation of the shared resource, and if you make a change to the Singleton you will likely have to make a change to every object connected to it.

b) It might break the Single Responsibility Principle (SRP): The singleton is not only responsible for managing its one instance and providing global access but also for whatever is the role of the Singleton class in the complete program; so it could be argued that it is taking on two responsibilities.

c) A class with a singleton cannot be subclassed since the constructor of the Singleton is private and a class cannot extend a class that has a private constructor. To be able to extend it the constructor would need to be changed to protected or private but then it's not really a Singleton anymore because other classes can also instantiate it. If we change the constructor then all the subclasses will share the same instance variable since the implementation is based on a static variable.

Code structure:

```
public class Singleton {
   private static Singleton uniqueInstance;

   private Singleton() {}

   public static Singleton getInstance() {
      if (uniqueInstance == null) {
         uniqueInstance = new Singleton(); //instantiation through private constructor
      }
      return uniqueInstance;
   }
}
```

Technical aspects: to implement the singleton pattern, you typically define a class with a private constructor, which prevents other classes (or clients) from creating multiple objects. You then define a static accessor method (often called getInstance) that returns the private single instance of the class. This method includes a check to see if the instance has already been created and creates the instance if it does not yet exist. The accessor method is optional, because it is also possible to implement the pattern by declaring the global instance to be a public constant. All the fields inside the Singleton class (apart must be static (so we don't need to instantiate a new object).

Why is getInstance a static method? because it allows you to access the single instance of the class without having to create a new instance, if it wasn't static then we would need to instantiate an object of the Singleton class to access the getInstance method and we would then have two instances of the singleton class which is clearly not what we want.

Risks: when we have multiple threads ("workers") at the same time, it cannot be ensured that only one instance of the Singleton class is created.

If multithreading is prevented, at most one instance of the Singleton class can be instantiated by other classes than Singleton itself by using the statement new Singleton().

Solutions:

1) Add synchronized keyword to getInstance() so that we force each thread to wait its turn before it can enter the method. This ensures that two thready never enter the method at the same time. Synchronizing a method can decrease performance by a factor of 100.

```
public static synchronized Singleton getInstance()
```

2) create the singleton instance in the static initializer.

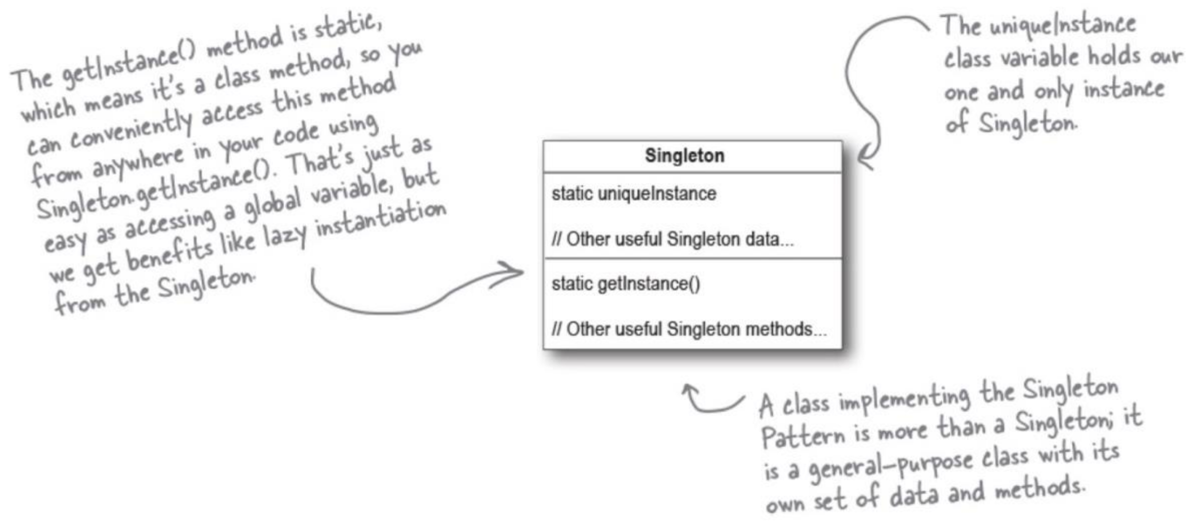
```
public class Singleton {
   private static Singleton uniqueInstance = new Singleton();

   private Singleton() {}

   public static Singleton getInstance() {
      return uniqueInstance;
   }
}
```

3) double-checked locking: we first check to see if an instance is created, and if not then we synchronize. This is not the most recommended approach, prefer either solution 1 or 2.

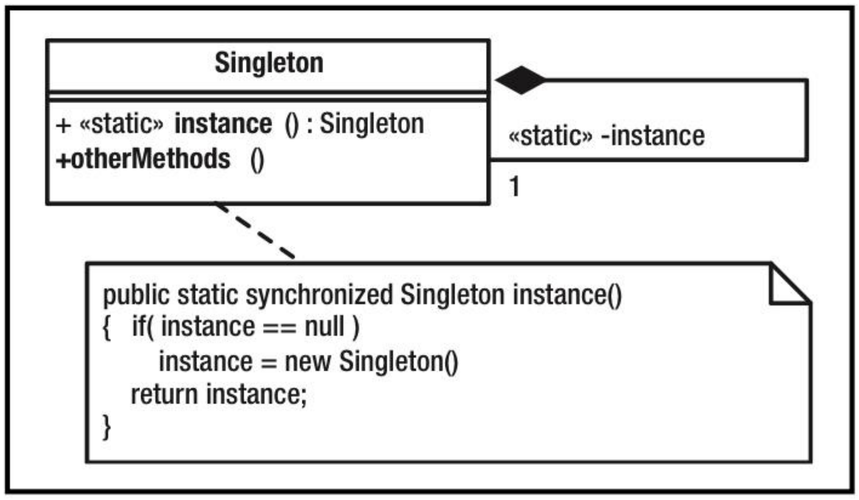
With the volatile keyword we can ensure that multiple threads handle the uniqueInstance variable correctly when it is being initialized to the Singleton instance.

```
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;
   }
}
```

Class Diagram:


The getInstance() method is static, which means it's a class method, so you can conveniently access this method from anywhere in your code using Singleton.getInstance(). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.
The uniqueInstance class variable holds our one and only instance of Singleton.
Singleton
static uniqueInstance
// Other useful Singleton data...
static getInstance()
// Other useful Singleton methods...
A class implementing the Singleton Pattern is more than a Singleton; it is a general-purpose class with its own set of data and methods.

UML diagram:



```
public static synchronized Singleton instance()
{   if( instance == null )
        instance = new Singleton()
    return instance;
}
```

Theoretically, you could create a class in which all variables and methods are static to simulate the Singleton pattern, but this can get very messy and it can result in hard-to-find bugs which involve the order of initialization. It also moves away from the object oriented point of view, and a class with all static methods and variables would break these design principles:

- Single Responsibility Principle (SRP): a class with all static methods and variables may have multiple responsibilities, which can make it more difficult to understand and maintain.

- Open-Closed Principle (OCP): This principle states that a class should be open for extension but closed for modification. A class with all static methods and variables cannot be extended or modified, since it does not have any instance variables or methods.

- Liskov Substitution Principle (LSP): a class with all static methods and variables cannot be subclassed, so it violates the LSP.

Global variables vs Singleton: in Java, a global variable is a static reference to an object. A global variable can provide global access to it but not that only one instance of a class exists.

Singleton with enum:

```
enum SingletonEnum {
   uniqueInstance;

}
class Main {

   public static void main(String[] args) {
      SingletonEnum singletonInstance = SingletonEnum.uniqueInstance;

   }
}
```

Enums might solve problems regarding serialization and reflection, but cannot extend other classes (it already extends java.lang.enum). Also, an enum cannot be abstract, whereas in classes you could have an abstract superclass that stores an instance of a subclass.

(Reflection is a feature of many programming languages that allows a program to inspect and modify itself at runtime. It allows a program to introspect on itself, to discover the structure of its own code, and to modify that code on the fly.

<u>**Iterator**</u>

Definition: the Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. It takes the job of iterating over an aggregate and encapsulates it in another object.

Use cases: 1) often used to provide a standard interface for traversing a collection of objects, such as a list or an array. This allows the client to access the elements of the collection without needing to know the underlying representation of the collection.

2) The Iterator can be used when you want to perform the same operations (hasNext, next, remove) on a large amount of aggregates without needing to have different code for each aggregate.

2) The iterator design pattern can be used to provide a uniform interface for traversing different types of collections, even if they have different underlying representations.

3) The iterator design pattern can be used to hide the implementation details of a collection from the client. This can make it easier to change the implementation of the collection without affecting the client code.

4) The iterator design pattern allows the client to perform multiple traversals of a collection, either in the same order or in a different order, without needing to know the underlying representation of the collection. It allows to access an aggregate's element without exposing its internal structure.

Advantages: 1) it promotes code reuse by hiding the implementation behind.

2) The aggregate is relieved from the responsibility of supporting operations for traversing its data.

Disadvantages: 1) Implementing the iterator design pattern might make the code too complex, as it requires creating separate iterator and aggregate classes and defining their interfaces.

2) A client may modify the elements of the aggregation, damaging the aggregate (for example by changing the key in the sorted aggregate).

3) The aggregate may store reference to its iterators; memory leaks are possible if an iterator is discarded without notifying the aggregate.

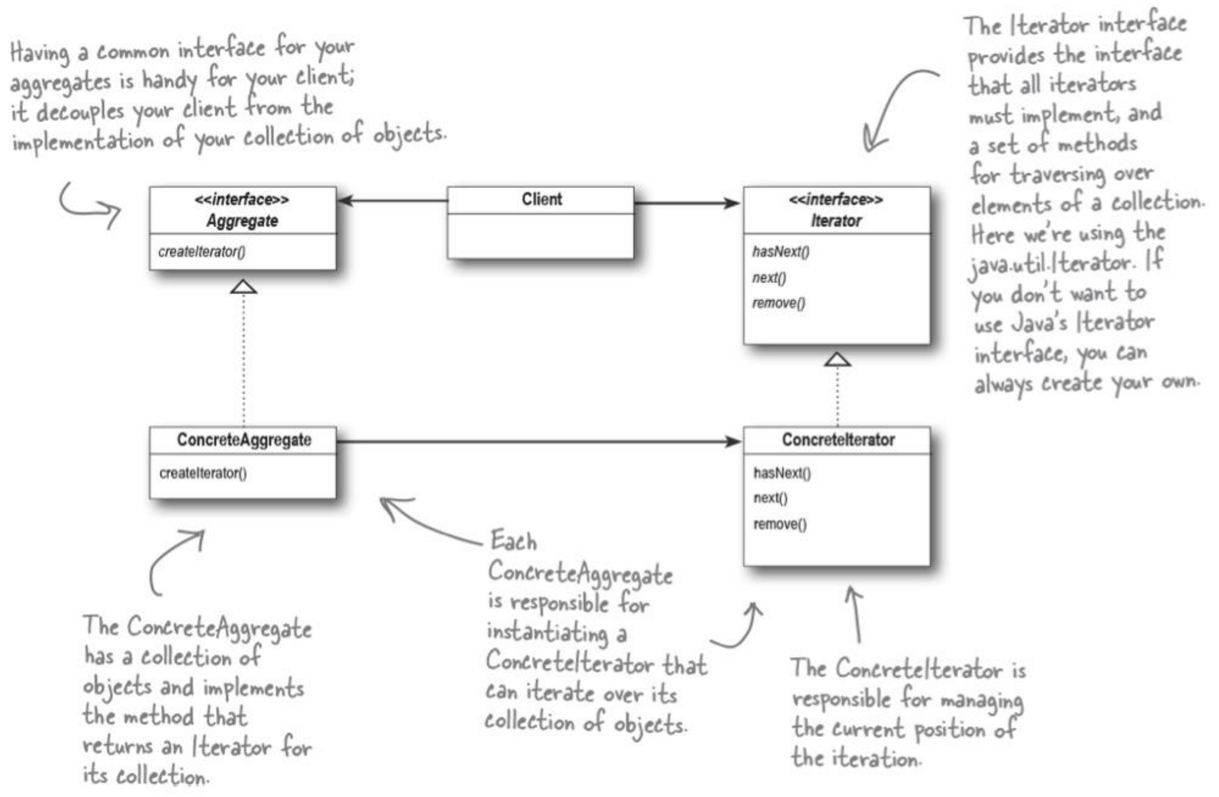
4) Difficult to implement in an environment that supports simultaneous iteration and modification: if a new item is added while iterations are in progress, should the iterator visit the newly added item? What if the aggregate (e.g. list) is ordered and you've already passed the place where the new item has to be inserted? There are no correct answers to these type of questions.

Code structure:

```
public interface Iterator {
    public boolean hasNext();
    public Object next();
    public void remove();
}



public class ConcreteIterator implements Iterator {
    private ConcreteAggregate aggregate;
    private int position = 0;

    public ConcreteIterator(ConcreteAggregate aggregate) {
        this.aggregate = aggregate;
    }

    public boolean hasNext() {
        if (position >= aggregate.size()aggregate.get(position) == null) {
            return false;
        }

       else {
        return true;

       }
    }

    public Object next() {

       ConcreteAggregate object = aggregate.get(position);
        position = position + 1;
       return object;



   }

    public void remove() {
        aggregate.remove(--current);
    }
}

public interface Aggregate {
    public Iterator createIterator();
    public void add(Object item);
}

public class ConcreteAggregate implements Aggregate {
    private ArrayList items = new ArrayList();
    public Iterator createIterator() {
        return new ConcreteIterator(this);
    }

    public int size() {
        return items.size();
    }

    public Object get(int index) {
        return items.get(index);
    }

    public void set(int index, Object value) {
        items.set(index, value);
    }

    public void add(Object item) {
        items.add(item);
    }
}
```

Technical aspects: The Iterator Pattern relies on an interface called Iterator, which contains the next() method, used to return the next object in the aggregate, and the hasNext() method, used to check if there are more elements in the aggregate to iterate through. Once we have this interface we can implement different iterators for any kind of objects (arrays, lists, hash maps, etc.).

Class Diagram:


Having a common interface for your aggregates is handy for your client; it decouples your client from the implementation of your collection of objects.
<<interface>> Aggregate
createIterator()
Client
<<interface>> Iterator
hasNext()
next()
remove()
The Iterator interface provides the interface that all iterators must implement, and a set of methods for traversing over elements of a collection. Here we're using the java.util.Iterator. If you don't want to use Java's Iterator interface, you can always create your own.
ConcreteAggregate
createIterator()
ConcreteIterator
hasNext()
next()
remove()
The ConcreteAggregate has a collection of objects and implements the method that returns an Iterator for its collection.
Each ConcreteAggregate is responsible for instantiating a ConcreteIterator that can iterate over its collection of objects.
The ConcreteIterator is responsible for managing the current position of the iteration.

The remove method allows to remove the last item returned by the next() method from the aggregate (it is considered optional, but it should be provided because it is part of the interface). If the remove method is removed from the interface, then an UnsupportedOperationException which tells that it’s not possible to remove an element from the aggregate should be thrown.

The concreteIterator implements the Iterator interface and keeps track of the current position. The aggregate defines n interface for creating an iterator and the concreteAggregate holds the data and implements the aggregate interface to manufacture an iterator.

The iterator pattern takes the responsibility of traversing elements and gives that responsibility to the iterator object, not the aggregated object.

Strategy

Definition: 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 Strategy Pattern encapsulates interchangeable behaviors an uses delegation to decide which behavior to use.

Use cases: 1) When the code has multiple algorithms that are similar but differ in their implementation, the Strategy pattern allows you to select the appropriate algorithm at runtime based on the specific needs of your application. When we would like to change the behavior of an object at runtime without modifying the object itself.

2) When we need different implementations of an algorithm for different types of objects without creating a separate subclass for each implementation.

Advantages:

1) The Strategy pattern allows you to encapsulate each algorithm or behavior in a separate class, which makes it easy to add new algorithms or behaviors without changing the existing code. The strategy pattern is flexible because it makes use of object composition, and clients can change their algorithms at runtime by (simply) using a different strategy object.

2) It is a good alternative to subclassing. Rather than overriding methods from the superclass, an interface is created.

3) An object can take on multiple forms -> Polymorphism.

Disadvantages: 1) The application must be aware of all the strategies to select the right one for the right situation.

2) Context and the Strategy classes normally communicate through the interface specified by the abstract Strategy base class. Strategy base class must expose interface for all the required behaviours, which some concrete Strategy classes might not implement.

3) communication overhead is small. Some of the arguments passed to the Strategy objects may not be used.

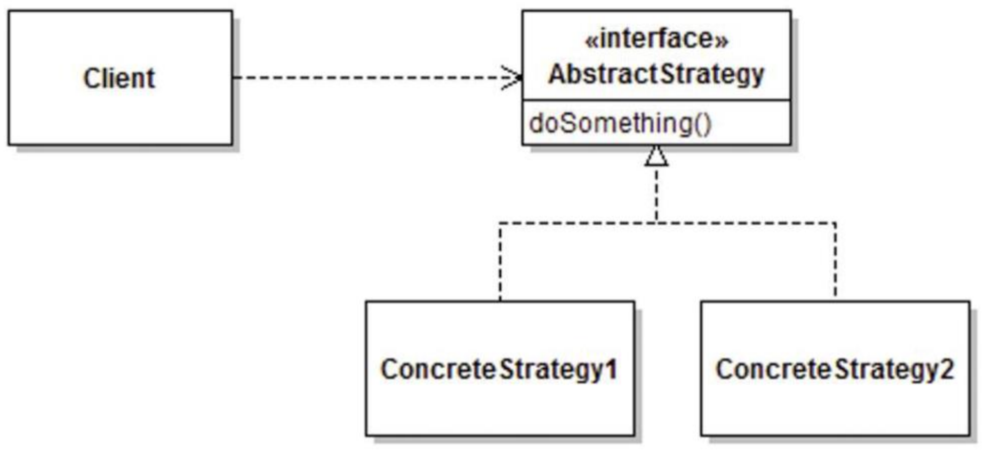
Code structure:

```
public interface Strategy {
    void execute();
}

class Context {
    private Strategy strategy;

    public Context(Strategy strategy) {
        this.strategy = strategy;
    }
```

```java
    public void setStrategy(Strategy strategy) {
        this.strategy = strategy;
    }

    public void executeStrategy() {
        strategy.execute();
    }
}

class ConcreteStrategyA implements Strategy {
    @Override
    public void execute() {
        // Implementation of algorithm A
    }
}

class ConcreteStrategyB implements Strategy {
    @Override
    public void execute() {
        // Implementation of algorithm B
    }
}
```

Technical aspects: the context object is composed of a strategy object, and it delegates the execution of the algorithm to the strategy. This allows to change the behavior of the context object at runtime by replacing the strategy object with a different one. Strategy is an interface that allows access to an algorithm, concrete strategy implements a particular algorithm to conform the Strategy interface and Context uses the algorithm through the Strategy interface.

Class Diagram:



Flyweight

Definition: the Flyweight pattern minimizes memory usage by sharing some of the initial objects data with other similar objects.

Use cases: in situations where instances of a class are heavily shared among objects and we want to control them in the same way without using too much memory.

Advantages:

1) It reduces the number of object instances at runtime, saving memory.

2) When using Flyweight, the equality of two objects can be determined with the == operator.

Disadvantages: 1) once implemented, the instances of the class will not be able to behave independently from other instances.

2) Flyweights add complexity to the code, impacting maintenance and increasing code size.

Code structure:

```java
public interface Flyweight {
    void operation(int extrinsicState);
}

public class ConcreteFlyweight implements Flyweight {
    private int intrinsicState;

    public ConcreteFlyweight(int intrinsicState) {
        this.intrinsicState = intrinsicState;
    }

    public void operation(int extrinsicState) {
        // Use both intrinsic and extrinsic states to perform the operation
    }
}

public class FlyweightFactory {
    private Map<Integer, Flyweight> flyweights = new HashMap<>();

    public Flyweight getFlyweight(int key) {
        if (!flyweights.containsKey(key)) {
            flyweights.put(key, new ConcreteFlyweight(key));
        }
        return flyweights.get(key);
    }
}
```

Technical aspects: The flyweight design pattern works by separating the intrinsic properties (those that are the same for objects of the class) from the extrinsic properties (those that are different for objects of the class). The intrinsic data is stored in the flyweight object itself, while the extrinsic data is passed to the flyweight object as a parameter when it is used. The Flyweight interface defines the interface for the flyweight objects, and the ConcreteFlyweight class implements the Flyweight interface. The FlyweightFactory class is responsible for creating and managing the flyweight objects. When a client requests a flyweight, the factory checks to see if it has already created an instance of that flyweight. If it has, it returns the existing instance; if not, it creates a new instance and adds it to its internal map. This allows multiple clients to share the same flyweight instance, reducing the overall memory usage. Flyweights are immutable. Instances of a flyweight class are called flyweight objects.

The flyweight design pattern consists of three main components:

1) The flyweight interface: This defines the interface for the flyweight objects. It may include methods for performing operations on the flyweight object, as well as methods for accessing and modifying the intrinsic state of the object.
2) The concrete flyweight class: This implements the flyweight interface and defines the intrinsic state of the flyweight object. It may also include methods for accessing and

modifying the intrinsic state of the object. The client class may pass extrinsic data to the flyweight object when it is used, which allows the flyweight object to perform operations using both the intrinsic and extrinsic data.

3) The flyweight factory: This is responsible for creating and managing the flyweight objects. It maintains a pool of flyweight objects and returns existing flyweight objects when they are requested. If a flyweight object does not exist, the factory creates a new one and adds it to the pool.

Instances of the flyweight class are called flyweight objects. The crucial aspect of flyweight is to control the creation of flyweight objects to an access method that ensures that no duplicate objects ever exist. A flyweight refers to an object that minimizes memory usage by sharing some of the initial object's data with other similar objects.

The Flyweight has a private constructor for the flyweight class, so clients cannot control the creation of objects of the class.

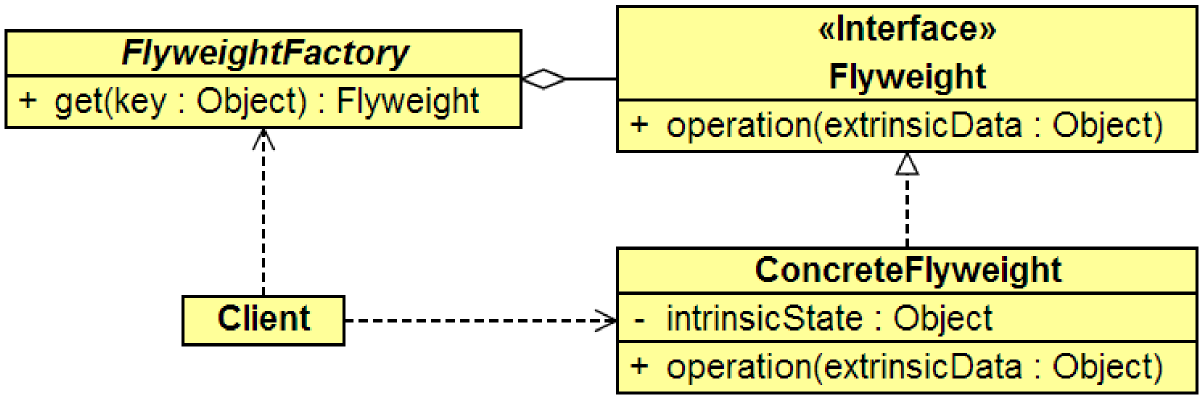
There is also a static flyweight score that keeps a collection of flyweight objects, and a static access method that returns the unique flyweight object that corresponds to some identification key. The access method typically checks whether the requested flyweight object already exists in the store, creates it if it does not already exist, and returns the unique object.

This pattern splits the state of the initial object into an intrinsic-immutable state and an extrinsic-mutable one.

The combination of the flyweight store and corresponding access method is sometimes referred to as the flyweight factory. The flyweight's factory returns the flyweight possibilities that we have.

An important concern when implementing the flyweight pattern is whether to pre-initialize the flyweight store, or whether to do this lazily, by creating objects as they are requested through the access method -> the answer is context-dependent. In general, in cases where there exists a small and finite set of flyweights, it may make sense to pre-initialize them. Flyweights are immutable.

Class Diagram:


FlyweightFactory
+ get(key : Object) : Flyweight
«Interface»
Flyweight
+ operation(extrinsicData : Object)
ConcreteFlyweight
- intrinsicState : Object
+ operation(extrinsicData : Object)
Client

Singleton vs Flyweight: the singleton pattern differs from Flyweight in that it attempts to guarantee that there is a single instance of a class, as opposed to unique instance of a class. Singleton objects are usually stateful and mutable, whereas flyweight objects are preferably immutable. Mistake to avoid: do not store a reference to an instance of the class in a static field called instance of something similar, without taking proper care to prevent client code from independently creating new objects (the class does not yield a single instance in this way). A singleton is essentially a global instance, accessible from anywhere in the code. Singleton makes a stateful object difficult to replace when necessary. The objects in a flyweight can be created on-demand. The accessor methods for the Flyweight must be static (in the FlyweightFactory).

Composite

Definition: 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 objects uniformly (with common methods to be called on composites and leaves). The fundamental idea is to define a class that represents multiple interfaces while still behaving like a single one. A composites contains components, components are either composites or leaf elements.

Use cases: used where we need to treat a group of objects with similar properties as a single object, for example when representing directories, which are files that contain other file (file system).

Advantages: 1) the Composite Pattern allows to build structures of objects in the form of trees that contain both compositions of objects and individual objects as notes. Using a composite structure, we can apply the same operations over both composites and individual objects. In most cases we can ignore the differences between compositions of objects and individual objects. The children of a composite can either be leaves or other composites theirselves.

2) The container is simple to implement because it treats all contents uniformly.

3) It is simple to add or remove objects from the tree structure and allows to work with objects in the tree as a single unit.

4) Individual objects in the tree structure do not need to be aware of their position in the tree.

Disadvantages: It’s not always meaningful or appropriate for every Composite or Leaf to implement every method of the Component. It’s an awkward runtime error if an unimplementable method throws an exception.

Code structure:

```java
// Component interface
public interface Component {
    void operation();
}

// Leaf class
public class Leaf implements Component {
    @Override
    public void operation() {
        // Implementation for leaf node
    }
}

// Composite class
public class Composite implements Component {
    private List<Component> children = new ArrayList<>();

    @Override
    public void operation() {
        // Implementation for composite node
        for (Component child : children) {
            child.operation();
        }
    }

    public void add(Component component) {
        children.add(component);
    }

    public void remove(Component component) {
        children.remove(component);
    }
}
```

Technical aspects: the composite treats all content uniformly. A composite doesn’t know whether its subcomponents are leaves or other composites. Leaves cannot become composites by adding other eaves to them.

The composite design pattern has three main components:

1) Component: This is the base interface for objects in the composition, and it defines the common behavior that all objects in the composition must have.
2) Leaf: This is a simple object that represents a leaf node in the tree structure. It implements the behavior defined in the component interface.
3) Composite: This is a composite object that represents a composite node in the tree structure. Two important features:
   a) It aggregates a number of different objects of the component type. Using the component interface type is important, as it allows the composite to compose any other kind of elements, including other composites.
   b) It implements the behavior defined in the component interface; this is what allows composite objects to be treated by the rest of the code in exactly the same way as leaf elements. The composite object also has methods for adding and removing child objects, as well as for accessing and manipulating them.

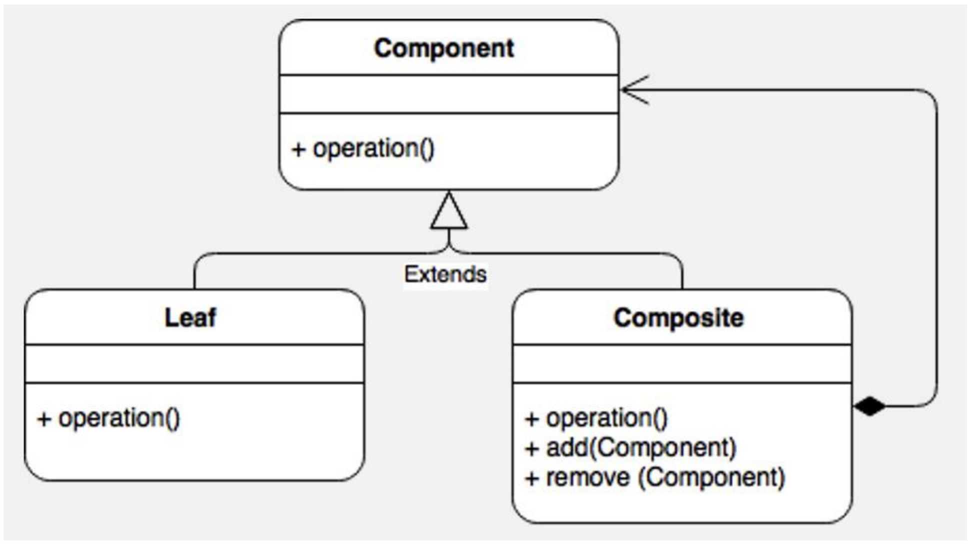
4) Client

The Component interface (or abstract class) defines the common behavior that all objects in the composition must have, and the Leaf and Composite classes implement this interface. The Composite class maintains a collection of child objects and has methods for adding and removing child objects, as well as for accessing and manipulating them. To use the composite design pattern, you would create instances of the Leaf and Composite classes and add them to the tree structure as needed. You can then call the operation() method on any object in the tree to perform the operation defined in the Component interface. If the object is a leaf node, then the request is handled directly, and if it is a composite the request its forwarded to its children (some operation + combined operation).

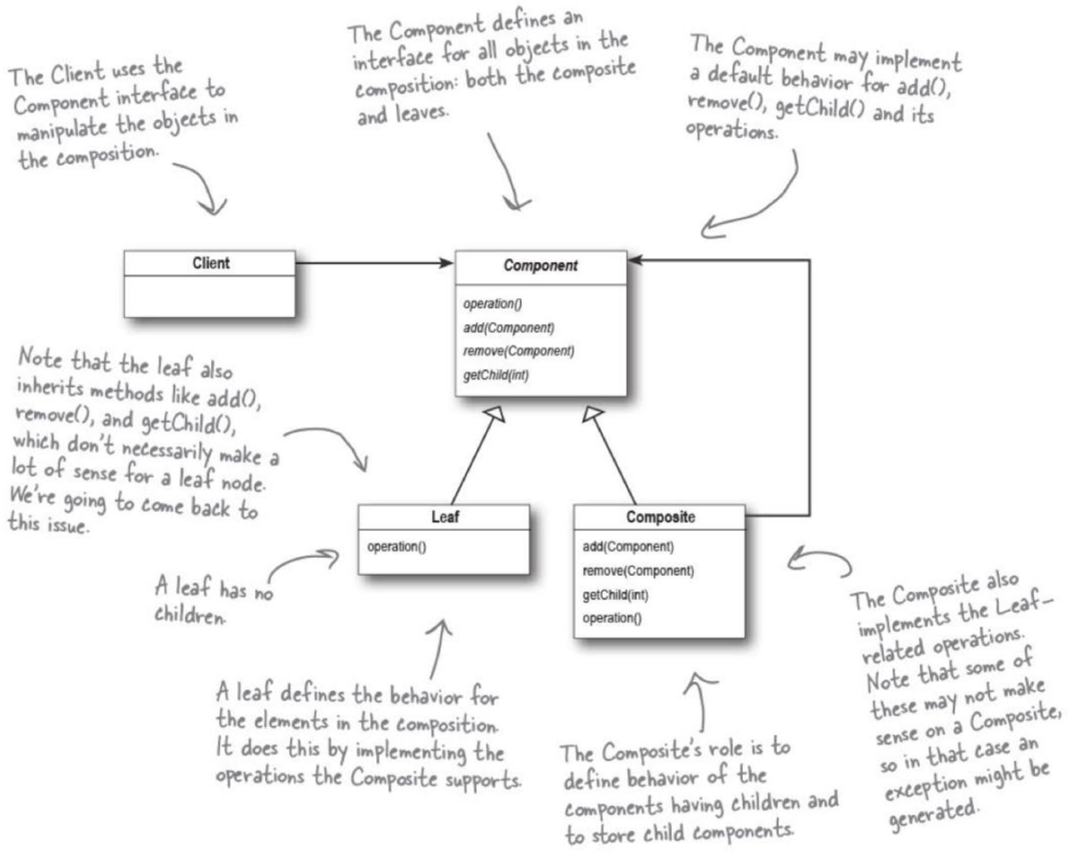
For the Composite to be effective, client code should depend primarily on the component type, and not manipulate concrete types directly. Having an interface in the Composite patterns allows to write loosely coupled code.

Class Diagram:

1)



2)


The Client uses the Component interface to manipulate the objects in the composition.
The Component defines an interface for all objects in the composition: both the composite and leaves.
The Component may implement a default behavior for add(), remove(), getChild() and its operations.
Client
Component
operation()
add(Component)
remove(Component)
getChild(int)
Note that the leaf also inherits methods like add(), remove(), and getChild(), which don't necessarily make a lot of sense for a leaf node. We're going to come back to this issue.
Leaf
operation()
Composite
add(Component)
remove(Component)
getChild(int)
operation()
A leaf has no children.
A leaf defines the behavior for the elements in the composition. It does this by implementing the operations the Composite supports.
The Composite's role is to define behavior of the components having children and to store child components.
The Composite also implements the Leaf-related operations. Note that some of these may not make sense on a Composite, so in that case an exception might be generated.

Composite and Flyweight: flyweights are often combined with Composite. Both Leaf and Component nodes can export extrinsic state to their containers.

<u>Decorator</u>

Definition: the Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Use cases: when we want to decorate some object with additional features (modify their functionality) at runtime, while being able to treat the decorated objects like any other object of the undecorated type and not modify the code of the Object class.

Advantages: 1) decorators can simplify class hierarchies by replacing subclassing with containment. If we relied on inheritance, we would have to change existing code anytime we wanted a new behavior.

2) The Decorator design pattern solves the problem of runtime configuration, which consists of not knowing exactly how an object should behave until runtime.

3) Decorator allows to assemble (at runtime) a composite object that contains exactly the mix of capabilities needed without having to know which of these capabilities will be needed when the code is being written.

4) Decorator helps braking up large complex operations into small simple operations. The size and complexity of the class is considerably reduced.

5) Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like. You can wrap a component with any number of decorators.

6) Decorators can be inserted transparently, and the client never has to know it's dealing with a decorator.

7) Prevents class explosion, which occurs if inheritance is used instead of composition with the Decorator pattern.

Disadvantages: 1) A feature introduced in a Decorator is hard (or even dangerous) to access if the decorator is itself decorated.

2) If you have code that relies on the concrete component's type, decorators will break that code. As long as you only write code against the abstract component type, the use of decorators will remain transparent to your code. However, once you start writing code against concrete components, you'll want to rethink your application design and your use of decorators.

3) Decorators can result in many small objects in our design, and overuse can be complex.

Code structure:

```
// Component interface
public interface Component {
    void doSomething();
}

// Decorator class
public abstract class Decorator implements Component {
    protected Component component;

    public Decorator(Component wrappedObj) {
        this.component = wrappedObj;
    }

    @Override
    public void doSomething() {
        // Add additional behavior
        System.out.println("Doing something extra");
        component.doSomething();
    }
}

// Concrete decorator class
public class ConcreteDecorator extends Decorator {
    public ConcreteDecorator(Component wrappedObj) {
        super(wrappedObj);
    }

    @Override
    public void doSomething() {
        // Add additional behavior
        System.out.println("Doing something even extra");
        super.doSomething();
    }
}

// Concrete base class
public class ConcreteComponent implements Component {
    @Override
    public void doSomething() {
        System.out.println("Doing something");
    }
}
```

Technical aspects: The concrete decorators store a reference to the object they decorate and add functionality to this object. Traditionally the Decorator Pattern does specify an abstract component, and in Java we could use an interface for that; using the component interface type is important, as it allows the decorator to decorate any other kind of components, including other decorators (and composites). Component is an interface for objects that can have responsibilities added to them (or have behavior modified) at runtime. Concrete Component is an object to which additional responsibilities or new behavior is attached. Decorator wraps a component and defines an interface that conforms the Component's interface but behaves differently. Concrete Decorator extends the Decorator to define the additional behavior.

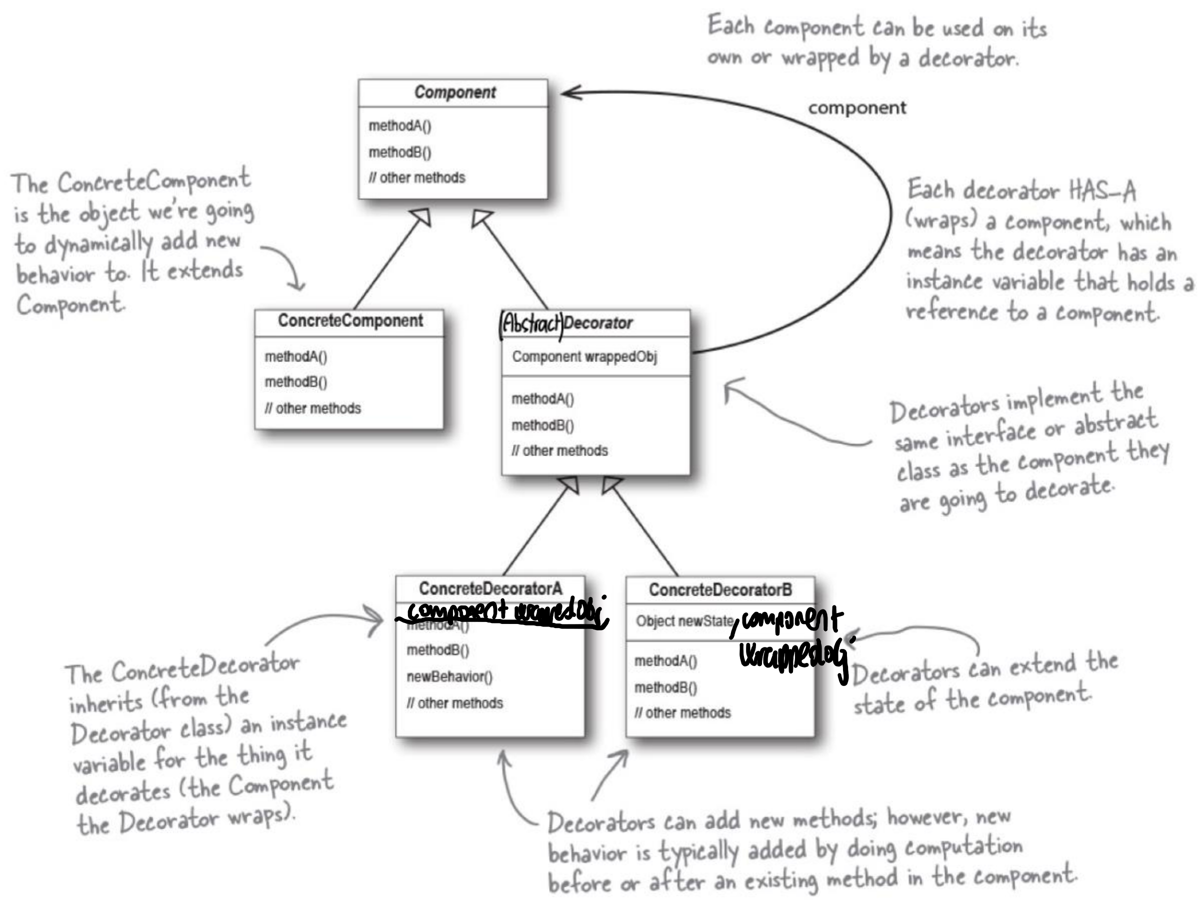
The wrapper class (abstract Decorator) is typically created using inheritance, so that it has the same interface as the object being decorated. It is called wrapper class because it wraps the behavior of an object and adds additional functionality to it.

Decorator classes mirror the type of the components they decorate. (In fact, they are the same type as the components they decorate, either through inheritance or interface implementation.

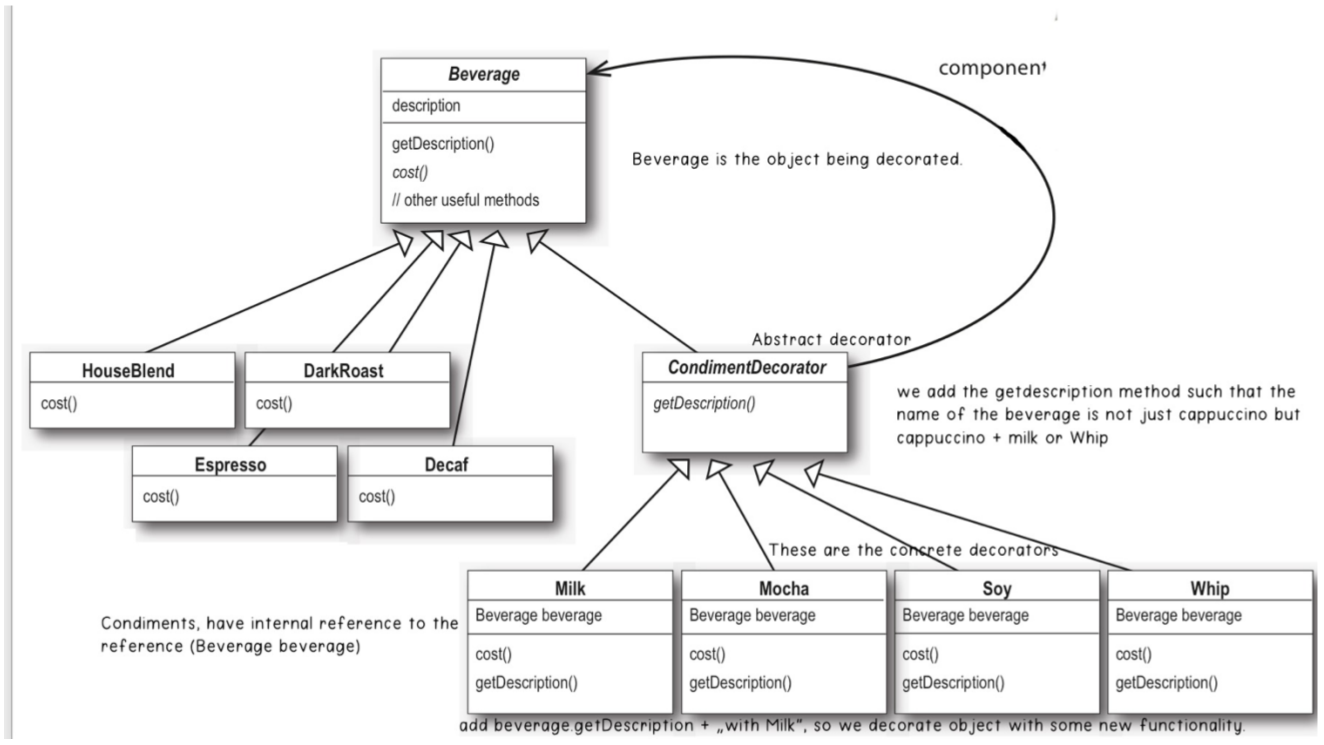
For the design to work, decorations must be independent and strictly additive. Being addictive means that the pattern should not be used to remove features from objects. When implementing the Decorator design pattern in Java, it is a good idea to specify as final the field that stores a reference to the decorated object, and to initialize it in the constructor. An important sequence of decorating objects using the Decorator design pattern is that decorator objects gain a different identity, in fact a decorated object is not the same as an undecorated object. A decorator generally decorates the same object throughout its lifetime.

Decorated elements are only accessed through the methods of the component interface. Decorators can simplify class hierarchies by replacing subclassing with containment.

Class Diagram:


Each component can be used on its own or wrapped by a decorator.
component
Component
methodA()
methodB()
// other methods
The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.
Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.
ConcreteComponent
(Abstract)Decorator
Component wrappedObj
Decorators implement the same interface or abstract class as the component they are going to decorate.
ConcreteDecoratorA
component wrappedObj
methodB()
newBehavior()
ConcreteDecoratorB
Object newState, component wrappedObj
Decorators can extend the state of the component.
The ConcreteDecorator inherits (from the Decorator class) an instance variable for the thing it decorates (the Component the Decorator wraps).
Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.

Example:


Beverage
description
getDescription()
cost()
// other useful methods
Beverage is the object being decorated.
componen
HouseBlend
DarkRoast
Espresso
Decaf
cost()
Abstract decorator
CondimentDecorator
getDescription()
we add the getdescription method such that the name of the beverage is not just cappuccino but cappuccino + milk or Whip
These are the concrete decorators
Milk
Mocha
Soy
Whip
Beverage beverage
getDescription()
Condiments, have internal reference to the reference (Beverage beverage)
add beverage.getDescription + „with Milk", so we decorate object with some new functionality.

In this example, we are using inheritance to achieve the type matching, but we are not using inheritance to get the behavior. We are acquiring new behavior not by inheriting it from a superclass, but rather by composing objects together. With composition, we can mix and match decorators any way we like at runtime.

Decorator vs Composite: Decorator also uses a containment strategy, but Decorators add or modify functionality of a single container. The point of Composite is to make it easier to manipulate a set of contained objects. Decorator adds responsibilities, Composites never do. Decorator and Composite patterns can often co-exist. They can co-exist and work hand-in-hand in supporting composition-based solutions to design patterns. A design context that is particularly good for these patterns is the development of some drawing feature.

Observer (Publish/Subscribe)

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

Use cases: when you want multiple objects to get notified of changes in the state of another object. The notifying object (publisher) send an event (publication) to all its observers (subscribers).

Advantages: 1) the Observer Pattern provides an object design where subjects and observers are loosely coupled. Observer nicely isolates subsystems, since the subjects in the subsystems don't need to know anything about each other except that they implement a certain interface (the Observer interface). This isolation makes the code much more reusable.

2) New observers can be added at any time. Because the only thing the subject depends on is a list of objects that implement the Observer interface, we can add new observers whenever we want. We can replace any observer at runtime with another observer or remove it and the subject will keep working.

3) Subjects or Observers can be reused independently of each other, since they are not tightly coupled.

4) Changes to either the subject or an observer will not affect the other. Because the two are loosely coupled, we are free to make changes to either, as long as the objects still meet their obligations to implement the Subject or Observer interfaces.

5) We never need to modify the subject to add new types of observers. If we have a new concrete class that needs to be an observer we don't need to make any changes to the subject; all we have to do is implement the Observer interface in the new class and register as an observer. The subject doesn't care, it will deliver notifications to any object that implements the Observer interface.

6) Because the subject is the sole owner of the data, the observers are dependent on the subject to update them when the data changes. This leads to a cleaner Object Oriented design than allowing many objects to control the same data.

Disadvantages: 1) Implementing the Observer Pattern might make the code too complex.

2) Notifying all the observers whenever the subject changes state can be computationally intensive, especially if there are a large number of subjects and observers.

3) The order of notifications is important, but the Observer does not guarantee it. If the order is important for the correct functioning of the program, then the program might fail.

4) There is no guarantee that a subscriber won't be notified of an event after the subscriber cancels its subscription. Memory leaks are easily created by passing references to observers, and they can

happen if the observer fails to unregister from the subject when it no longer needs to receive notifications.

5) Publication events can propagate alarmingly when observers are themselves publishers.

Code structure:

```java
// Subject interface
public interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}

// Observer interface
public interface Observer {
    void update(Subject subject);
}

// Concrete subject class
public class ConcreteSubject implements Subject {
    private List<Observer> observers;
    private int state;

    public ConcreteSubject() {
        observers = new ArrayList<>();
    }

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(this);
        }
    }

    public int getState() {
        return state;
    }

    public void setState(int state) {
        this.state = state;
        notifyObservers();
    }
}

// Concrete observer class
public class ConcreteObserver implements Observer {
    private int state;

    @Override
    public void update(Subject subject) {
        state = ((ConcreteSubject) subject).getState();
        System.out.println("Observer: " + state);
    }
}
```

Technical aspects: The observers have subscribed to (registered with) the Subject to receive updates when the Subject's data changes.

In the Observer we can include a helper method called notification method. If it is public, clients with references to the model get to control when notifications are issued. If private, it is assumed that the model is called at appropriate places in the state-changing methods of the model. Observer isolates subsystems, since the classes in the subsystems don't need to know anything about each other except that they implement certain "listener" interfaces.

We talk about inversion of control because, to find out information from the model, the observers do not call a method on the model, they instead wait for the model to call them back. The method called by the model on the observer is called a callback. A callback is not to tell observers what to do, but rather to inform them about some change in the model, and let them deal with it. In the Observer pattern, an observable object notifies its observer objects by calling their callback method(s).

To ensure that the model notifies observers whenever a state change occurs, two strategies are possible:

1) A call to the notification method must be inserted in every state-changing method, in this case the notification method can be declared as private. Downside: notifying observers with every state change may lead to some performance problem.
2) Clear documentation has to be provided to direct users of the model class to call the notification method whenever the model should inform observers. In this case the notification method needs to be non-private.

How to the observers access the information that they need to know about from the model? Two strategies:

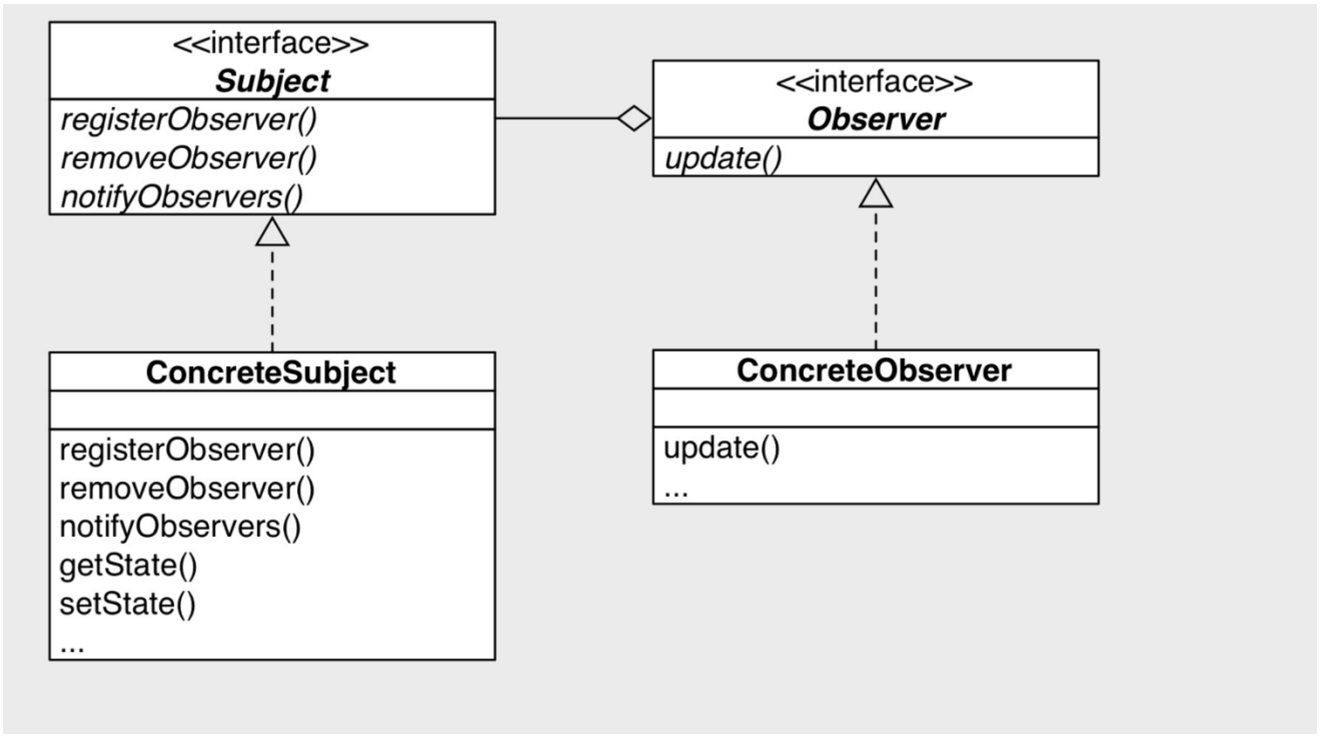
a) Push-data flow strategy: the components that generates the data sends it directly to the component that will use it. We make the information of interest available through one or more parameters of the callback. The model pushes data of a pre-determined structure to the observers. Assumption: we know in advance what type of data from the model the observers will require.

Pull-data flow strategy: the components that use the data request it from the component that generates it. We let observers pull the data they want from the model using query methods defined on the model. Drawbacks: it increases the coupling between observers and the model. By holding a reference to the model, observers have access to much more of the interface of the model than they need.

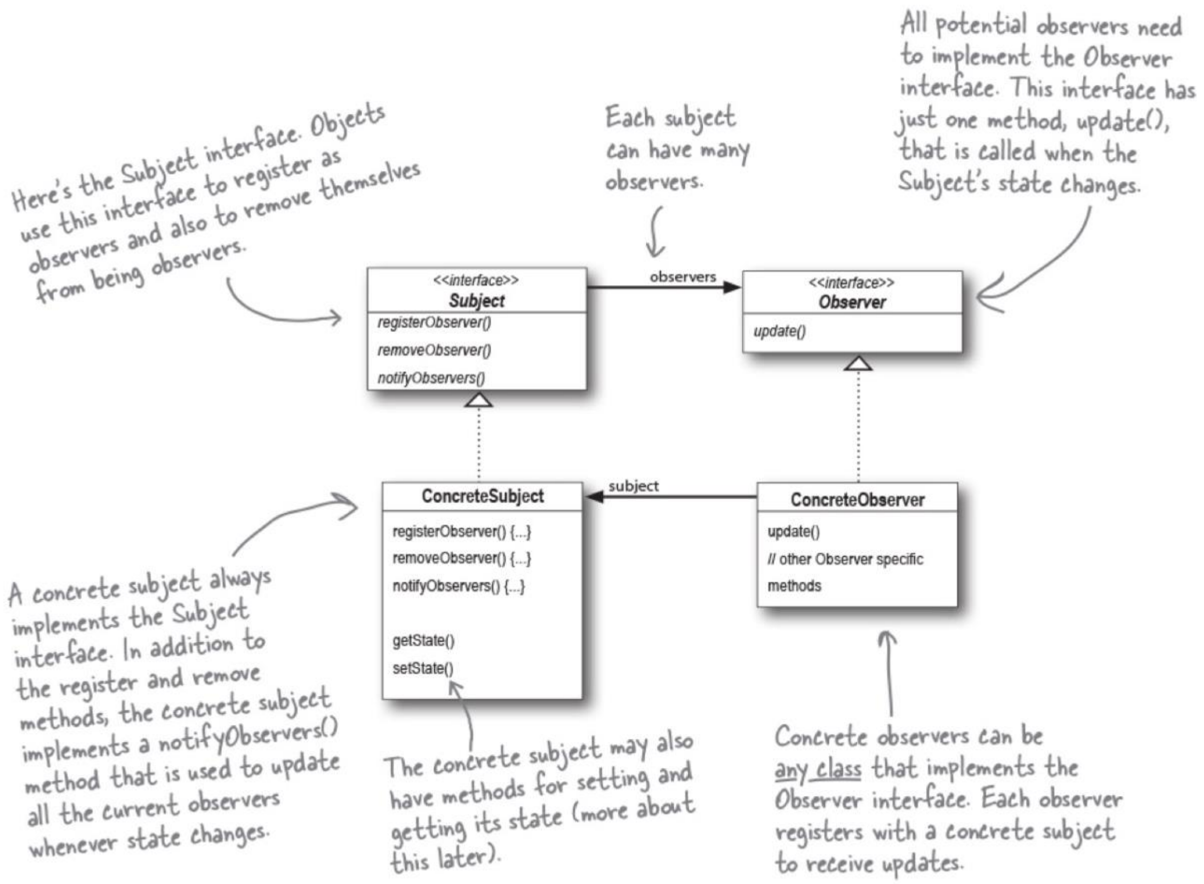
The push and pull strategies can be combined. For example, it is possible to specify a callback that includes a parameter for both data from the model and a reference back to the model. For simple design contexts it might be the case that the only information that needs to flow between the model and the observers is the fact that a given callback was invoked. In this case neither the pull nor the push strategy is required, since receiving the callback invocation is enough information for the observers to do their job. When starting out with the Observer it is recommended to have callbacks to return void. Pull is considered more correct.

Class Diagram:

1)



2)



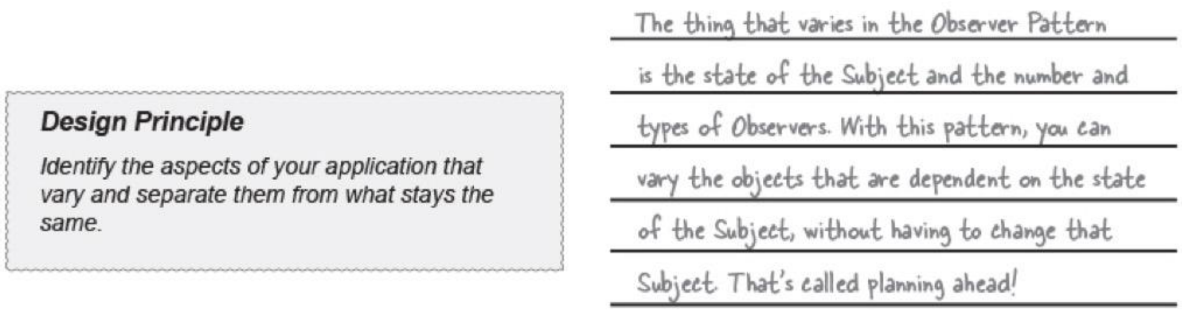
Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from being observers.
Each subject can have many observers.
All potential observers need to implement the Observer interface. This interface has just one method, update(), that is called when the Subject's state changes.
<<interface>>
Subject
registerObserver()
removeObserver()
notifyObservers()
observers
<<interface>>
Observer
update()
ConcreteSubject
registerObserver() {...}
removeObserver() {...}
notifyObservers() {...}
getState()
setState()
subject
ConcreteObserver
update()
// other Observer specific methods
A concrete subject always implements the Subject interface. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method that is used to update all the current observers whenever state changes.
The concrete subject may also have methods for setting and getting its state (more about this later).
Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

Examples: 1) When publisher (one) releases a new newspaper to which members are subscribed, the subscribers (many) will receive the new edition as long as they remain subscribed. If the observers unsubscribe when they don’t want newspapers anymore, they stop receiving them. While the publisher remains in business, people, hotels, airlines and other businesses constantly subscribe and unsubscribe to the newspaper.

2) A user interface that displays a list of emails, and an observer object might be used to update the list whenever a new email arrives or an existing email is deleted.

Observer vs strategy: Strategy objects define a strategy for performing some work. Observers do implement a notification strategy but, unlike Strategy objects, are not called from within methods to do work.

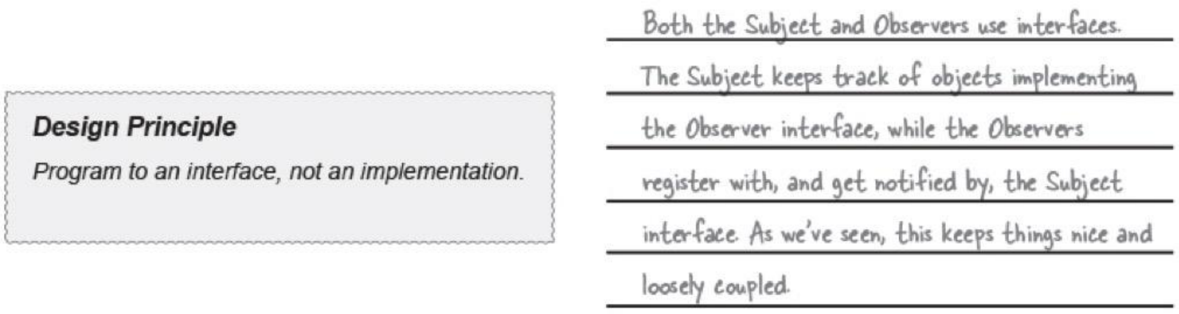
Fulfillment of design principles:



Design Principle

Identify the aspects of your application that vary and separate them from what stays the same.

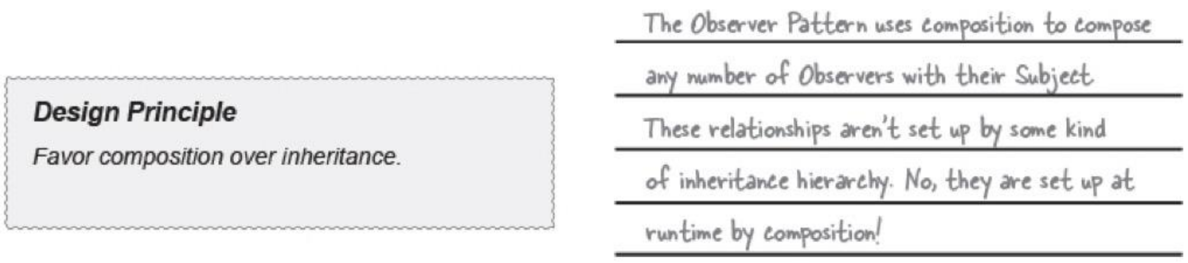
The thing that varies in the Observer Pattern is the state of the Subject and the number and types of Observers. With this pattern, you can vary the objects that are dependent on the state of the Subject, without having to change that Subject. That's called planning ahead!



Design Principle

Program to an interface, not an implementation.

Both the Subject and Observers use interfaces. The Subject keeps track of objects implementing the Observer interface, while the Observers register with, and get notified by, the Subject interface. As we've seen, this keeps things nice and loosely coupled.



Design Principle

Favor composition over inheritance.

The Observer Pattern uses composition to compose any number of Observers with their Subject. These relationships aren't set up by some kind of inheritance hierarchy. No, they are set up at runtime by composition!

<u>Adapter</u>

Definition: the Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

Use cases: when two systems want to communicate to each other but their interfaces are not compatible, an adapter would allow them to work together.

Advantages: 1) makes it easy to for existing classes to work with other classes without changing their code.

2) Loose coupling between the Adaptee and the adapting class, as it allows the two classes to work together without one being dependent on each other. An update or change in one class would not affect the other.

Disadvantages:

1) An adapter may not provide the exact functionality of the adaptee.

Code structure:

```java
// Target interface
public interface Target {
    void request();
}

// Adaptee class
public class Adaptee {
    public void specificRequest() {
        // Implementation of the specific request
    }
}

// Adapter class
public class AdapteeAdapter implements Target {
    private Adaptee adaptee;

    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    @Override
    public void request() {
        adaptee.specificRequest();
    }
}

// Client
public class Client {
    public static void main(String[] args) {
        Adaptee adaptee = new Adaptee();
        Target target = new AdapteeAdapter(adaptee);
        target.request();
    }
}
```

Technical aspects: the Adaptee is the object that does not support the desired interface and needs to be adapted to the Target interface. The Target is the interface that the client expects the Adaptee to support. The Adapter is the class that adapts the Adaptee to support the Target interface.The Client class uses the adapter class to communicate with the Adaptee, as if it was the Target interface.

How the Client uses the Adapter:

1) The client makes a request to the adapter by calling a method on it using the target interface.
2) The adapter translates the request into one or more calls on the adaptee using the adaptee interface.
3) The client receives the results of the call and never knows there is an adapter doing the translation. The Client and the Adaptee are decoupled, neither knows about the other.

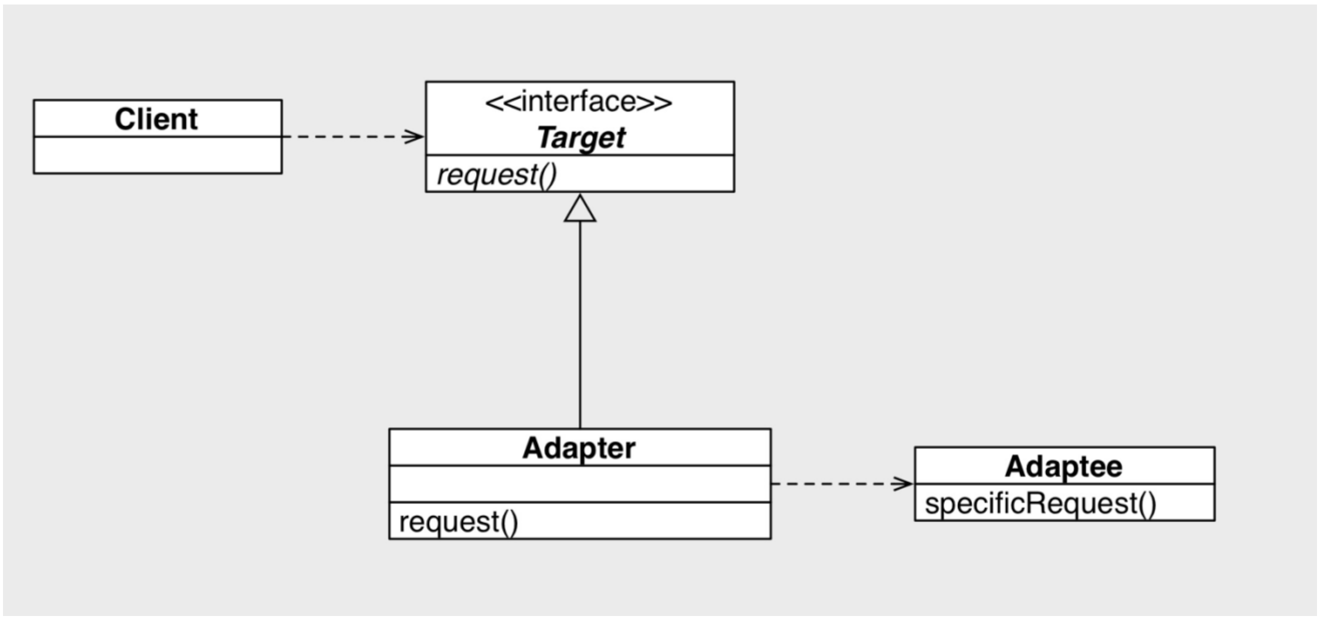
The Adapter does not help achieving inversion of control because it does not involve the reversal of control between the caller and called object.

There are two types of adapters: object adapters and class adapters.

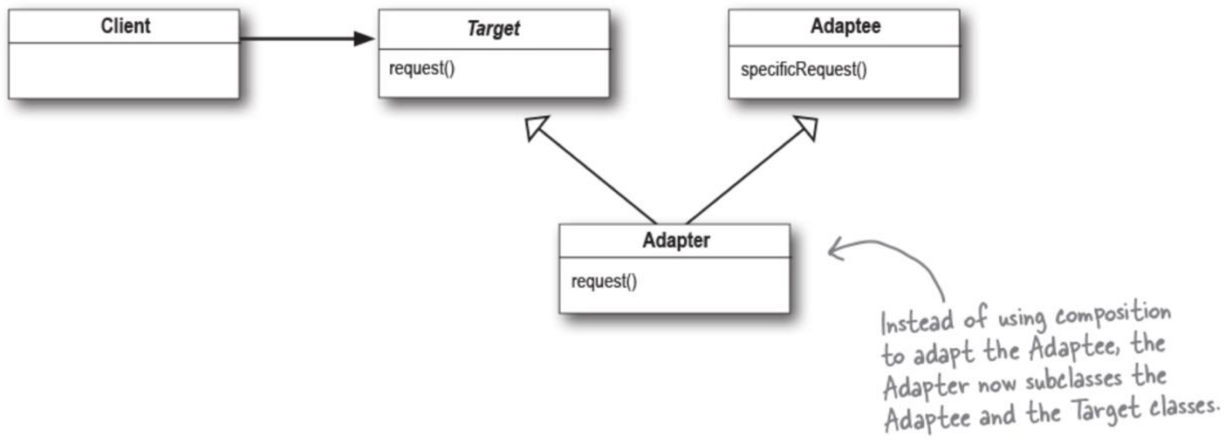
An object adapter uses composition to adapt one interface to another. It contains an instance of the class it is adapting, and delegates calls to the adaptee.

A class adapter uses inheritance to adapt one interface to another. It extends the class it is adapting and implements the target interface.

Class Diagram (for object adapters):



Class Diagram (for class adapters):


Client
Target
request()
Adaptee
specificRequest()
Adapter
request()
Instead of using composition to adapt the Adaptee, the Adapter now subclasses the Adaptee and the Target classes.

For class adapters you need multiple inheritance, which is not possible in Java.

Comment on diagram: the Client sees only the Target interface, The Adapter implements the Target interface and is composed with the Adaptee. All requests get delegated to the Adaptee.

Adapter vs Decorator: adapter wraps an object to change its interface, Decorator wraps an object to change its behavior, adding new behaviors and responsibilities to it. The encapsulated object in Decorator has the same interface as the container. Decorator modifies the behavior of some method or adds methods, but otherwise looks exactly like the wrapped object. Object adapters have different interfaces than the wrapped object and don't change its behavior. In both cases you pass a reference to the object but in Decorator we add new functionality while in Adapter we make sure that the code of one interface is adapted to the other interface.

<u>Command</u>

Definition: 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.

Use cases: when we want to separate the object that invokes a command from the object that executes the action that corresponds to the command or support undo (or redo) operations.

Advantages: 1) useful for tasks such as “undo” operations, Command gives a mechanism for actively rolling back state (generally not possible) by reversing side effects such as database updates.

2) Command decouples operations from the object that actually performs the operation.

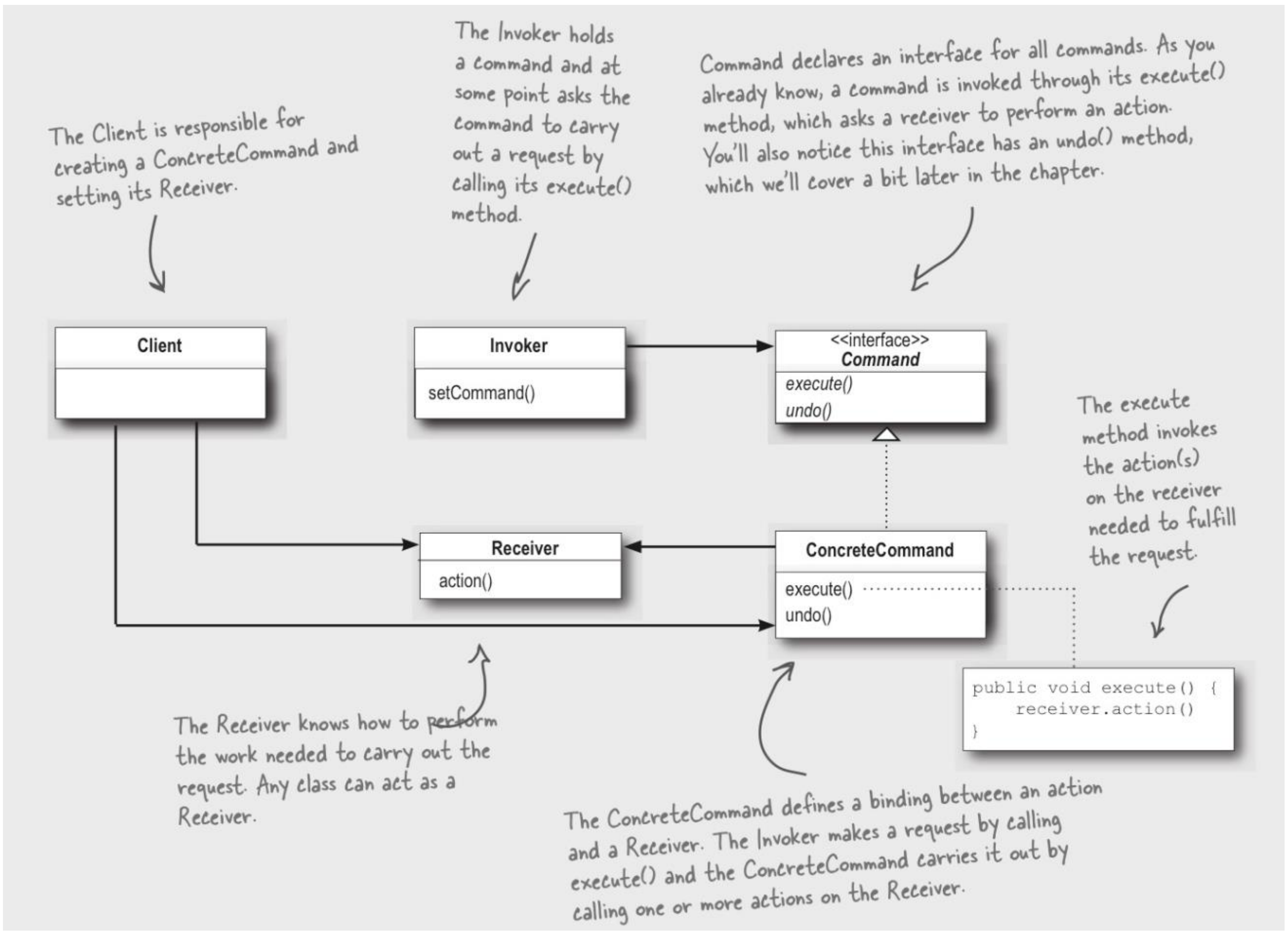
Disadvantages: the command pattern can be inflexible in situations where you need to modify the behavior of a command after it has been created. For example, if you want to change the receiver associated with a command, you will need to create a new command object.

Code structure:

```
// The command interface
public interface Command {
    void execute();
    void undo();
}

// The receiver class
public class Receiver {
    public void action() {
        // Perform some operation
    }

}

// The concrete command class
public class ConcreteCommand implements Command {
    private Receiver receiver;

    public ConcreteCommand(Receiver receiver) {
        this.receiver = receiver;
    }

    @Override
    public void execute() {
        receiver.action();
    }

    @Override
    public void undo() {
        receiver.undoAction();
    }
}

// The invoker class
public class Invoker {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void executeCommand() {
        command.execute();
    }
    public void undoCommand() {
        command.undo();
    }
}

// The client class
public class Client {
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        Command command = new ConcreteCommand(receiver);
        Invoker invoker = new Invoker();
        invoker.setCommand(command);
        invoker.executeCommand();
        invoker.undoCommand();
    }
}
```

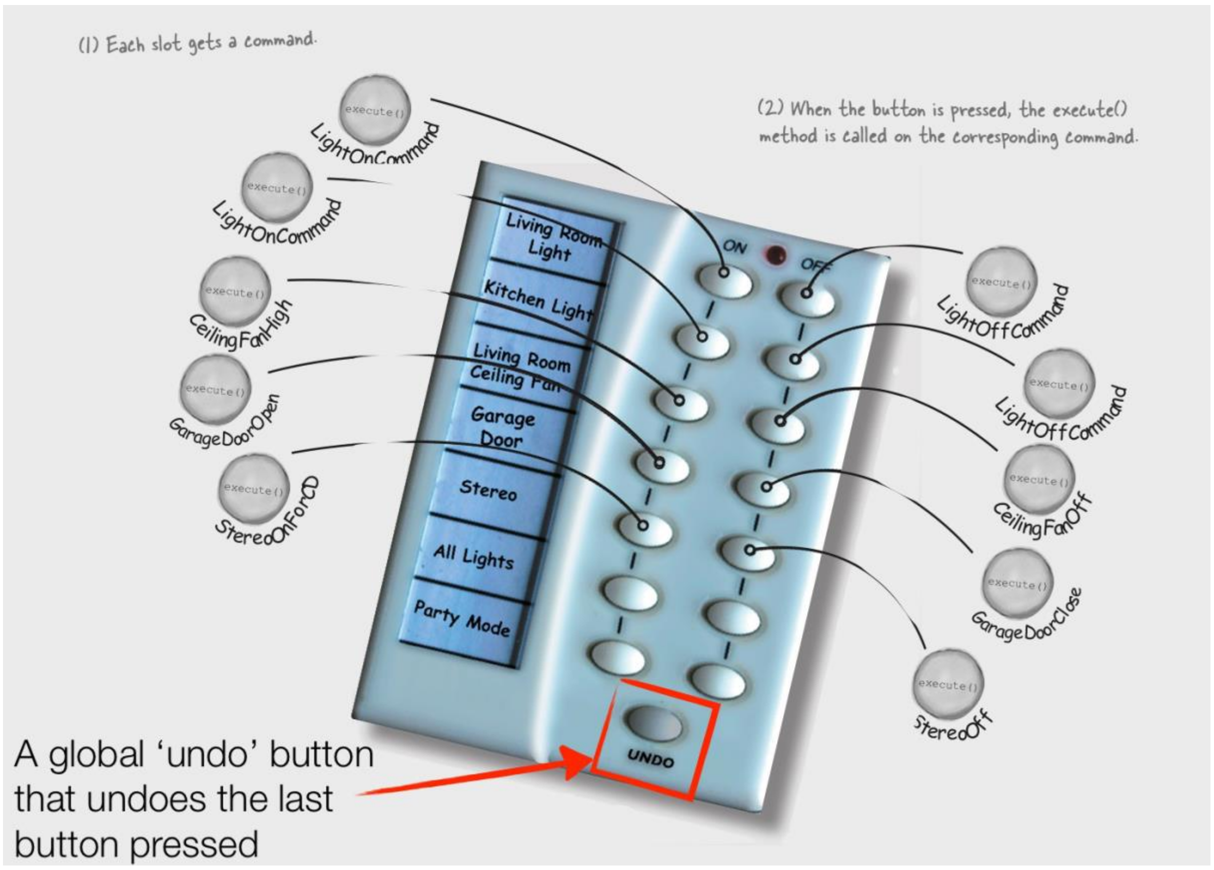
Technical aspects: Command defines an interface for executing an operation or a set of them through the execute() method. The command may also define other methods for managing the command, such as undo(). Concrete Command implements the command interface to perform the operation. It typically acts as an intermediary to a Receiver object. Invoker is the class that knows how to execute a command; it maintains a list of commands and has a method for executing them. The client is the object that created and configures the command and invoker objects, and then sends the requests to the invoker.Receiver knows how to perform the operation associated with the command. The execute method can also be of non-void type. To be able to undo an operation, one solution could be to store information about the previous state inside the command object. If the command object needs to access fields of the target object, it makes sense to add a factory method in the target object that returns instances of an anonymous function. To be able to alter the state of an object, it is a common approach to store a reference to this object in the command object. When applying the command pattern, be careful not to break the encapsulation of classes simply to allow command objects to operate on target objects. For designs where function objects need to be explicitly managed by client code, for example to store them or share them between code locations, the command design pattern provides a recognizable solution template.

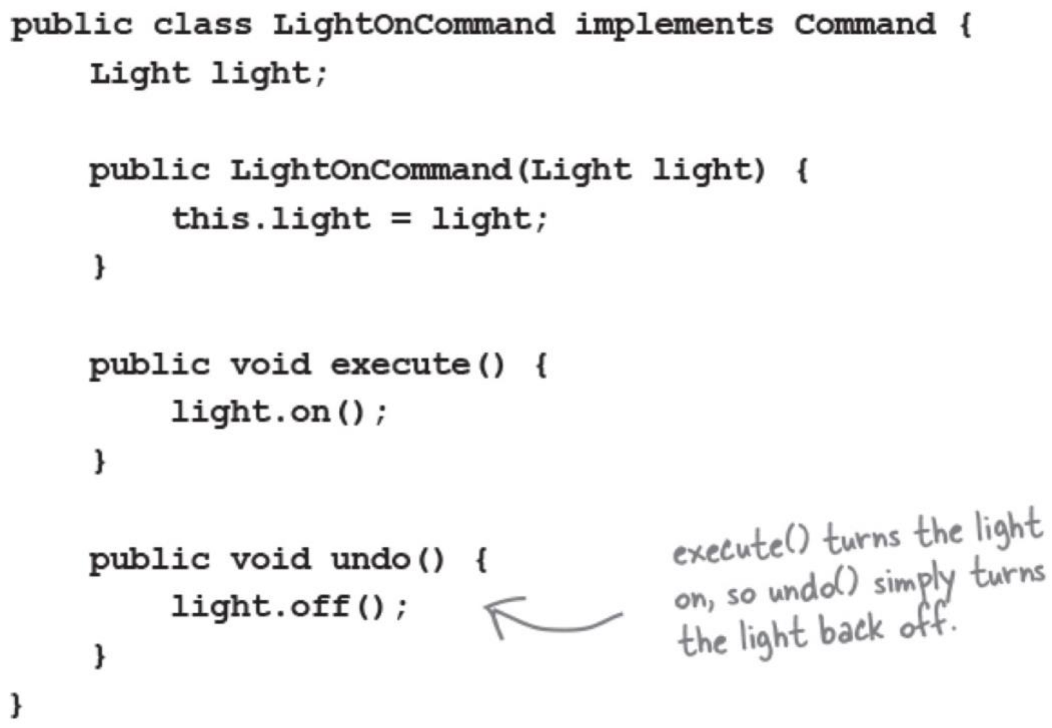
Class Diagram:


The Client is responsible for creating a ConcreteCommand and setting its Receiver.
The Invoker holds a command and at some point asks the command to carry out a request by calling its execute() method.
Command declares an interface for all commands. As you already know, a command is invoked through its execute() method, which asks a receiver to perform an action. You'll also notice this interface has an undo() method, which we'll cover a bit later in the chapter.
Client
Invoker
setCommand()
<<interface>>
Command
execute()
undo()
The execute method invokes the action(s) on the receiver needed to fulfill the request.
Receiver
action()
ConcreteCommand
execute()
undo()
The Receiver knows how to perform the work needed to carry out the request. Any class can act as a Receiver.
The ConcreteCommand defines a binding between an action and a Receiver. The Invoker makes a request by calling execute() and the ConcreteCommand carries it out by calling one or more actions on the Receiver.

```
public void execute() {
    receiver.action()
}
```

Example:





```java
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }

    public void undo() {
        light.off();
    }
}
```

execute() turns the light on, so undo() simply turns the light back off.

Command vs Strategy: the invoker of a Command doesn't know what the Command object does.. A Strategy object encapsulates a method for performing a specific action for the invoker.

Command vs Observer: Command objects are very generic. Observers are used solely for notification. Command is not an alternative to the observer pattern.

<u>State</u>

Definition: the State pattern allows an object to alter its behavior when its internal state changed. The object will appear to change its class. It encapsulates state-based behavior and delegates it to the current state.

First sentence: since the State pattern encapsulates state into separate classes and delegates to the object representing the current state, we know that behavior changes along with the internal state.

Second sentence: from the client perspective if an object can completely change its behavior, then it appears that it is actually instantiated from another class. In reality, composition is used to give the appearance of a class change by simply referencing different state objects.

Use cases: the State Pattern is used when we want to implement finite state machines (systems that have a finite number of states), or when we need to handle a large number of conditional statements in a correct way (avoiding the switch statement antipattern).

Advantages: 1) prevents the Switch statement antipattern, which consists of using a large switch statement or a series of if-else statements to execute different code paths based on the value of a particular variable; and if a change request comes, code is really hard to be modified without changing it and causing bugs in code that was working before. The State Design Pattern eliminates these long and hard-to-maintain switch statements when an object changes its state.

2) State machines are easier to maintain since all the behavior for a given state is in one place.

3) The behavior of each state is localized into its own class.

4) Closes each state for modification, and lets the Context object open to extension by adding new state classes -> Open/Closed Principle.

5) These class structure maps more closely to the object (e.g. Gumball Machine) diagram and is easier to read and understand.

Disadvantages:

1) It increases the number of classes in the system and maintenance problems might arise.

2) Code may become too complex, especially if only a few methods change behavior with state. But if we choose to have a program that has a lot of state and you decide not to use separate objects, you'll instead end up with very large, monolithic conditional statements. This makes your code hard to maintain and understand.

Code structure:

```java
// Define the base interface for states
interface State {
    void handle();
}

// Define a concrete state class
class ConcreteStateA implements State {
    public void handle() {
        System.out.println("Handling state A");
    }
}

// Define another concrete state class
class ConcreteStateB implements State {
    public void handle() {
        System.out.println("Handling state B");
    }
}

// Define the context class that maintains the current state
class Context {
    private State currentState;

    public Context() {
        currentState = new ConcreteStateA();
    }

    public void setState(State state) {
        currentState = state;
    }

    public void handle() {
        currentState.handle();
    }
}

// Test the state design pattern
public class Main {
    public static void main(String[] args) {
        // Create an instance of the context class
        Context context = new Context();
        // Change the state of the context and invoke the handle method
        context.setState(new ConcreteStateB());
        context.handle(); // Output: "Handling state B"
    }
}
```

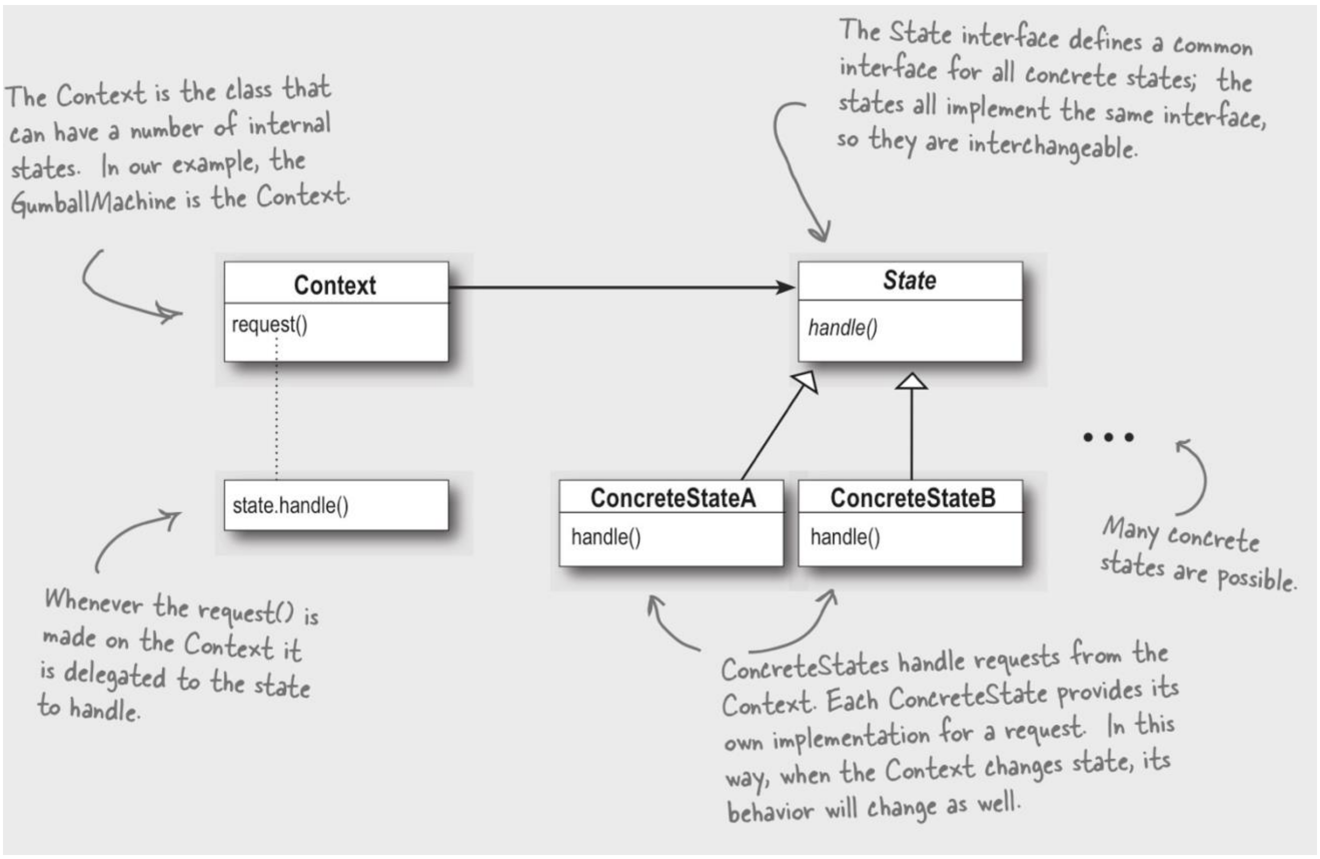
Comment: the Context class keeps a reference to the current state object, and delegates the handling of its behavior to the state object. The State Interface defines the method for handling the behavior, and the Concrete State classes implement this handle() method in different ways. The Context class allows the state of the object to be changed at runtime by simply replacing the current state object with a different one. This allows the behavior of the Context class to change dynamically based on its internal state.

Technical aspects: define a State interface (or abstract class) that contains a method for every action for an object (e.g. a Gumball Machine). Implement a State class for each state of that object; these State classes will be responsible for the behavior of the object when it is in the corresponding state. We can get rid of all the conditional code (if-else statements) and instead delegate the work to the state classes. The Context gets its behavior by delegating to the current state object it is composed with. The Context changes its behavior as its state changes.The current state of the object is always one of the states implementing the State interface. The ConcreteStates do not always decide what state to go to next, an alternative is to let the Context decide on the flow of state transitions. When the state transitions are fixed, it is appropriate to place them in the Context class, but when the transitions are more dynamic it is better to place them in the state classes themselves. The disadvantage of having state transitions in the state classes is that we create dependencies between the state classes.

The states are used by the Context to represent its internal state and behavior, so all requests to the states come from the Context. Clients don't directly change the state of the Context, so they don't interact directly with the states. It is the Context's job to supervise its state, and you don't usually want a client changing the state of a Context without that Context's knowledge.

If there are lots of instances of the Context in the program, it is a good idea to share the state objects across them. The only requirement is that your state objects do not keep their own internal context; otherwise, you'd need a unique instance per context. To share your states, you'll typically assign each state to a static instance variable. If your state needs to make use of methods or instance variables in your Context, you'll also have to give it a reference to the Context in each handler() method.

Class Diagram:


The Context is the class that can have a number of internal states. In our example, the GumballMachine is the Context.
The State interface defines a common interface for all concrete states; the states all implement the same interface, so they are interchangeable.
Context
request()
State
handle()
state.handle()
ConcreteStateA
handle()
ConcreteStateB
handle()
Many concrete states are possible.
Whenever the request() is made on the Context it is delegated to the state to handle.
ConcreteStates handle requests from the Context. Each ConcreteState provides its own implementation for a request. In this way, when the Context changes state, its behavior will change as well.

State vs Strategy: The State and the Strategy Pattern have the same class diagram, but they differ in intent. The state objects do implement a strategy for implementing a single state, but that strategy is not provided by an outside entity. The state design pattern allows an object to alter its behavior when its internal state changes. It does this by encapsulating the behavior associated with each state within a separate class, and allowing the object to switch between these classes at runtime. The main goal of the state design pattern is to allow an object to change its behavior based on its internal state, without requiring the client code to be aware of the specific implementation details of the behavior.

The strategy design pattern, on the other hand, allows an object to alter its behavior by encapsulating a group of algorithms within separate classes and allowing the object to switch between these classes at runtime. The main goal of the strategy design pattern is to allow an object to change its behavior by selecting from a range of algorithms, without requiring the client code to be aware of the specific implementation details of the algorithms.

The Strategy Pattern has to be thought as a flexible alternative to subclassing; since if inheritance is used to define the behavior of a class, then we’re stuck with that behavior even if we don’t need to change it. With Strategy you can change the behavior by composing with a different object. The State Pattern has to be thought as an alternative to putting lots of conditionals in the code. By encapsulating the behaviors within state objects, you can simply change the state object in content to change its behavior.

Design principles:

- Single Responsibility Principle (SRP): This principle states that a class do one thing and thing only (only have one responsibility), and have only one reason to change. Every responsibility of a class is an area of potential change; more than one responsibility means more than one area of change. Cohesion is a measure of how closely a class supports single purpose or responsibility.

- Identify the aspects of your application that vary and separate them from what stays the same (principle of separations of concerns). The separation of concerns designing systems that are more flexible and easier to modify and maintain over time. It allows to focus on specific parts of the code without having to worry about the impact on other parts of the system, which can help us to develop and test our code more efficiently.

- Program to an interface, not to an implementation. This principle advises to design our code in such a way that it depends on abstractions (such as interfaces) rather than concrete implementations. Rather than making a piece of code depend on a specific and concrete implementation, we should write our code to program interfaces. This principle enforces loose coupling, since by depending on abstraction rather than concrete implementations, the code becomes less tied to specific implementations. This makes it easier to change implementations without having to make changes to the code that depends on them. It also reinforces reusability since by depending on abstractions, the code can be reused in different contexts.It also become easier to write tests for the code, as we can use mock implementations of the abstractions to test the code in isolation.

- Favor composition (has-A) over inheritance (is-A). This principle states that we should prefer to combine simple objects to create more complex ones. One reason for this is that inheritance can lead to tight coupling between classes, where changes to the superclass can have unintended consequences on the subclass. Composition, on the other hand, allows for more flexibility and modularity, as objects can be composed and reused in different ways without affecting one another. Example: to represent a zoo we could have an Animal class and different types of animals inheriting it, or a big zoo class with all the animals as composites. If we would need to add the feature to classify animals as carnivores or herbivores we would need to change animal class and its subclasses if we used composition, while with composition I could make a Diet class which has a getDietType method and would make core more flexible and easily testable. Not only does composition let you encapsulate a family of algorithms into their own set of classes, but it also lets you change behavior at runtime as long as the composing object implements the correct behavior interface.

- Open-Closed Principle: classes should be open for extension but closed for modification. It should be possible to add new functionality to classes by extending them without needing to change the existing code. When following this principle, the risk to break the existing functionality when making changes to the code or adding new capabilities is reduced. To achieve this principle classes should be designed in a way that allows them to be extended through inheritance while making sure at the same time that the main functionality of the class is not changed. Following the Open-Closed principle usually introduces new levels of abstractions, which add complexity to the code. Applying the Open-Closed principle everywhere is wasteful and unnecessary, and can lead to complex, hard-to understand code.

- Strive for loosely coupled designs between objects that interact (loose coupling). The loose coupling principle that states to strive for loosely coupled designs between objects that interact. The idea of loose coupling is that components of a system should be designed in such a way that they have minimal dependencies on each other, and changes to one component should have minimal impact on the other components. The major benefit of loose coupling is that it makes the system more flexible and easier to change, because changes to one component do not require changes to other components, so the interdependency between objects is minimized. When two objects are loosely coupled, they can interact, but they have very little knowledge of each other.

- Single Responsibility Principle (SRP): this principle states that a class do one thing and thing only (only have one responsibility) and have only one reason to change.

- Principle of Least Knowledge: talk only to your immediate friends. When designing a system for any object, be careful of the number of classes it interacts with and also how it comes to interact with those classes.

- Liskov Substitution Principle (LSP): This principle states that a subclass (or an implementing class of an interface) should be able to interchange and substitute/replace its superclass without affecting the correctness of the program, so it should be able to support each operation performed by the superclass. Objects of the superclass should be used in the same way as objects of the subclass (e.g. rectangle and shape, and not bankingaccount and checkingaccount). Keep in the interface or superclass only the methods that each implementing subclass supports, and exclude operations that at least one subclass does not support.

- Interface Segregation Principle (ISP): client code should not be forced to depend on/implement methods of an interface it does not need. ISP allows developers to create systems with loose coupling. Break up large interfaces into smaller ones if you find that many methods of a type are not used in certain parts of the code or are not needed by the client.

Encapsulation: protect the internal state of an object from being accessed or modified directly by client code. This is achieved by declaring the data fields of the object as private and providing public methods for accessing and modifying the data. Encapsulation allows you to control the way in which client code interacts with the object and can prevent client code from directly modifying the internal state of the object in unintended ways.

Encapsulation is good for information hiding, low (or loose) coupling and high cohesion, since it groups related data and functionality together (into a single unit) to achieve a single and well-defined purpose.

Polymorphism: ability of an object to take many forms. It allows to write code that can operate on multiple types of objects in a uniform way, without needing to know the specific type of object. It is achieved through inheritance, an interface or function overloading.

Appendix A

Every java program must have a function called public static void main(String[] args); this is the function where client code is run.

Java is not a dynamic language, Java is a static language (variables are known at compile time).

Two major data types: *primitive* types and *reference* types. Primitive types are used to represent numbers, Strings or boolean values: the variables of a primitive data type store the actual data that represents the value. Reference types represent more complex arrangements of data as defined by classes. A variable of a reference type T stores a *reference* to an object of type T. Values of reference types are not the data itself but a reference to its data. Copying a value means sharing a reference, Arrays are also reference types.

To create a new object of a class you need to have the new word, e.g.: Book book = new Book();

Dereferencing operator is . -> e.g. String title = book.title, where title is called the instance variable (or field).

Static fields: fields that are not associated with any object. A single copy of the field is created when the corresponding class is loaded by Java. If any property is different for every object of that class than it would be non-static variable and if any property is same for all the objects of that class than it would be static.

Static means that the value of the static variable is the same for each object of the class. Static variables don't need an instance of the object to be defined, a static variable can be accessed without creating an instance of the class (since they are class variables). It is a variable that is common or shared by all the objects. Static methods cannot access attributes of the class (we cannot use the this keyboard to access instance variables or methods). Since static methods don't have access to the his keyword, they cannot be used to implement polymorphism. A static method is a method for which the instantiation of an object is not needed to invoke it, it can only call other static methods. Inner classes can access both static and non-static members of the outer class. A static class can access only the static members of the outer class. A static class can only be declared if it is a nested class. Objects of a static class can be instantiated with first instantiating the inner class. Static classes cannot be extended, so it is not possible to create subclasses from a static class; this is because static classes do not have a behavior on their own so it's not possible to modify it in the subclasses.

Access modifiers: keywords that control what parts of the code can *access* certain program elements (e.g. classes, fields, methods). The idea of restricting access to fields is very similar to that of visibility and scope for local variables. Examples: public, protected, private, default (package-private access).

Scope: lexical region that acts as a sort of boundary for variables. In Java, scopes are defined using curly braces. For example, in the main method: {int a = 0;} and we cannot write {int b = a} since a is not visible in the second scope.

Members marked as public are visible anywhere in the code. In contrast, members marked private are only visible within the scope of the class, namely, between the left curly brace that begins the declaration for the class body and the last right curly brace of the class declaration. If you make any variable as final, you cannot change the value of final variable. Final variables can only be declared/ in the Constructor of a class (compile-time) or when declaring them, but they cannot be changed at compile-time. Attempting to reassign the value of the field anywhere else in the code leads to a compilation error. The final keyword goes a long way in limiting the state space of an object, because any field marked as final can only take a single value.

If we use the final keyword for reference types, the value stored in a variable is a reference to an object. So, although it is not possible to reassign a final field, it is possible to change the state of the object referenced (if the object is mutable).

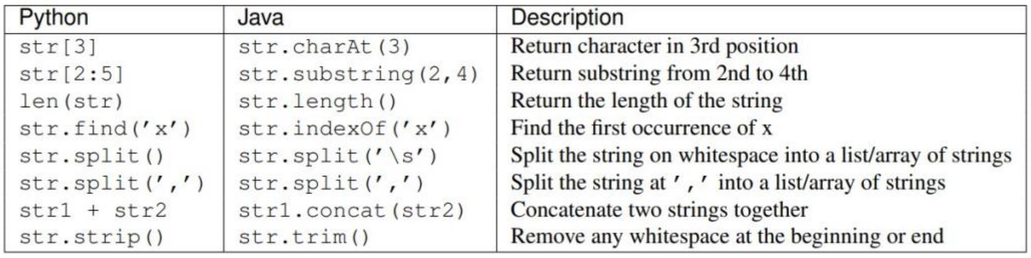
Local variables (including method parameters) can also be declared as final, not only instance variables. The case is much less clear because local variables are not long-lived, meaning that they only exist for the duration of the execution of the code in their scope.

Type members declared as protected are only accessible within methods of the same class, classes in the same package, and subclasses in the same package. Protected methods are also accessible via the extends keyword (inheritance).

To respect the principles of encapsulation, accessibility of fields should be minimized. Increasing the visibility of a field from private to protected has a negative impact on encapsulation, because with protected it is possible to refer to the field, and thus mutate the object it refers to, from many different classes instead of just one. To circumvent this issue we can for example use super calls.

Exception handling: with throw keyboard, e.g. if (month < 1 || month > 12) throw new InvalidDateException();

Operations on Strings: Java vs Python



Python	Java	Description
`str[3]`	`str.charAt(3)`	Return character in 3rd position
`str[2:5]`	`str.substring(2,4)`	Return substring from 2nd to 4th
`len(str)`	`str.length()`	Return the length of the string
`str.find('x')`	`str.indexOf('x')`	Find the first occurrence of x
`str.split()`	`str.split('\s')`	Split the string on whitespace into a list/array of strings
`str.split(',')`	`str.split(',')`	Split the string at `','` into a list/array of strings
`str1 + str2`	`str1.concat(str2)`	Concatenate two strings together
`str.strip()`	`str.trim()`	Remove any whitespace at the beginning or end

A constructor has the same name as that of the class and does not have any return type. You can think of a constructor as a special kind of method that has the same name as the class and does not have a return type.

Constructor types:

No-Arg Constructor - a constructor that does not accept any arguments

Parameterized constructor - a constructor that accepts arguments

Default Constructor - a constructor that is automatically created by the Java compiler if it is not explicitly defined.

A constructor cannot be abstract or static or final, but it can be private.

Chapter 1 – Introduction to Software Design

The verb *to design* refers to the process we follow when we design software, and the noun *a design* refers to the outcome of this process.

The process of software design is the construction of abstractions of data and computation and the organization of these abstractions into a working software application.

A design artifact is an external representation of one or more design decisions.

A software development practice is a well-understood way of doing something to achieve a certain benefit. Examples include version control, coding conventions and refactoring (= rewrite code without changing its functionality)

Design knowledge can be captured in source code, code comments, specialized documents, discussion forms and models.

Software design is a highly heuristic process: it consists of iterative problem solving guided by experience, general principles and design techniques.

The Unified Modeling Language (UML) is a modeling language organized in terms of different types of diagrams. Using UML can be an effective way to illustrate different aspects of software without getting caught up in details. UML diagrams are models, this means that they are not intended to capture every single detail of a solution.

Not all development teams use UML. However, those who use it use it in different ways for different reasons. For example, UML can be used to produce formal design documentation in waterfall-type development processes. Others use the UML to describe enough of the software to be able to automatically generate the code from the models, following the idea of generative programming. UML diagrams can be used as a language to communicate among software developers.

UML reduces risks by documenting assumptions (domain models, architecture, design, implementation). It represents industry standard and is well-defined. UML is open.

It is important to remember that the point of UML diagrams is not to be an exact translation of the code. As models, they are useful to capture the essence of one or more design decisions without having to include all the details. The concept of navigability, represented with an arrowhead, models how code supports going from objects of one type to objects of another type. Navigability can be unidirectional, bidirectional or unspecified. In UML there is no good way to indicate that a class does not have a certain member (field or method). It would be necessary to include it using a note.

Chapter 2 - Encapsulation

An essential technique in software design is to decompose a system into distinct, manageable abstractions. For a decomposition to be useful, the resulting abstractions have to be well isolated from each other. For good design, an idea that should be inseparable from that of software abstraction is *encapsulation.* The idea of encapsulation is to enclose something as if it were in a capsule. The idea of encapsulation is so to hide the internal implementation of an abstraction behind an interface that tightly controls how an abstraction can be used.

Benefits of encapsulation: makes a piece of code in isolation easier to understand and makes the use of the isolated part by the rest of the code less error-prone, and it makes it easier to change one part of the code without breaking anything. The equivalent of a shell, in software design, is called interface. Following the principle of information hiding, encapsulated structures should only reveal the minimum amount of information that is necessary to use them and hide the rest. Information hiding follows the principle to provide only the information that is absolutely needed and hide the rest. A client code is any code that is not part of the definition of this element. With encapsulation, we avoid that any variable that forms part of an object can be accessed indiscriminately. Encapsulation is obtained by making variables protected and private, with setters and getters. Having a class that is mostly accessed through getters and setters points to a design weakness, because the abstraction the object represents is not effective. This problem is also known as the *Inappropriate Intimacy* problem.

Advantages of Encapsulation:

1) It makes it easier to understand a piece of code in isolation.
2) It makes the use of the isolated part by the rest of the code less error prone.
3) It makes it easier to change one part of the code without breaking the rest of the code (loose coupling)

Abstraction: conceptual building block for a software system. Examples of common abstractions in computing include data structures and operations (sorting, iterating). Abstractions can also refer to ideas in the problem domain, for example a playing card. Defining an abstraction means deciding what the abstraction represents, and what it will look like in terms of source code.

To facilitate code understanding and help avoid programming errors, the representation of values should ideally be tied to the concept they represent. For example, the general type int maps to the concept of an integer (a type of number), not that of a playing card.

The tendency to use primitive types (like int, strings, etc.) to represent abstractions is a common problem in software design that has been captured by the antipattern *Primitive Obsession.*

The best tools at our disposal to encode abstractions are enumerated types (enums): they are a simple yet effective feature for creating or implementing robust designs. They help avoid Primitive Obsession and generally make the code clearer and less error prone. Strings and enums are immutable, lists are mutable. Enums are immutable since their fields as final and cannot be changed once they are initialized. Enumerated types should be used to represent a value in a collection of a small number of elements that can be enumerated.

Object: mechanism to group variables together and access their values through the process of dereferencing. An object is an instance of a class; a class represents a blueprint for its objects.

To provide encapsulation, methods should reveal as little as possible about implementation decisions meant to be encapsuled and should not allow to change variables that should not be accessed. Public methods of a type (a class) represent what client code can do with objects of the type, and the design of all other (non-public fields) and methods remain hidden from that client. Encapsulation makes it difficult or impossible for client code to corrupt the value of a variable.

Object Diagram: it is a type of UML diagram that represents objects and their relations. It is similar to a class diagram, but it represents the actual instances of classes and their relationships, rather than the classes themselves.

In an object diagram, a rectangle represents an object, with its name and type indicated as name:type. It is useful to have at least one of the two (name or type, they are optional). In UML diagrams in general, the name of objects (as opposed to classes) is underlined. Objects can contain *fields*, which are variables inside a class that can contain a value of a primitive type or a value of a reference type. Object diagrams are part of the UML standard. Classes in object diagrams are not visualized.

When the value is a reference type, it is represented as a directed arrow to the object being referenced.

Example of an object diagram:

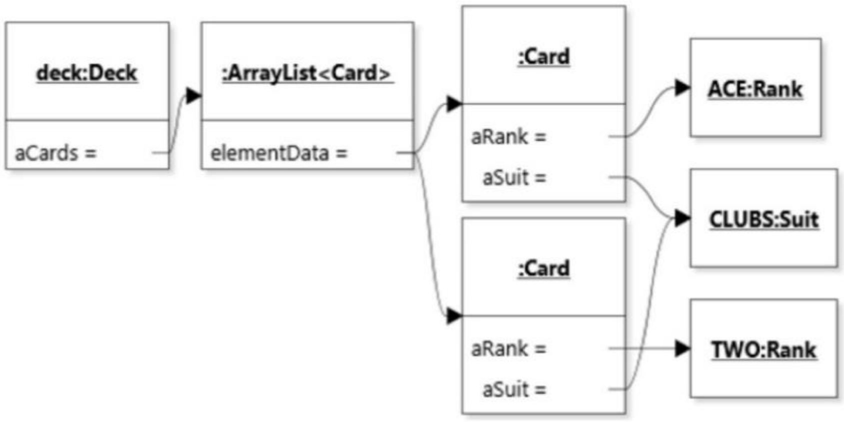


Fig. 2.1 Object diagram showing a detailed model of the object graph for a deck of cards

Objects are *immutable* if their class provides no way to change their internal state after initialization. A class that yields (generates) immutable objects is called an *immutable class.* For the designer of a class, the only way to ensure immutability is to carefully design the class to prevent any modification (e.g. by providing no setter methods, leaking no reference, etc.). Leaking references occurs when a program holds references to objects and reserves memory for them, but they are no longer required or existing. The main problem with leaking references is that it breaks encapsulation of a class, and memory leaks occupy memory for nothing.

Immutable objects have many advantages, the immediate one is to support sharing information encapsulated in an object without breaking encapsulation. Immutable objects are thread-safe, so there is no need of synchronization. An immutable class is useful when we want to have a class that behaves in the same way, no matter in which state the client is, an example is with Enums.

The client can still access the code of an immutable class if its field are accessible to the client (e.g. a string name of a player can still be changed if it is not private/protected nor final but still a String.

In Java, it is only possible to ensure that a class is immutable through careful design and inspection.

A common technique to copy objects is to use a *copy constructor.* The idea is to design a constructor that takes as argument an object of the same class, and to copy matching field values.

Input validation is important, so that the user cannot for example set parameters equal to null or assign variables to invalid values. Input validation ensures that client code does not misuse an object, but it may not be necessary if the client code is written in a way that it precludes erroneous values. Everywhere a variable can be assigned, with input validation we check whether the input value is a null reference or invalid and throw an exception if that is the case.

The main idea of design by contract is for method signatures (and related documentation) to provide a sort of contract between the client (the caller method) and the server (the method being called). This contract takes the form of a set of *preconditions* and a set of *postconditions*. A precondition is a predicate that must be true before a method starts executing. The predicate typically involves the value of the method's arguments, including the state of the target object upon which the method is called. Similarly, postconditions are predicates that must be true after the execution of a method is completed. The contract is basically that the method can only be expected to conform to the postconditions if the caller conforms to the preconditions.

Preconditions and post-conditions are specified with @pre and @post. It is possible to make pre- and postconditions *checkable* using the assert keyword. Always specify contracts for public methods.

Design by contract supports clear *blame assignment* while debugging: if a precondition check fails, the client (caller method) is to blame. If a postcondition check fails, the actual method being called is to blame.

In design by contract, it is preferrable to check input validation with asserts rather than by throwing Exception Errors.

Assertions can be used anywhere in the code, not just for pre- or postconditions, in fact whenever an assertion fails, we know exactly where the problem is, and we can thus save on debugging time.

The addition of preconditions to a method's interface relieves us of the requirement to handle the condition. When designing method interfaces, it is important to decide whether the method will be in charge of rejecting illegal values, or whether these will simply be specified as invalid.

Diligent programmers can help eliminate ambiguities by stating the precise range of allowed values in a method's documentation.

Assertions want to make sure that the client is in the condition the code expects, so that he fulfills them (pre, post). Errors are always enabled; assertions might not be.

Errors are situations from which it is not possible to recover (but can still be raised), whereas from Exceptions we may have a way to handle them.

Throwable class has two children: Exceptions and errors.

Finally block in try catch is always executed (either if an exception is thrown or not), its execution cannot be skipped.

The compiler does not require that a method declares to throw two exceptions (if thrown in code of method) if one exception extends the other exception or one of the exceptions extends a type of runtime exception (e.g., NullPointerException, IllegalArgumentException, et.c) so it is not needed to handle it in a try-catch block nor to declare it with the throws keyword, since the compiler does not check to see if a method handles or declares it (unchecked exceptions).

Mocking: process of creating objects that simulate the behavior of other objects.

The ArrayList class is a subtype of List, so it may be declared as: public class ArrayList<E> implements List<E> {...}. The ArrayList class has concrete implementations for all the methods one would expect of a List-like object (since it implements the List interface). The class List cannot be declared as: public class List<E> implements ArrayList<E> {...}.

Do not define nested classes, define each class in a separate file, but not nested. You can also combine multiple classes in one file, but there can be only one public class per file.

Chapter 3 – Types and Interfaces

An interface to a class consists of the methods of that class that are accessible to another class. The interface keyword is used to declare a special type of class that only contains abstract methods. To access the interface (or abstract class) methods, the interface must be implemented by another class with the implements keyword. Interfaces provide a specification of the methods that it should be possible to invoke on the objects of a class. Abstract methods are methods that don't have a body. An abstract method is forced by the compiler to be overridden. Interfaces cannot be instantiated, so, abstract classes, they cannot be used to create objects; but objects of classes that either implement an interface or extend an abstract class can be instantiated. Interface methods do not have a body - the body is provided by the implementing classes. Interface methods are by default abstract and public. Interface attributes are by default public, static and final. An interface cannot contain a constructor (as it cannot be used to create objects). Interfaces can define static and default methods. Default methods are the only methods that have an implementation in the interface which is applicable to instances of all implementing types. Static and default methods are, by definition, not abstract. Interface types should be used to decouple a specification from its implementation if you plan to have different implementations of that specification as part of your design. Abstract classes can define abstract methods, and that methods of the abstract class can call their own abstract methods.

A class can implement multiple interfaces, but only extend one class at most.

If a class implements all the methods of an interface or abstract class correctly, then a subtype relationship between the implementing/extending class and the interface/abstract class is established.

Abstract classes permit to create functionality that subclasses can implement or override. Only abstract methods in the abstract class do not have any body.

An interface defines the functionality and cannot be implemented by its own. All the methods do not have a body. An interface cannot have instance variables and its methods can be of any scope (also private).

Classes cannot inherit from an interface, since an interface is by definition empty: it only dictates the mandatory implementation of certain members.

Why and when to use Interfaces?

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can implement multiple interfaces. Note: To implement multiple interfaces, separate them with a comma.

The implements keyword has two effects: first of all, it provides a formal guarantee that instances of the class type will have concrete implementations for all the methods in the interface type (enforced by compiler). Second it creates a subtype relationship between the implementing class and the interface type. Because the interface implementation relation defines a subtype relation, references to objects of concrete classes declared to implement an interface can be assigned to variables declared to be of the interface type. The subtype relationship between a concrete class and an interface is what enables *polymorphism.* Polymorphism is the ability to have different shapes.

Polymorphism provides two benefits in software design: *Loose coupling*, because the code using a set of methods is not tied to a specific implementation of these methods (classes are independent), and *extensibility*, because we can easily add new implementations of an interface (new "shapes" in the polymorphic relation).

Polymorphism means "many forms", and it can be achieved with inheritance, interfaces, composition or overloading.

Polymorphism helps make a design extensible by decoupling client code from the concrete implementation of a required functionality.

Polymorphism and inheritance: inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks.

An interface can't implement an interface nor a class, but only extend other interfaces (multiple inheritance is allowed with interfaces).

Code reuse: refers to the ability to reuse existing code for a new implementation rather than by writing new code from scratch.
With method implementation it is always possible to inspect the code, whereas with an interface specification reverse-engineering what the method does is not possible.

Using small interfaces encourages the respect of a software design principle called *separation of concerns.* The idea of this principle is a system should be divided into smaller and more manageable parts, and each of these parts should map a separate concern (or area of interest); meaning that they perform a specific set of tasks.

Use interfaces over abstract classes when I have the choice between the two.

Advantages of Interfaces over Abstract classes:

- An interface defines a set of methods that the implementing classes need to implement, but it does not provide an implementation for these methods. An abstract class can instead only be subclassed, and all of its methods must be required to the implementation defined in the abstract class, and this does not allow to provide different implementations for the methods in different subclasses, which you can instead to with interfaces.

- Interfaces support multiple implementation and abstract classes do not support multiple inheritance

- Interfaces are easier to extend: if we want to add or modify the methods defined in an interface, we can do it without breaking existing code. In an abstract class you may update the code in all of the subclasses to make sure it still works correctly.

Abstract classes are used not to have duplicate code (i.e., put methods to be used by multiple different classes in the abstract class), polymorphism is so enforced.

An abstract class is a class that is declared abstract and may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed. When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. An abstract class technically represents a correct but incomplete set of class member declarations. An abstract class cannot be used to create objects. To access the abstract class, it must be inherited from another class: the constructor of an abstract class can only be called from within the constructors of the subclasses, for this reason it makes sense to declare the constructors of abstract classes as protected.

Declaring a class as abstract has three main consequences:

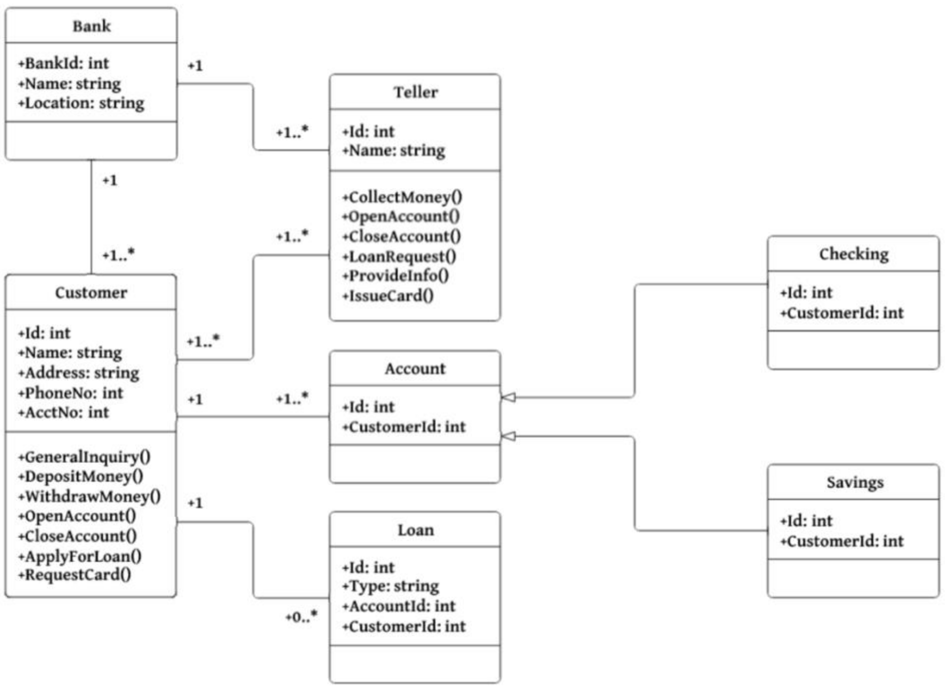
1) The class cannot be instantiated. This is a good thing because abstract classes should represent abstract concepts that it makes no sense to instantiate.
2) The class no longer needs to supply an implementation for all the methods in the interface(s) it declares to implement.
3) The class can declare new abstract methods (methods without a body which have to be implemented by the subclasses).

Abstract classes can define both abstract methods, which are methods that are not implemented and must be implemented by the extending subclasses, and concrete methods with have an implementation provided by the abstract class and the implementation for these methods is required to be the same in the subclasses.

Class diagrams: represent the static structure of the system in terms of classes and their relations. In a class diagram, each class is represented by a rectangle that contains the class name, as well as the names and types of its attributes and operations. Classes can be related to one another in various ways, such as inheritance (when one class is a subclass of another) or association (when one class uses or depends on another). These relationships are shown using lines and arrows.

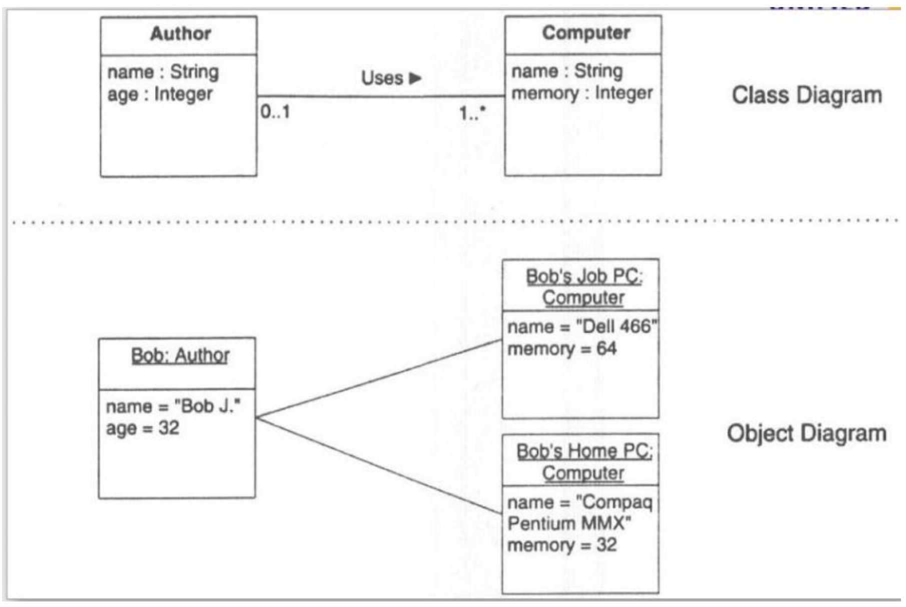
Class diagrams can also include interfaces (sets of related operations that a class can implement), associations (relationships between classes), and dependencies (relationships between classes in which one class uses or depends on another).

Example of a class diagram:



Class diagrams show generic descriptions of possible systems, and object diagrams show particular instantiations of systems and their behavior. Both of them model the static structure of a system, together with the behavior of individual classes or objects.

Danger: class diagrams risk turning into data models, so be sure to focus on behavior.



Advantages of Class Diagrams:

1) They are useful to represent how types are defined and related.
2) They are useful to capture the essence of one or more design decisions.

Disadvantages:

1) Poor tool for capturing any runtime property of the code.

Switch statement antipattern: occurs when there are too many switch statements in the code, which can make the code hard to test, hard to change and hard to understand. Switch-statements are not an antipattern per se, but if you're coding object oriented you should consider if the use of a switch is better solved with polymorphism instead of using a switch statement. A switch statement antipattern is not solved by replacing the switch statements with if else blocks, but instead by applying design patterns such as the Strategy pattern or the State pattern, or simply by applying polymorphism.

A function object is an object which behaves like a function. It is an object that provides the implementation for a single method. Their interface typically maps one-to-one to that of an interface.

Example:

```
@FunctionalInterface

public interface Function<T, R> {

  R apply(T t);

}
```

This interface has a single abstract method called apply, which takes an argument of type T and returns a value of type R.

To create a function object, you can implement the apply method in a class that implements the Function interface, or you can use a lambda expression to define an anonymous function that implements the apply method.

Example of a function object that adds 5 to a number:

```
Function<Integer, Integer> adder = (x) -> x + 5;

int result = adder.apply(10);  // result is 15
```

Iterator: it is an object that can be used to loop through variables, like ArrayList and HashSet.

In the UML, a stereotype is a variation on an element type, with the name of the variation placed in angle brackets. The Iterator design pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

One of the major benefits of interfaces and polymorphism is to promote flexible designs. The use of function objects such as comparators to customize the behavior of another part of the code (e.g. the sorting behavior) is recognized as one application of a more general idea called the *strategy* design pattern. It is useful to think of a part of the design of an application of Strategy when that part of the design is focused on allowing the switch between algorithms

Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application. Dependency Injection involves providing a class with its dependencies (with the objects passed as arguments) rather than by having the class create them itself. Dependency Injection makes our programming code loosely coupled. Loose coupling has one major disadvantage for cases where a client might be interested in more than one slice of behavior.

Example:

```java
public class EmailService {
    private EmailRepository emailRepository;

    // Constructor-based dependency injection
    public EmailService(EmailRepository emailRepository) {
        this.emailRepository = emailRepository;
    }
}
```

With interfaces client code is not coupled with the details of the implementation, and only depends on the methods it actually requires.

Functional Decomposition: decompose according to the functions a system is supposed to perform.

Object-Oriented Decomposition: Decompose according to the objects a system is supposed to manipulate. It is better for complex and evolving problems.

The result of the design process is not a final product, design decisions may be revisited, even after implementation. Design is not linear but iterative.

The design process is not algorithmic: a design method provides guidelines, not fixed rules.

Responsibility driven design: evenly distribute responsibilities among classes, such that there are no classes that have too many responsibilities. Responsibilities are meant to convey a sense of the purpose of an object and its place in the system. In practice responsibilities are the knowledge an object maintains and provides, and the actions it can perform. Responsibilities represent the public services an object may provide to clients.

To find classes: start with requirement specifications: What are the goals of the system being designed, its expected inputs, outputs and their responsibilities? Look for noun phrases.

Class-responsibility-collaboration (CRC) cards:

- Record the candidate Class Name and superclass (if known)
- Record each Responsibility and the Collaborating Classes
- CRC Cards are not a specification of a design; they are tools to explore possible designs

How to assign Responsibilities:

- Evenly distribute system intelligence: avoid centralization of responsibilities, have a clear control flow, but hard-wired system's behavior.
- State responsibilities are generally as possible: draw yourself vs draw a rectangle.
- Keep behavior together with any related information -> known as encapsulation
- Keep information about one thing in one place
- Share responsibilities among related objects

Collaborations: client requests to servers needed to fulfill responsibilities. Collaborations reveal control and information flow, and ultimately subsystems. Collaborations can uncover missing responsibilities. The analysis of communication patterns can reveal wrong assigned responsibilities.

Guidelines for good Responsibility Driven Design:

- You can talk only to your collaborators; if you need to talk to someone else, talk to a collaborator who can talk to the other person. If that turns out not to be possible, add a collaborator to your CRC cards.
- You may not ask the information you need to do something. Rather, you must ask the collaborator who has the information to do the work. It is okay to give your collaborator some bit of information that they need to do the work but keep this sort of passing to a minimum.
- If something needs to be done and nobody can do it, create a new class with its corresponding CRC card or add a responsibility to an existing class and its corresponding CRC card.
- If a CRC card gets too full, you must create another class and its corresponding CRC card to handle some of the responsibilities.
- Model what would happen if real people who were domain experts were solving the problem. Pretend computers didn't exist (getX is not very used in real life).
- The interactions of the objects represent the dynamic model of the program.
- The finished set of CRC cards is the static model of the program.

Protocol: set of signatures (i.e., an interface) to which a class will respond. Generally, protocols are defined for public responsibilities. Protocols for protected responsibilities should be specified if they will be used or implemented by subclasses. Therefore, construct protocols for each class, write a design specification for each class and subsystem, and write a design specification for each contract.

Chapter 4 – Object State

Static perspective: best represented by the source code or a class diagram.
Dynamic perspective: corresponds to the set of all values and references held by the variables in a program at different points in time.
Object state: refers to the particular pieces of information the object represents at a given moment. It is useful to distinguish between concrete state and abstract state.

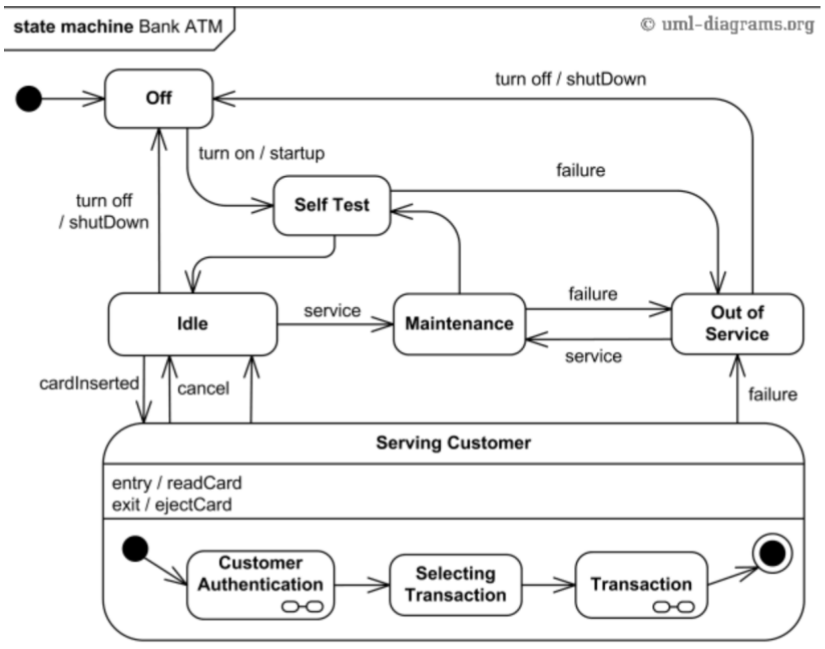
Concrete state of an object: collection of values stored in the object's fields. For example a Player object that stores his score.
Abstract state: arbitrarily defined subset of the concrete state space. Because abstract states are arbitrary partitions of the state space, they can be defined as anything.
Meaningful abstract state: abstract state that captures states that impact how an object would be used.
State space: set of all possible values and attributes of an object (includes all potential scenarios).
State: period of time during which an object is waiting for an event to occur.
It should be avoided to have states outside of the state space. State spaces should be kept to be as small as possible. The space inside the state space which is not the abstract space is called unwanted space, with design by contract we want to avoid entering this space. Input validation techniques (e.g. with enums) we can assure that the unwanted space is never entered.

A stateless object is an instance of a class without instance fields (instance variables).
A stateless object represents just one state for its whole life cycle.
A stateful object is an instance of a class that may morph itself into various states.

Absence of a transition (in state diagrams): means that the absent transition is not possible (i.e., invalid) for that state.

State Diagram: represents the dynamic behavior of a class in terms of (evolution of) its state. It shows the different states that the object or component can be in, as well as the transitions between those states, and so the condition of a system at different time instances. For example, a state diagram can be used to model the behavior of a light switch, which can be in either the "on" or "off" state and can be switched between these states by pressing the switch. In a state diagram, each state is represented by a rounded rectangle, and the transitions between states are represented by arrows. The conditions that trigger a transition are usually labeled on the arrow. State diagrams can also include other elements, such as activities (actions that are performed while in a particular state) and events (external stimuli that cause a transition to occur).

Example of a state diagram:



The state diagram model of an object can also be referred to as its life cycle, because it describes the “life” of an object, from its initialization to its abandonment. A good design principle to avoid objects with complex life cycles is to minimize the state of objects to what is absolutely necessary for the object to conduct its business. In practice, this means designing the class so that it is both impossible to put the object in an invalid or useless state, and that there is no unnecessary state information.

Speculative Generality antipattern: code is written with so much caution for possible future changes or requirements, that it becomes unnecessarily complex and harder to read. To avoid it, it is better to focus on the current needs of the system, rather than trying to anticipate possible future requirements or changes.

Temporary field antipattern: occurs when information is stored in a field (instance variable), but it does not uniquely contribute to the intrinsic value represented by the object; and is used to store data that is needed only for the duration of a single method. For example it occurs when we decide to use a field to store temporary data that we need during the process, but it makes the code hard to read as it is not understandable why this field has been created or what is represents. To avoid it we can use local variables to store this type of data.

Example:

```java
public class MyClass {
    private int temp;

    public int calculateResult(int a, int b) {
        temp = a + b;
        return temp * temp;
    }

    public void processInput(int input) {
        switch (input) {
            case 1:
                temp = input * 2;
                break;
            case 2:
                temp = input * 3;
                break;
            // ...
        }
    }
}
```

In this example it is not clear what the temp field corresponds to since it is used in two different methods. It is not clear whether this field is used correctly or what will be the consequences after modifying this code.

Long method antipattern: when a method has too many lines of codes in it, and is too complex and hard to understand (usually more than 10). This type of code is difficult to maintain, debug and test.

Object Identity: it refers to the fact that we are referring to a particular object, even if this object is not in a variable. The identity of an object usually refers to its memory location, or reference/pointer to.

Object equality: when two different objects of a class represent the same concept (e.g. ace of clubs). The Object class is the root of the inheritance hierarchy in Java, so all classes inherit its methods, including the equals method (but also hashCode, toString, clone, getClass, etc.).

Object uniqueness: objects in a class are unique if is not possible for two distinct objects to be equal. If the objects of a class can be guaranteed to be unique, then we no longer need to define equality, because in this case, equality becomes equivalent of identity and we can compare objects using the == operator (usually we cannot, to check the equality of two objects we need to overwrite the equals() and Hashmap() method, otherwise the equality of two objects with the same characteristics will evaluate to false).

The presence of the hashCode() function is necessary to respect the equality requirements.

The hashCode() method should return the same integer value for two objects that are equal according to the equals() method.

Example of an implementation of equals() and hashCode() methods for the Student class:

```java
public String id;
private String name;

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Student object2 = (Student) o;
    return((Student) otherStudent).id == this.id;

}

@Override
public int hashCode() {
  return Objects.hash(id);
}
```

If we change the hashCode() method to only consider the name field, then two Student objects with the same name but different id values will have the same hashCode, which violates the contract of the hashCode() method.

Serialization: involves converting an object into a data structure that can be stored outside a running program, and then reconstructed later. The reconstruction of a serialized object almost invariably leads to a copy of the serialized object being created.

The classic way to prevent instantiation is to make the class constructor private.

Hashode() returns an integer representing the current instance of the class. If we override equals(), we also have to override hashcode(). Generally, we override either both of them or neither of them.

Three criteria in the hashcode() contract:

a) Internal consistency: the value of hashcode() only changes if a property in equals() changes.
b) Equals consistency: objects that are equal to each other must return the same hashcode.
c) Collisions: unequal objects may have the same hashcode.

<u>Chapter 4 – Insights</u>

Be explicit about whether objects of a class should be unique or not. If objects are not designed to be unique, override the equals and hashCode methods; if objects should be unique, consider using the flyweight pattern to enforce uniqueness. Consider an explicit structure, such as singleton or dependency injection, for managing classes that should yield only one instance of a stateful object. Additional data can be attached to instances of inner classes, either in the form of a reference to an instance of an outer class, or as copies of local variables bundled in a closure.

Chapter 5 – Unit Testing

One way to detect bugs is by testing the code. The idea of unit testing is to test a small part of the code in isolation, so that if a test fails, it is easy to know where to look for problems. A unit test consists of one or more executions of a unit under test (UUT) with some input data and the comparison of the result of the execution against some oracle. A UUT is whatever piece of code we wish to test in isolation, they are often methods but, in some cases, they can also be entire classes, initialization statements, or certain paths through the code. The term oracle designates the correct or expected result of the execution of a UUT. The comparison of the result of executing the UUT with the oracle is called assertion: the name captures the idea that the role of the comparison is to assert that result is what we expect. Unit testing doesn’t help with source code indentation, and it is not about the interaction among different classes. Unit testing is a practice where we automatically execute code to check that it works as expected.

Exhaustive testing: process of testing absolutely everything just to make sure that the product cannot be destroyed and if fully functional and free of detects (it is almost impossible).

Benefits of unit testing: helps detecting bugs in new code, checks that a tested behavior that is used to meet some specific expectation still does meet that expectation after the code changes.

Disadvantages of unit testing: it cannot verify the code to be correct, when a test passes it only shows that the one specific execution of the code that is being tested behaves as expected.

Unit testing is normally done automatically through a unit testing framework. UTFs automate a lot of the mundane (trivial) aspects of unit testing, including collecting tests, running them, and reporting the results. Frameworks also include a set of constructs to allow developers to write tests in a structured way and (if they so choose) efficiently. Unit testing frameworks help automize the execution of tests.

The major constructs supported by testing frameworks are test cases, test suites, test fixtures and assertions. The dominant testing framework for Java is called JUnit. In Junit, a unit test maps to a method.

Test fixtures do not compare the observed behavior of the executed code against the expected one, that is the role of tests itself. Test fixtures are used to set up the conditions for the test to be run in a consistent and repeatable way.

The @Test annotation indicates that the annotated method should be run for a unit test. To constitute proper tests, test methods should contain at least one execution of a unit under test. The way to automatically verify that the execution of a unit under test has the expected effect is to execute various calls to assert methods. Assert methods are different than assert statements. Assert methods are static methods part of the org.junit.jupiter.api.Assertions class and all they do is verify a statement and if it is false, report a test failure. The Junit framework includes a graphical user interface component called test runner which automatically scans some input code, detects all the tests in the input, executes them, and then reports whether the tests passed or failed. If a test contains no asserts, then it cannot detect errors, tests only work if they contain asserts.

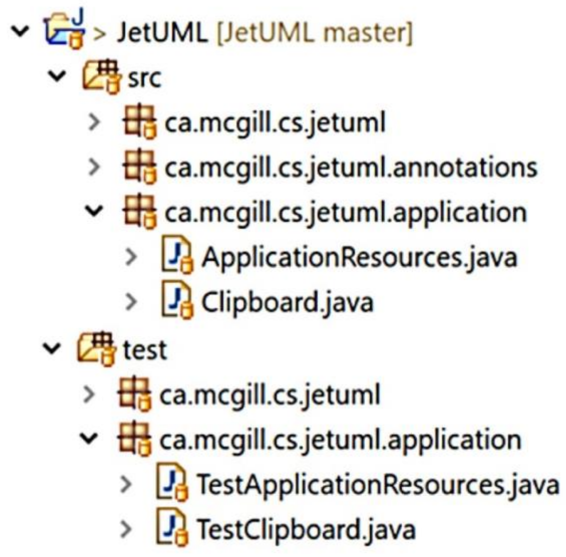
@BeforeEach is used to signal that the annotated method should be executed before each @Test method in the current test class.

Each test case is embedded into one test method.

A test class contains one or more test methods, and also methods to set up the state before and update the state after each test and before and after all the tests.

Final methods are not executed as unit tests by Junit.

A test suite is a collection of tests for a project. By default, it consists of all the unit tests for the production code in the project. However, for some reason, it may be desirable to run only a subset of different tests at different times. Junit provided a @Suite construct that allows a developer to list a number of test classes to be executed together. In Java a common idiom is to have one test class per project class, where the test class collects all the tests that tests methods or other usage scenarios that involve the class. Furthermore, it is also a common practice to locate all the testing code in a different source folder with a package structure that mirrors the package structure of the production code. Classes and methods in the test package can refer to non-public (but non-private) members of classes in the production code, while still being separated from the production code because since they have the same package name, they are in the same package scope as the production code.



```
> JetUML [JetUML master]
  src
    ca.mcgill.cs.jetuml
    ca.mcgill.cs.jetuml.annotations
    ca.mcgill.cs.jetuml.application
      ApplicationResources.java
      Clipboard.java
  test
    ca.mcgill.cs.jetuml
    ca.mcgill.cs.jetuml.application
      TestApplicationResources.java
      TestClipboard.java
```

Test oracle: uses the expected results (oracles) to decide whether a test passed or failed.

Unit tests should be:

a) Fast: they are intended to be run often, and in many cases within a programming-compilation-execution cycle. For this reason, whatever test suite is executed should be able to complete in the order of a few seconds.

b) Independent: each unit test should be able to execute in isolation. This means that one test should not depend on the fact that another test executes before to leave an input object in a certain state. It is often desirable to execute only a single test. Each test should start with a fresh initialization of the state used as part of the test.
c) Repeatable: the execution of unit tests should produce the same result in different environments. The test oracles should not depend on environment-specific properties such as display size, CPU speed or system fonts.
d) Focused: tests should exercise and verify a slice of code execution behavior that is as narrow as reasonably possible. A test that checks a single input on a single method call will make it easy to home in on a problem. Tests should ideally focus on testing only one aspect of one unit under test. If that UUT is a method, we can refer to it as the focal method for the test.
e) Readable: the structure and coding style of the test should make it easy to identify all the components of the test (unit under test, input data, oracle), as well as rationale for the test.

Duplicate Code antipattern: designates a part of code which is repeated multiple times in a program, rather than being isolated into a block which could be reused. This antipattern is achieved by practically copying and pasting code into multiple parts of the code rather than by encapsulating it and making it reusable. To avoid this antipattern, we could use the principle of separation of concerns.

In some cases, it makes sense to test private methods using metaprogramming, and in some cases, it makes sense to ignore private methods during testing.

How can we test private methods? Private methods are internal elements of other, accessible methods, and therefore are not really units that should be tested. The code in private methods should be tested indirectly through the execution of the accessible methods that call them. The private access modifier is a tool to help us structure the project code, and tests can ignore it. This second (and latter) option is the most reasonable. Situations where private method should probably not be tested separately are when their parameters or return types encode detailed information about the internal structure of the class or make narrow assumptions about the internal implementation of the class.

Any data used by unit tests must be initialized before every test, since tests are not guaranteed to be executed in any specific order.

For the vast majority of UUTs it is not physically possible to exhaustively test the input space. Clearly, we need to select some input out of all the possibilities to test. This is a problem known as test case selection, where test case can be considered to be a set of input values for an assumed UUT. The basic challenge of the test case selection problem is to test efficiently, meaning to find a minimal set of test cases that provides us a maximal amount of testing for our code.

Two basic approaches to the selection of test cases:

1) Functional (or black-box) testing: tries to cover as much of the specified behavior of a UUT as possible, based on some external specification of what the UUT should do. One of its advantages is that it is not necessary to access the code of the UUT. Another is that tests can reveal problems with the specification, and one is also that tests can reveal missing logic. Black box testing focuses on the inputs and the outputs of a method, without considering its

internal implementation. The testing method does not need to know how the methos is being implemented.

2) Structural (or white-box) testing: tries to cover as much of the implemented behavior of the UUT as possible, based on analysis of the source code of the UUT. The main advantage of white-box testing is that it can reveal problems caused by low-level implementation details that are invisible at the level of the specification. White box testing focuses on the internal implementation of a method.

One common method for determining what to test is based on the concept of coverage. A test coverage metric is a number (typically a percentage) that determines how much of the code executes when we run our tests. The simplest coverage metric is statement coverage. Statement coverage is the number of statements executed by a test or test suite, divided by the number of statements in the code of interest. The logic behind statement coverage is that if a fault is present in a statement that is never executed, the tests are not going to help find it. Although this logic may seem appealing, statement coverage is actually a poor coverage metric. A first reason for that is that it depends heavily on the detailed structure of the code, the second reason is that not all statements are created equally, and there can be quite a bit that goes on in a statement, especially if this statement involves a compound Boolean expression.

The second coverage metric is Branch Coverage. It is the number of program branches (decision points, outcomes of a condition) executed by the test divided by the total number of branches in the code of interest. Branch coverage is a stronger metric than statement coverage in the sense that for the same coverage result, more of the possible program executions will have been tested. Branch coverage is one of the most useful test coverage criteria: it is well supported by testing tools and relatively straightforward to interpret. Branch coverage subsumes Statement coverage, meaning that achieving complete branch overage always implies complete statement coverage.

The third coverage metric is Path Coverage. It is stronger than branch coverage and consists of a metric that calculates the number of executions paths actually executed over all possible execution paths in the code of interest. Path Coverage subsumes almost all other coverage metrics and is a very close approximation of the entire behavior that is possible to test. In many cases the number of paths through a piece of code is unbounded or troublesome (e.g. in loops) so it is not possible to compute this metric. For this reason, it is considered a theoretical metric.

Testing – wrong approach: reserve time for testing at the end of the development, develop more test cases, spend more money on testing.

Testing – right approach: do testing throughout the development process, write better test cases rather than more test cases, spend cost-effective money on testing (i.e., to improve testability).

Software failure: A component or a software system behaves in a way it is not expected.

Software defect (or bug or fault): a flaw in a component or in any part of the software that causes the system to behave incorrectly.

Error (or mistake): human action produces the incorrect result.

Software Validation: process of making sure that the system delivers what the user needs it to deliver: is the right system being built?

Software Verification: process of making sure that the system delivers what the specification wants it to deliver, and that is free of bugs: is the system built right?

Test Automation: it is the use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions.

Benefits of test automation reduces costs, human error, variance in test quality from different individuals and also reduces the cost of regression testing.

Regression testing: process of running tests to ensure that what was tested as correct still is.

Software Testability: it’s the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. It is dominated by two problems: how hard is it to provide the faults in the software? how hard is it to observe the results of test execution?

Controllability: denotes how easy it is to provide a program with the needed inputs, in terms of values, operations and behaviors.

Observability: denotes how easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components. A software that affects hardware devices, databases or remote files have low observability.

Chapter 6 – Composition

In object-oriented design, parts are connected through two main mechanisms: composition and inheritance. Unprincipled composition can lead to troubles.

Composition means essentially that one object is composed of other objects (e.g. a string object that contains char objects). The object that is composed of other objects is called aggregate, and the objects being aggregated are the elements. Composition is also helpful to break down a class that would otherwise be too big and complex. Unprincipled composition can lead to troubles.

Examples of composition are a human body with its organs or a deck of cards.

God class antipattern: when there is a class in the code (called God class) that is unmanageable, knows everything and does everything. This antipattern often occurs when a single class has too much functionality assigned to it and has too many responsibilities. God classes are a big problem because they violate practically every major principle of good software design.

Delegation: aggregate object delegates some services to the objects that serve a role of specialized service to the aggregate. This mechanism is also known as aggregation.

Composition: subclasses can’t exist by themselves.

Aggregation: subclasses can exist by themselves.

Composition is transitive: an object that is composed by other objects can, itself, be one component or delegate of another parent object.

We can specify the set of all possible implementations of an Interface statically (in the source code) rather than dynamically (when the code runs). There are three major limitations to this static structure:

1) The number of possible structures of interest can be very large, supporting all possible configurations leads to a combinatorial explosion of class definitions.
2) Each option requires a class definition, even if it is used very rarely. This clutters the code unnecessarily, because most implementations would probably look very similar.
3) In running code, it is very difficult to accommodate the situation where a type of an interface configuration is needed that was not anticipated before launching the application.

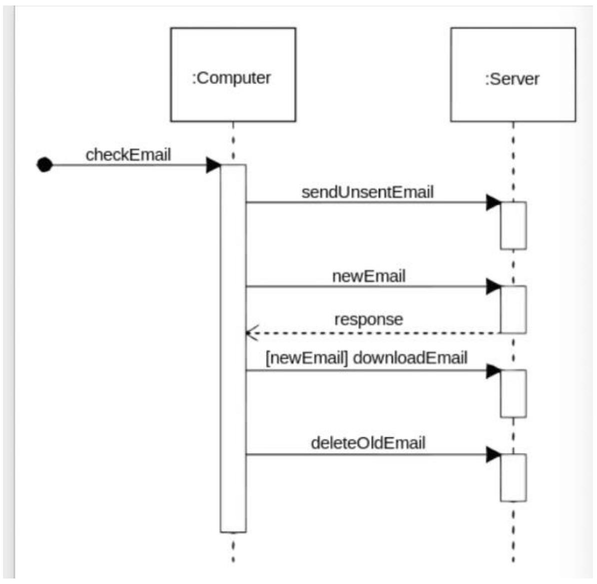
A general solution to these limitations is to support an open-ended number of configurations by relying on object composition as opposed to class definition.

Sequence diagrams show the interactions among a set of objects in temporal order. They model the dynamic behavior of a software system and represent a specific execution of the code. Sequence diagrams are a subtype of behavioral diagrams. In the sequence diagram, the object names are underlined and follow the convention name:type. An object in a sequence diagram is also referred to as implicit parameter, because it is the object upon which a method is called. Each rectangle at the top of the sequence diagram represents an object. Events or message dispatches are shown as horizontal arrows from the sender to the receiver.

Objects are represented by vertical bars and the interactions between objects are represented y arrows. These interactions show the order in which the interactions occur and include the name of the message and its return type (if there is any).

While sequence diagrams can show the interactions between objects or components at different points in time, they do not explicitly represent the condition of the system at those points. Instead, they focus on the interactions between objects and the flow of control between them.

Example of a sequence diagram:



A covariant return type means that the return type of an implementing method can be more specific than the return type of the corresponding interface method it implements.

The context of the Prototype design pattern is the need to create objects whose type may not be known at compile time. Prototype lets you copy existing objects without making your code dependent on their classes.

Message chain antipattern: occurs when the client requests another object, that object requests yet another one, and so on. A series of objects pass a request a long a chain of objects, rather than directly to the object that is responsible to handle the request, and each object forwards the request to the next object in the chain.

Example where the message chain antipattern is violated:

```java
public class Bank {
    private Branch branch;

    public Bank(Branch branch) {
        this.branch = branch;
    }

    public void processTransaction(Transaction transaction) {
        branch.processTransaction(transaction);
    }
}

public class Branch {
    private Manager manager;

    public Branch(Manager manager) {
        this.manager = manager;
    }

    public void processTransaction(Transaction transaction) {
        manager.processTransaction(transaction);
    }
}

public class Manager {
    private Employee employee;

    public Manager(Employee employee) {
        this.employee = employee;
    }

    public void processTransaction(Transaction transaction) {
        employee.processTransaction(transaction);
    }
}

public class Employee {
    public void processTransaction(Transaction transaction) {
        // code to process the transaction
    }
}
```

The Law of Demeter ("don't talk to strangers" principle) is a design guideline intended to help avoid the consequences message chain. This law states that the code of a method should only access:

a) The instance variables of its implicit parameter.
b) The arguments passed to the method.
c) Any new object created within the method.
d) (If needed) globally available objects.

This law states that an object should only communicate with its neighbors, so that the knowledge that the object has about the rest of the program is limited.

Chapter 7 – Inheritance

Inheritance is a powerful feature that offers a natural solution to many design problems related to code reuse and extensibility.

Inheritance allows to define some classes in terms of other classes and helps avoiding the Duplicated Code antipattern. The idea of inheritance is to initialize a new class (a subclass) that inherits some information from the base class (also called superclass). Inheritance avoids repeating declarations of class members because the declarations of the base class will automatically be taken into account when creating instances of the subclass. In Java, the subclass-superclass relation is declared using the extends keyword.

Inheritance should only be used for extending the behavior of a superclass. As such, it is bad design to use inheritance to restrict the behavior of the superclass, or to use inheritance when the subclass is not a proper subtype of the superclass.

The run-time type of an object refers to the time at which the object is used or accessed during the execution of the code. The run-time type of an object never changes for the duration of the object's lifetime. Example: s1.getName() or String name = s1.getName().

The compile-time (or static) type of an object refers to the time at which an object is created and initialized. In a correct program the static type of an object can correspond to its run-time type (or to any of its supertypes). The static type of an object can be different at different points in the code, depending on the variables in which an object is stored. Example: Student s1 = new Student();

In Java, once an object is created, its compile time remains unchanged. The runtime type of an object can change depending on when it is accessed or modified in the program. If an object is called multiple times, the runtime of the object will be the time of the last call to it.

Downcasting: usage of a cast operation to enable unsafe type conversion operations.

Explicit downcasting: involves explicitly specifying the target type using a typecast operator.

Example:

SuperClass superObj = new SubClass();

SubClass subObj = (SubClass) superObj;

Implicit downcasting: occurs when you assign a reference variable of a superclass or interface type to a subclass type, without specifying the type in the parentheses. Example:

SuperClass superObj = new SubClass();

SubClass subObj = superObj; //compile-error

Implicit downcasting is generally considered to be less error-prone than explicit downcasting, as it eliminates the need to use the typecast operator and eliminates the risk of runtime errors caused by invalid typecasts. To handle downcast, the best way is to check the object type before the downcast, using instanceof operator.

The instanceof operator tells the runtime type of an object and is used to test whether the object is an instance of the specified type (class or subclass or interface).

A subclass is instance of itself but also of the class it instantiates (superclass).

Field declarations define the structure of information stored by the instantiation object. When declaring a new object, this object will have a field for each field declaration in the class names in the *new* operation, and each of its superclasses, transitively. It does not matter if the fields are private, because accessibility is a static concept, meaning that it is only relevant for the source code (doesn't change anything if code cannot access the field declared in superclass, this does not change anything about the fact that this field is part of the object).

To refer to the specific implementation of a method located in the superclass from within a subclass the keyword super followed by the method call is used (e.g. super.draw()). This mechanism is referred to as a super call. Its effect is to statically bind the method call to the first overridden implementation of the method found by going up the class hierarchy. The implementation does not need to be in the immediate superclass, but there needs to be at least one inherited method.

Fields of an object are initialized top down, from the field declaration of the most general superclass down to the most specific class (the one named with the *new* operation). The order of constructor calls is bottom up (the first instruction of any constructor is to call a constructor, generally of its superclass).

If no constructor is declared for a class, a default constructor with no parameter is invisibly made available to client code. Declaring any non-default constructor in a class disables the automatic generation of a default constructor.

Inheriting methods is different from inheriting fields (attributes) because method declarations do not store information that represents the state of an object, and so do not require any initialization. An instance (non-static) method is just a different way to express a function that takes an object of its declaring class as an argument. Methods of a superclass are automatically applicable to instances of a subclass because instances of a subclass can be assigned to a variable of any supertype. If we want, we can redefine (override) the behavior of an inherited method by supplying an implementation in the subclass that only applies to instances of the subclass.

Overriding methods allows programmers to declare different versions of the same method, so that the most appropriate method will be selected based on the run-time type of the implicit parameter. The selection procedure is often referred to as dynamic dispatch, or dynamic binding.

It is not possible to override private methods. It is not possible to override the Constructor. We cannot change the scope (visibility) of an overridden method (e.g. from public to protected) if accessibility of the method decreases, otherwise we can. It is allowed to have multiple constructors in Java, but their arguments can't be the same. A subclass can override methods from the superclass, but it must not implement the same methods as the superclass, also new methods can be added (the methods in the superclass must be implemented in the subclass too). In Java, Classes are organized into a single-rooted class hierarchy. If a class does not declare to extend any class, by default it extends the library class Object. Class Object constitutes the root of any class hierarchy in Java code.

If a class has private fields, every subclass of this class also has these fields, but the subclass cannot access them. In Java, the fields of an object are initialized top down, from the field declarations of the most general superclass down to the most specific class (the one named in the new statement).

Possible object instantiation: Person is superclass with method eat, walk and Student extends Person with methods eat and study. Person p = new Student(); p.walk(); is allowed. Student s = new Student(); s.walk(); is allowed. Person p = new Student(); p.study(); is not allowed. Student s = new Person(); is not allowed.

Java and many other programming languages support another mechanism for specifying different implementations of the same method, this time by selecting the method based on the types of the explicit parameters. This mechanism is known as overloading. The selection of a specific overloaded method or constructor is based on the number and static types of the explicit arguments. When the method signature (name and parameters) is the same in the superclass and the child class, it's called overriding. When two or more methods in the same class have the same name but different parameters, it's called overloading. Overloading improves the readability of the program by allowing the same name for the methods that execute the same operation, but if it is not used clearly and in an efficient way, Overloading can lead to code being hard to understand. This is the case when the types of the parameters of overloaded versions of a method or constructor are related within a type hierarchy. In general, avoid overloading except for constructor overloading or library methods that support different primitive types.

The clone() method of Object class is used to clone an object. Java's cloning mechanism should only be used to support polymorphic copying with inheritance when no better alternative is available. To clone an object, it is necessary to override Object#clone() as a public method, and make a super call to clone() from within the method. Example:

Not to do: Deck clone = new Deck();

To-do: Deck clone = (Deck) super.clone();

Use polymorphic copying to make copies of objects whose concrete type is not known at compile time.

Composition vs Inheritance: Composition-based reuse generally provides more run-time flexibility. Composition should therefore be favored in design contexts that require many possible configurations, or the opportunity to change configurations at run time. At the same time, composition-based solutions provide fewer options for detailed access to the internal state of a well-encapsulated object. Inheritance-based reuse solutions tend to be better in design contexts that require a lot of compile-time configuration, because a class hierarchy can easily be designed to provide privileged access to the internal structure of a class to subclasses (as opposed to aggregate and other client classes). With composition we store several implementations, each of which can have unique behaviors depending on the context of when they are called. Composition is necessary when one item's properties have to be included into another object in order for the relationship to make sense. In order to create a new class that is composed of pre-existing classes, an object from each of the existing classes must be specified as a member of the new class. Composition is beneficial because it provides a better way to use an object without violating the internal information of the object. Inheritance instead is a type of capability that allows one thing to gain the properties of one or more other objects through the use of another object (is-a relationship). In Inheritance, a subclass should be a natural subtype of the base class that extends the behavior of the base class. Inheritance is used to achieve type matching, and not to get behavior. We acquire new behavior not by inheriting it from a superclass, but by composing objects together. Inheritance can be efficient but may also lead to tight coupling between classes, where changes to the superclass can have unintended consequences on the subclasses. Composition instead might be preferred as allows for more flexibility and makes code easier to change, as objects can be composed and reused in different ways without affecting one another.

Template method design pattern: put all the common code in the superclass, and define some hooks to allow subclasses to provide specialized functionality where needed. We define an algorithm at the superclass level (the common method in the superclass is a template, that gets instantiated differently for each subclass). Within the algorithm, call an abstract method to perform operations that can’t be generalized in the superclass. This way you can change the behavior of an algorithm without changing its structure. The steps in the algorithm are defined as non-private methods in the superclass.

Final methods cannot be overridden by subclasses. The main purpose for declaring a method to be final is to clarify out intent that a method is not meant to be overridden. One important reason for preventing overriding is to ensure that a given constraint is respected. Final methods are exactly what is needed for the Template Method, because we want to ensure that the template is respected for all subclasses. The final keyword can also be used with classes: classes declared to be final cannot be inherited. Generally, stating that a class cannot be inherited tends to make a design more robust because it prevents unanticipated effects caused by inheritance. (Declaring classes as final also has benefits in terms of run-time performance).

The intuition that inheritance should only be used for extension is captured by the Liskov Substitution Principle (LSP). It states that subclasses should not restrict what clients of the superclass can do with an instance. Methods of the subclass:

a) Cannot have stricter preconditions
b) Cannot have less strict postconditions
c) Cannot take more specific types as parameters
d) Cannot make the method less accessible (e.g. from public to protected)
e) Cannot throw more checked exceptions
f) Cannot have a less specific return type

Summary of the Template Method Pattern

The method with the common algorithm in the abstract superclass is the template method; it calls the concrete and abstract step methods. If, in a given context, it is important that the algorithm embodied by the template method be fixed, it could be a good idea to declare the template method final, so it cannot be overridden (and thus changed) in subclasses. It is important that the abstract step method has a different signature from the template method for this design to work. Otherwise, the template method would recursively call itself, quite possibly leading to a stack overflow; following the advice of Section 7.5 about avoiding unnecessary overloading, I would recommend actually using a different name in all cases. The most likely access modifier for the abstract step methods is protected, because in general there will likely not be any reason for client code to call individual steps that are intended to be internal parts of a complete algorithm. Client code would normally be calling the template method. The steps that need to be customized by subclasses do not necessarily need to be abstract. In some cases, it will make sense to have a reasonable default behavior that could be implemented in the superclass. In this case it might not be necessary to make the superclass abstract. In our example, there is a default implementation of log() that can be overridden by subclasses. In a different context, it might make more sense to declare this method abstract as well.

Inheritance accomplishes two things:

• It reuses the class member declarations of the base class as part of the definition of the subclass;

• In introduces a subtype–supertype relation between the subclass and the superclass.

To use inheritance properly, it must make sense for the subclass to need both of these features. A common abuse of inheritance is to employ it only for reuse and overlook the fact that subtyping does not make sense: Inheritance is appropriate only in circumstances where the subclass really is a subtype of the superclass. In other words, a class B should extend a class A only if a meaningful “is-a” relationship exists between the two classes.

Chapter 7 – Insights

Even in the presence of inheritance, consider keeping your field declarations private to the extent possible, as this ensures tighter encapsulation. Subclasses should be designed to complement, or specialize, the functionality provided by a base class, as opposed to redefining completely different behavior. Because it can easily lead to code that is difficult to understand, keep overloading to a minimum. Java’s cloning mechanism should be used to implement polymorphic copying when the fields of the superclass are not accessible. However, cloning is a complex and error-prone mechanism that must be used very carefully. Ensure that any inheritance-based design respects the Liskov Substitution Principle. In particular, do not use inheritance to restrict the features of the base class. Consider using the template method pattern in cases where an algorithm applies to all subclasses of a certain base class, except for some steps of the algorithm that must vary from subclass to subclass. If there is no scenario for overriding a method, consider declaring it final. Similarly, if there is no specific reason for a class to be extensible using inheritance, consider declaring it final.

Chapter 8 – Inversion of Control

Inversion of control involves reversing the usual flow of control from caller code to called code to achieve separation of concerns and loose coupling. One of the main realizations of the principle takes the form of the Observer pattern.

Pairwise dependencies antipattern: whenever the user changes a class, the other classes also have to be changed since the classes have dependencies on each other (e.g., in a class I call methods of another class on an object of that other class).

Designs with pairwise dependencies have two limitations:

a) High Coupling: each panel depends on many other panels.
b) Low extensibility: to add or remove a panel it is necessary to modify all other panels.

One way to get rid of pairwise dependencies to synchronize multiple representations of the same data is to separate abstractions responsible for storing data from abstractions responsible for viewing data and from abstractions responsible for changing data. This idea is known as Model-View-Controller model: the Model is the abstraction that keeps unique copy of the data of interest. The View is the abstraction that represents one view of the data. The Controller is the abstraction of the functionality necessary to change the data stored in the model.

The central idea of the Observer pattern is to store data of interest in a specialized object, and allow other objects to observe this data. The object that stores the data of interest is called subject, model or observable and it corresponds to the Model in the Model-View-Controller decomposition.

In the Observer we can include a helper method called notification method. If it is public, clients with references to the model get to control when notifications are issued. If private, it is assumed that the model is called at appropriate places in the state-changing methods of the model.

We talk about inversion of control because, to find out information from the model, the observers do not call a method on the model, they instead wait for the model to call them back. The method called by the model on the observer is called a callback. A callback is not to tell observers what to do, but rather to inform them about some change in the model and let them deal with it. In the Observer pattern, an observable object notifies its observer objects by calling their callback method(s).

To ensure that the model notifies observers whenever a state change occurs, two strategies are possible:

1) A call to the notification method must be inserted in every state-changing method, in this case the notification method can be declared as private. Downside: notifying observers with every state change may lead to some performance problem.
2) Clear documentation has to be provided to direct users of the model class to call the notification method whenever the model should inform observers. In this case the notification method needs to be non-private.

One way to think about callback methods is as events, with the model being the event source and the observers being the event handlers. In cases where an observer does not need to react to an event, the unused callbacks can be implemented as empty (do-nothing) methods. If the reliance on empty method occurs too often, it is possible to implements these methods in a class and inherit from it instead.

The code that implements the Graphical User Interface (GUI) portion of an application makes heavily use of the Observer. JavaFX is an extensive GUI for the Java framework. Th code that makes up a GUI framework is split into two parts:

1) The framework code: consists of a component library and an application skeleton. The component library is a collection of reusable types and interfaces that implement typical GUI functionality: buttons, windows, etc. The application skeleton is a GUI application that takes care of all the inevitable low-level aspects of GUI applications, and in particular monitoring events triggered by input devices and displaying objects on the screen. By itself, the application skeleton does not do anything visible: it must be extended and customized with application code
2) Application code: code written by GUI developers to extend and customize the application skeleton so that it provides the required user interface functionality.

With GUI frameworks, the application must be started by launching the framework using a special library method. The framework then starts an event loop that continually monitors the system for input from user interface devices. Throughout the execution of the GUI application, the framework remains in control of calling the application code. The application code gets executed only at specific points, in response to calls by the framework. This process is thus a clear example of inversion of control. Application code does not tell the framework what to do: it waits for the framework to call it.

Once the framework is launched and displaying the desired component graph, its event loop will automatically map low-level system events to specific interactions with components in the graph (for example, placing the mouse over a text box, or clicking a button). Such interactions are called events; unless specific application code is provided to react to an event, nothing will happen as a result of the framework detecting this event.

The application code for a GUI application can be split into two categories:

a) The component graph: it's the actual user interface and is comprised of a number of objects that represent both visible (e.g., buttons) and invisible (e.g., regions) elements of the application. These objects are organized as a tree, with the root of the tree being the main window or area of the GUI. The component graph is the collections of objects that forms what we usually think of as the user interface: windows, textboxes, buttons, etc.
b) The event handling code

To build interactive GUI applications, it is necessary to handle events like button clicks and other user interactions. Event handling in GUI frameworks is an application of the Observer pattern, where the model is a GUI component (such as a button). It can be useful to distinguish the GUI in three different perspectives: user experience, source code, and run time.

User experience perspective: corresponds to what the user experiences when interacting with the component graph. Because not every object in the component graph is necessary visible, it is important to remember that the user experience perspective does not show the complete picture of the application.

Source code perspective: shows the kind of information about the component graph that is readily available from the declarations of the classes of the objects that form the component graph. This information is best summarized by a class diagram.

Run-time perspective: it's the instantiated component graph for a graphical interface. This perspective can best be represented as an object diagram.

The local variable that holds a reference to the invisible component is named root, to indicate that it is the root of the component graph.

In GUI frameworks, objects in the component graph act as models in the Observer. Once the framework is launched, it continually goes through a loop that monitors input events and checks whether they map to events that can be observed by the application code.

Process of handling the action event on a text field:

a) Define a handler for the event: define a class that is a subtype of EventHandler<ActionEvent>, this class will be the event handler class.
b) Instantiate a handler: create an instance of the class defined in the previous step. The instance is the event handler instance, also called event handler or even just handler.
c) Register the handler: call the registration method on the model and pass the handler as an argument.

There are two main strategies to decide where to place the definition of the handling code:

a) Define the handler as a function object using an anonymous class or a lambda expression. This is a good choice if the code of the handler is simple and does not require storing data that is specific to the handler.
b) Delegate the handling to an element of the component graph by declaring to implement the observer interface. This is a good choice if the code handler is more complex or requires knowing about many different aspects of the internal structure of the target component.

The context for applying the visitor pattern is when we want to make it possible to support open-ended number of operations that can be applied to an object graph, but without impacting the interface of the objects in the graph. The visitor provides a solution by supporting a mechanism whereby it is possible to define an operation of interest in a separate class and inject it into the class hierarchy that needs to support it. The visitor pattern has two core ideas:

1) Enable the integration of an open-ended set of operations that
2) Can be applied by traversing an object graph, often a recursive one.

There are two main ways to implement the traversal of the object graph in the Visitor: one option is to place the traversal code in the accept method of aggregate types. The other option is to place this code in the visit methods that serve as callbacks for aggregate types.

The main advantage of placing the traversal code in the accept method is that it can help achieve stronger encapsulation because the internal structures can be accessed without being part of the class’s interface. The main disadvantage of it is that the traversal order is fixed in the sense that it cannot be adapted by different visitors.

If the traversal code is implemented in accept, concrete visitors cannot change it. If encapsulation of target elements is more important, it is better to place the traversal code in the accept method. If the ability to change the traversal order is more important, then it is better to place the traversal code in the visit method. Because the traversal code is implemented within the class of the aggregate, it can refer to the private field that stores the aggregation.

In an UML diagram, a black diamond represents composition, whereas a white diamond represents aggregation. Aggregation (“consists of”) implies a relationship where the child can exist independently of the parent, whereas composition (“contains) implies a relationship where the child cannot exist independent of the parent (e.g. room and house, cards and deck, organs and body).

The cornerstone of the Visitor pattern is an interface that describes objects that can visit objects of all classes of interest in an object graph. This interface is called abstract visitor. A concrete visitor is an implementation of this interface. The concrete visitor provides a way to organize code in terms of functionality as opposed to data. One of the benefits of the Visitor is to allow a different style of assignment of responsibilities to classes, and thus a separation of concerns along a different criterion (functionality-centric vs data-centric). Although the concrete visitor separates a well-defined operation into its own class, it still needs to be integrated with the class hierarchy that defines the object graph on which the operation will be applied. This integration is accomplished by a method, usually called accept, that acts as a gateway into the object graph for visitor objects. The accept method takes a single argument.

Any object graph with more than one element has an aggregate node to its root.

Factory method: method whose primary role is to create and return an object.

Chapter 8 – Insights

With an application of the observer, the model class can be used without any observer, and generally does not depend on the specific types of any observer that observes it. In the observer, the model aggregates a number of abstract observers. The abstract observer is an interface that is implemented by the concrete observers. The abstract observer interface should define one or more callback methods that map to state-changing events in the model. The model needs to notify observers when it changes its state, but when to issue that notification is a design decision. There are two strategies for exchanging data between a model and its observers: push or pull. These strategies can be combined. Callback methods can be thought of as events to support a type of event-based programming. In this case, models are the event source and observers are the event handlers. An abstract observer can define multiple callbacks. Abstract observer interfaces can also be split up in smaller observer interfaces to afford more flexibility in defining how observers can respond to events. If it is often the case that observers implement callbacks by doing nothing, consider using adapter classes or default methods.

Chapter 9 – Functional Design

The main idea behind Functional Design is being able to pass functions as input to other functions.

An anonymous function is a function without a name. Anonymous functions are often arguments to other functions.

Higher-order functions: functions that take other functions as argument.

An application is designed in the functional programming paradigm when they can be built from the principled use of higher-order functions. Using higher-order functions does not by itself mean than an application's entire design becomes functional.

First-class functions: functions that are treated as value and can be passed as argument, stored in variables, or stored by other functions. First-class functions behave like variables.

First-class functions make it possible to define small pieces of behavior, such as to filter or compare objects. First-class functions also allow us to specify some processing behavior and to defer its execution to the point where it is required.

There are three mechanism that enable first-class functions: functional interfaces, lambda expressions and method references.

Functional interface: type of interface that declares a single abstract method and is designed to be used with lambda expressions and method references (with :: operator). Functional interfaces define a function type. It can have any number of static methods. In a class that implements the functional interface I can write Interfacename p=() -> {System.out.println("Hello)}

Lambda expression: compact form of expression of functional behavior. They are basically anonymous functions. The function parameter is declared on the left of the arrow (->), and the expression on the right of the arrow represents the body of the function. We do not care about the actual name of the function because it will get polymorphically through the name of the method in the functional interface. Information about the behavior of the lambda expression is typically the code of the lambda expression, or at best an informative variable name. Lambda expressions consists of three parts: a list of parameters, a right arrow, and a body. When the lambda expression requires no parameter, we provide an empty set of parameters () ->. The body of lambda expressions can take one of these two forms:

a) a single expression (e.g. a == 1)
b) a block of one ore more statements (e.g. {return a == 1;})

Normally, when writing lambda expression, we define them as expressions whenever possible and, when the computation is complex and we need multiple statements, fall back on defining them as a block.

Essentially, lambda expressions are used to instantiate functional interfaces, just like anonymous classes. The single method implemented through the a lambda expression is called like any other method.

When the Java compiler sees a lambda expression, it tries to match it to a functional interface.

The compiler looks that the type of the variable is a functional interface, then it looks the parameter that the types of the lambda expression are compatible with those of the functional interface. In the end it checks that the type of the value returned by the body of the lambda expression is compatible with that of the abstract method of the functional interface.

Lambda expressions provide better code readability and work efficiently with Immutables in functional programming.

Method references: indicated with P::m where m refers to the name of the method of interest and P is a prefix that can take any different forms. This consists of a reference to an instance method of an arbitrary object of a particular type. It is also possible to supply a reference to an instance method of a particular object using the notation o::m, where o is an expression that evaluates to a reference to an object and m is the method. Method references do not have to match their corresponding functional interface exactly: a method reference only needs to be compatible with its required assignment, invocation, or casting context. Method references support using both static and instance methods as first-class functions, and the mapping between the reference and the interface method is based on the parameter and return types. It is also possible to use method references to refer to constructors and array initializers. In the case of the instance method of a given type, the parameter is the implicit parameter of the method; in the case of the static method, the parameter is the explicit parameter of the method. In the case of the instance method of a particular object, the parameter is also the explicit parameter of the method, whose implicit object is specified in the method reference.

Example of lambda expressions and method reference:

```
// Sort the list using a lambda expression
    words.sort((s1, s2) -> s1.compareTo(s2));

// Sort the list using a method reference
    words.sort(String::compareTo);
```

Types of deferred processing: supplying an object consuming an object, and mapping an object to an other object. Supplier and consumer support one-way deferred data flow, but there can also be situations where we need to both consume and supply a value. The generic function type that captures this requirement is T -> R; this function maps an object of type T into an object of type R.

Instead of creating objects of different classes and enabling polymorphism through a common supertype, we can define families of functions whose type is compatible and invoke them interchangeably. In the simple cases where strategies are stateless and their interface boils down to a single method, abstract strategy can be expressed as a functional interface.

In context where we can define an abstract observer with a single callback, we can use functional-style design to create a compact application of the pattern. One scenario where functional-style design shines is to structure code responsible for processing a collection of data.

A design pattern functional-style strategy is used in cases when the strategy is stateless and their interface depends on a single method.

Stream: sequence of data elements, a bit like a collection. A collection represents a store of data whereas a stream represents a flow of data. Elements in a collection have to exist before they are added to the collection, but elements in a stream can be computed on-demand. Although collections can only store a finite number of elements, streams can technically be infinite; for example, although it is not possible to define a list that contains all even numbers, it is possible to create a stream that produces this data. Collections can be traversed multiple times, but the traversal code is located outside of the collection, for example in a for loop or iterator class. In contrast, streams can only be traversed once: their elements are consumed as part of the traversal. The traversal code is hidden within the higher-order functions provided by the stream's interface.

Streams can be used to transform all data elements in a collection to a derived value, by applying a function to all of them.

obj.stream().map(x) applies a function x to every data element of an object obj. The .map(x) function can be used with lambda functions. Mapping functions is also very common in other programming languages.

The main way that streams support functional-style programming is that they define a number of higher-order functions. A basic higher-order function for streams is forEach, which applies a consumer function to all elements of the stream. Stream functions that do not produce a stream of results that can be further processed as part of a pipeline are called terminal operations.

The map function takes as input an object of type T and returns an object of type R. This means that the map function transforms a stream of objects into another stream where every object is obtained by applying a function to an object in the first stream. A flat map operation maps each input object to a stream, but merges the resulting streams into a single one instead of collecting the streams as individual elements of another stream.

When working with streams of data, a common scenario is that we want to not only process each data element, but also do something with the data as a whole. Typically, this means either aggregating the effect of the operations into a single result. Terminal operations such as count() and sum() are good examples of data aggregation in that sense;
or collecting the individual results of the operations into a stored data structure. In
Java this would typically mean a List or other collection type.
Although they seem different, these alternatives actually have in common that, conceptually, they represent reducing a stream to a single entity. In the second case, the entity may be a collection of many elements, but conceptually it is nevertheless a single, stored structure as opposed to a stream.

Chapter 9 – Insights

Favor short lambda expressions where the body is also an expression (as opposed to a sequence of statements). To emphasize flexibility and extensibility in your design, use library functional interfaces to define function types; to emphasize design constraints and intent, use application-defined functional interfaces. Compose functions using functions, as opposed to imperative statements. Use the helper methods of library types to compose functionality in intuitive ways. Consider using supplier, consumer, and mapping function types to parameterize behavior. Consider defining strategies in the strategy pattern as functional interfaces. Consider using first-class functions to define abstract observers in the observer pattern. Structure data-processing code so that it is more declarative than imperative in style. Use the mapping abstract operation to convert data elements into the values that are directly used by a computation. Use collector objects to accumulate the result of stream operations.