Pattern Book

Desiree Trepanier

[Abstract Factory 2](#_Toc449312338)

[Adapter 3](#_Toc449312339)

[Bridge 3](#_Toc449312340)

[Builder 4](#_Toc449312341)

[Chain of Responsibility 5](#_Toc449312342)

[Command 6](#_Toc449312343)

[Composite 7](#_Toc449312344)

[Decorator 8](#_Toc449312345)

[Double Dispatch 9](#_Toc449312346)

[Façade 10](#_Toc449312347)

[Factory Method 11](#_Toc449312348)

[Null Object 11](#_Toc449312349)

[Observer 12](#_Toc449312350)

[Proxy 13](#_Toc449312351)

[Prototype 13](#_Toc449312352)

[Singleton 14](#_Toc449312353)

[State 14](#_Toc449312354)

[Strategy 15](#_Toc449312355)

[Template Method 15](#_Toc449312356)

[Visitor 16](#_Toc449312357)

[Unlisted Patterns 17](#_Toc449312358)

# Abstract Factory

|  |  |
| --- | --- |
| Issue | Need to create a family of related objects without specifically specifying their classes. |
| Solution | Define the objects for Factory and Product(s). The Factory is used to create the products and the Product defines the object being made. This will return products based on the factory and allow factories to be swapped out with other factories that extend the same base class. |
| Related | **Builder** and Abstract Factoryare very similar, except that Builder offers more control over the construction process.  **Singleton**, **Factory Method**, and **Prototype** can be used in Abstract Factory’s implementation. |
| Code | public abstract class AbstractProductA {  public abstract void doThisA();  public abstract void doThatA();  }  public abstract class AbstractProductB {  public abstract void doThisB();  public abstract void doThatB();  }  public class ProductA1 extends AbstractProductA {  public void doThisA() {  // do something  }  ...  }  public class ProductA2 extends AbstractProductA {  public void doThisA() {  // do something different  }  ...  }  public class ProductB1 extends AbstractProductB {  public void doThisB() {  // do something else different  }  ...  }  ...  public abstract class AbstractFactory {  public abstract AbstractProductA createProductA();  public abstract AbstractProductB createProductB();  }  public class Factory1 extends AbstractFactory {  public AbstractProductA createProductA() {  return new ProductA1();  }  ...  }  public class Factory2 extends AbstractFactory {  public AbstractProductA createProductA() {  return new ProductA2();  }  ...  } |

# Adapter

|  |  |
| --- | --- |
| Issue | Need to translate one interface into another for them to work together. |
| Solution | One interface will be the Target object and the other will be the Adaptee object. Define an Adapter object that will handle translating the Adaptee’s interface to the Target’s. |
| Related | Adapter is a type of **Wrapper**.  **Bridge** and Adapter are similar in that Bridge makes things work before they’re designed and Adapter makes them work after.  **Façade** and Adapter are similar in that Adapter reuses an object’s interface while Façade creates a new one. |
| Code | public interface Target {  public void doThis();  }  public class Adapter implements Target {  private Adaptee adaptee = new Adaptee();  public void doThis() {  // do work if necessary  adaptee.doThat();  }  }  public class Adaptee {  public void doThat() {  // do something  }  } |

# Bridge

|  |  |
| --- | --- |
| Issue | Need to an abstraction and its implementation to be able to change independently of the other. |
| Solution | Define an Abstraction object that holds onto its Implementation object and calls functions on that Implementation. The Implementation object extends from its base class and defines its function specific to itself. |
| Related | **Wrapper** is used in Bridge’s implementation.  **Adapter** and Bridge are similar in that Bridge makes things work before they’re designed and Adapter makes them work after.  **Strategy**, **State**, and Bridge are very similar except that they solve different issues. |
| Code | public interface Implementor {  public void doThis();  }  public class ImplementorA implements Implementor {  public void doThis() {  // do something  }  }  public class ImplementorB implements Implementor {  public void doThis() {  // do something else  }  }  public class Abstraction {  private Implementor implementor;  public void setImplementor(Implementor anImplementor) {  implementor = anImplementor;  }  public void doThis() {  implementor.doThis();  }  } |

# Builder

|  |  |
| --- | --- |
| Issue | Need to construct complex objects while allowing the same construction process to be reused to construct other objects. |
| Solution | Define the objects Product, ProductBuilder, and Director. Product is the object that is constructed from the process. The ProductBuilder is used to provide the specifications for constructing the product. The Director actually constructs the Product using the Builder. This also provides the ability to use optional parameters without creating multiple constructors. |
| Related | **Abstract Factory** and Builder are very similar, except that Builder offers more control over the construction process.  **Singleton** can be used in Builder’s implementation. |
| Code | public abstract class ProductBuilder {  protected Product product;  public Product getProduct() {  return product;  }  public void createNewProduct() {  product = new Product();  }  public abstract void buildParamA();  public abstract void buildParamB();  }  public class ProductBuilderA extends ProductBuilder {  public void buildParamA() {  product.setParamA(“bar”);  }  public void buildParamB() {  product.setParamB(1);  }  }  public class Product {  private String paramA = “”;  private int paramB = 0;  public setParamA(String aParam) {  paramA = aParam;  }  ...  }  public class Director {  private ProductBuilder productBuilder;  public void setProductBuilder(ProductBuilder aProductBuilder) {  productBuilder = aProductBuilder;  }  public Product getProduct() {  return productBuilder.getProduct();  }  public void constructNewProduct() {  productBuilder.createNewProduct();  productBuilder.buildParamA();  productBuilder.buildParamB();  }  }  public class Example {  public static void main(String[] args) {  Director director = new Director();  ProductBuilder productBuilderA = new ProductBuilderA();  director.setProductBuilder(productBuilderA);  director.constructNewProduct();  Product product = direct.getProduct();  }  } |

# Chain of Responsibility

|  |  |
| --- | --- |
| Issue | Need to efficiently process a sender’s requests among multiple handlers without hard-coding sender-receiver relationships |
| Solution | The receivers are chained together in a pipeline (such as a linked list) that the request is passed down in until it reaches a receiver capable of handling the request. This will decouple the sender from a single receiver and allow the possibility of other objects handling the requests without modifying the receivers. |
| Related | **Null Object** is at the end of a Chain of Responsibility.  Chain of Responsibility, **Command**, and **Observer** all deal with decoupling a sender from a receiver. |
| Code | public abstract class Handler {  protected Handler next = null;  public abstract void handle(Request aRequest);  public void setNext(Handler aNext) {  next = aNext;  }  }  public class HandlerA extends Handler {  public void handle(Request aRequest) {  if (/\* meets criteria to handle\*/) {  // handle request  } else if (next) {  next.handle(aRequest);  }  }  }  ...  public class Chain {  private Handler head = null;  public Chain(Request aRequest) {  head = new HandlerA();  Handler nextHandler = new HandlerB();  Handler nextNextHandler = new HandlerC();  head.setNext(nextHandler);  nextHandler.setNext(nextNextHandler);  head.handle(aRequest);  }  } |

# Command

|  |  |
| --- | --- |
| Issue | Need to send series of requests to the same object. |
| Solution | Encapsulate a request as an object using command, receiver, invoker, and client objects. The Command object is used to call a method in the receiver using execute(), which decouples the sender and the receiver. The Invoker object is used to queue and execute commands as well as any logging. The Client object holds onto the others and decides when and which commands to execute. |
| Related | **Chain of Responsibility** can use Command to represent requests as objects.  Chain of Responsibility, **Observer**, and Command all deal with decoupling a sender from a receiver.  **Composite** and Command can be used together to create macros by using Command to create a series of composite commands. |
| Code | public abstract class Command {  protected Receiver receiver = null;  public Command(Receiver aReceiver) {  receiver = aReceiver;  }    public abstract void execute();  }  public class CommandA extends Command {  public void execute() {  receiver.actionA();  }  }  ...  public class Receiver {  public void actionA() {  // do something  }  ...  }  public class Invoker {  private List<Command> commandList = new ArrayList<Command>();  public void addCommand(Command aCommand) {  commandList.add(aCommand);  }  public void executeCommands() {  for (int i = 0; i < commandList.size(); i++) {  commandList.get(i).execute();  }  commandList.clear();  }  }  public class Client {  public Client() {  Invoker invoker = new Invoker();  Receiver receiver = new Recevier();  invoker.addCommand(new CommandA(receiver));  invoker.addCommand(new CommandB(receiver));  ...  invoker.executeCommands();  }  } |

# Composite

|  |  |
| --- | --- |
| Issue | Need to treat a group of objects uniformly. |
| Solution | Define an object, Component, using the desired functions a group of objects should use. These objects, be them individual (Leaf) or an actual composite of objects (Composite), implement Component and add their own functionality. Composite is further expanded with the ability to hold onto individual objects and will call a function from Component on each of its children. |
| Related | **Decorator** is like a Composite with only one component.  **Command** and Composite can be used together to create macros by using Command to create a series of composite commands. |
| Code | public interface Component {  public void doThis();  }  public class Leaf implements Component {  public void doThis() {  // do something  }  }  public class Composite implements Component {  private List<Component> leaves = new ArrayList<Component>;  public void doThis() {  for (int i = 0; i < leaves.size(); i++) {  leaves.get(i).doThis();  }  }  public void add(Component aComponent) {  leaves.add(aComponent);  }  public void remove(Component aComponent) {  leaves.remove(aComponent);  }  } |

# Decorator

|  |  |
| --- | --- |
| Issue | Need to add functionality to an existing object without modifying it. |
| Solution | Define a Decorator object that holds onto a Component object and wrap its methods in the Decorator’s own with additional functionality. The component object is the object that needs have the functionality added to it. |
| Related | Decorator is a type of **Wrapper**.  **Adapter** and Decorator are similar in that Adapter changes an object’s interface and Decorator enhances an object’s interface.  Decorator is like a **Composite** with only one component. |
| Code | public interface Component {  public void doThis();  }  public class ComponentA implements Component {  public void doThis() {  // do something  }  }  public abstract class Decorator {  private Component component;  public Decorator(Component aComponent) {  component = aComponent;  }  public void doThis() {  component.doThis();  }  }  public class DecoratorA extends Decorator {  public void doThis() {  super.doThis();  // do something in addition to the component  }  } |

# Double Dispatch

|  |  |
| --- | --- |
| Issue | Need to call functions based on the object type of both the sender and receiever. |
| Solution | Define an object, Bar, for the senders to implement that the receivers will call when it receives function doThis() with Bar passed in. The receivers will respond with doThat() with themselves passed in. The sender will then run the appropriate function that corresponds to the receiver. |
| Related | **Visitor** is a type of Double Dispatch. |
| Code | public interface Foo {  public void doThis(Bar aBar);  }  public interface Bar {  public void doThat(FooA fA);  public void doThat(FooB fB);  }  public class FooA implements Foo {  public void doThis(Bar aBar) {  aBar.doThat(this);  }  }  ...  public class BarA implements Bar {  public void doThis(FooA fA) {  // do something specific to BarA and FooA  }  public void doThis(FooB fB) {  // do something specific to BarA and FooB  }  }  public class BarB implements Bar {  public void doThis(FooA fA) {  // do something specific to BarB and FooA  }  public void doThis(FooB fB) {  // do something specific to BarB and FooB  }  } |

# Façade

|  |  |
| --- | --- |
| Issue | Need to hide the complexities of a subsystem and provide a simplified interface. |
| Solution | Define a Facade object that wraps the subsystems and redirect its methods to the subsystems’ methods. This will allow only one one object the client has to make calls to. |
| Related | Façade is a type of **Wrapper**.  Façade can be a type of **Singleton**.  **Adapter** and Façade are similar in that Adapter reuses an object’s interface while Façade creates a new one. |
| Code | public interface Subsystem {  public void doThis();  }  public class SubsystemA implements Subsystem {  public void doThis() {  // do something  }  }  public class SubsystemB implements Subsystem {  public void doThis() {  // do something else  }  }  public class Facade {  private SubsystemA subsystemA = new SubsystemA();  private SubsystemB subsystemB = new SubsystemB();  public void doThat() {  subsystemA.doThis();  }  public void doThatToo() {  subsystemB.doThis();  }  }  public class Example {  public static void main(String[] args) {  private Facade facade = new Facade();  facade.doThat();  façade.doThatToo();  }  } |

# Factory Method

|  |  |
| --- | --- |
| Issue | Need to create an object, but don’t know what kind of object to create. |
| Solution | Define a Factory object that returns a newly created Product object. This allows an object’s creation to be deferred and for the Factory to decide which type of Product to create. |
| Related | Factory Method can be used in **Abstract Factory**’s and **Template Method**’s implementation.  **Prototype** deals with creation through delegation and Factory Method deals with creation through inheritance.  **Singleton** can be used in Factory Method’s implementation. |
| Code | public interface Product() { ... }  public class ProductA implements Product() { ... }  public class ProductB implements Product() { ... }  public abstract class Factory() {  public abstract Product factoryMethod();  }  public class FactoryA implements Factory() {  public Product factoryMethod() {  return new ProductA();  }  }  public class FactoryB implements Factory() {  public Product factoryMethod() {  return new ProductB();  }  } |

# Null Object

|  |  |
| --- | --- |
| Issue | Need to do nothing or a default action in the absence of an object. |
| Solution | Define an object, Foo, which can be implemented by another object, NullFoo. Override the functions in NullFoo to either do nothing or do a default behavior. |
| Related | Null Object is oftentimes a type of **Singleton**.  Null Object is at the end of a **Chain of Responsibility**. |
| Code | public abstract class Foo {  public abstract void doThis();  }  public class NullFoo extends Foo {  private static final NullFoo INSTANCE = new NullFoo();  private NullFoo () {}    public static void doThis() {  // do a default action or nothing  }  public static NullFoo getInstance() {  return instance;  }  } |

# Observer

|  |  |
| --- | --- |
| Issue | Need in layered architectures to get notified when the state of an object changes, such that lowers layers do not have access to upper layers. |
| Solution | Define a Subject object that maintains a list of observers. When the state of Subject changes, notify all its observers by calling update() on each. The Observer objects will then get the Subject’s updated state and carry out their specific functionality on update. |
| Related | **Chain of Responsibility**, **Command**, and Observer all deal with decoupling a sender from a receiver. |
| Code | public abstract class Observer {  protected Subject subject;  public Observer(Subject aSubject) {  subject = aSubject;  subject.add(this);  }  public abstract void update();  }  public class ObserverA extends Observer {  public void update() {  // do something with subject.getState();  }  }  public class Subject {  private List<Observer> observers = new ArrayList<Observer>;  private int state;  public Subject getState() {  return state;  }  public void setState(int aState) {  state = aState;  notifyObservers();  }    public void add(Observer anObserver) {  observers.add(anObserver);  }  public void notifyObservers() {  for (int i = 0; i < observers.size(); i++) {  observers.get(i).update();  }  }  } |

# Proxy

|  |  |
| --- | --- |
| Issue | Need to control access to an object. |
| Solution | Define a Proxy object with the exact interface as the object, Subject, whose access is being controlled that holds onto the Subject object. When a method is called in the Proxy, the Proxy will initialize the Subject if it hasn’t been already then call the method. This will prevent the object from being initialized until it is needed. |
| Related | Proxy is a type of **Wrapper**. |
| Code | public interface Subject {  public void doThis();  }  public class SubjectA implements Subject {  public void doThis() {  // do something  }  }  public class Proxy implements Subject {  private Subject subject;  public void doThis() {  if (!subject) {  subject = new Subject();  }  subject.doThis();  }  } |

# Prototype

|  |  |
| --- | --- |
| Issue | Need to create objects that differ very little and creating the objects from scratch is costly. |
| Solution | Clone the object. |
| Related | **Factory Method** deals with creation through inheritance and Prototype deals with creation through delegation.  **Singleton** can be used in Prototype’s implementation. |
| Code | public interface Prototype extends Cloneable {  public abstract Object clone() throws CloneNotSupportedException;  }  public class PrototypeA implements Prototype {  public Object clone() {  return (PrototypeA)super.clone();  }  } |

# Singleton

|  |  |
| --- | --- |
| Issue | Need only one instance of an object. |
| Solution | Define an object, Singleton, that controls its own creation and can return the single instance it created when the proper function is called. Overriding its constructor to be made private prevents other objects from creating it. |
| Related | **Null Object**, **Façade**, and **State** can be a type of Singleton.  **Abstract Factory**, **Builder**, and **Prototype** can use Singleton in their implementation. |
| Code | public class Singleton {  private static final Singleton INSTANCE = new Singleton();  private Singleton() {}  public static Singleton getInstance() {  return INSTANCE;  }  } |

# State

|  |  |
| --- | --- |
| Issue | Need an object to change its behavior based on its state. |
| Solution | Define a Context object that holds onto its current State object and calls functions on that state. The State object extends from its base class and defines its functions specific to the state. States can be changed either in the State object themselves or in the Context. |
| Related | State can be a type of **Singleton**.  **Bridge**, **Strategy**, and State are very similar except that they solve different issues.  **Flyweight** determines when and how State objects are shared. |
| Code | public interface State {  public void doThis(Context aContext);  }  public class StateA implements State {  public void doThis(Context aContext) {  // do something based on state  aContext.setState(new StateB());  }  }  public class StateB implements State {  public void doThis(Context aContext) {  // do something else based on another state  aContext.setState(new StateA());  }  }  public class Context {  private State state;  public void setState(State aState) {  state = aState;  }  public void doThat() {  state.doThis(this);  }  } |

# Strategy

|  |  |
| --- | --- |
| Issue | Need an object to change its behavior based on an interchangeable algorithm in a family of algorithms. |
| Solution | Define a Context object that holds onto its current Strategy object and calls functions on that strategy. The Strategy object extends from its base class and defines its algorithms specific to the strategy. Strategies can be changed either in the Strategy object themselves or in the Context. |
| Related | **Bridge**, **State**, and Strategy are very similar except that they solve different issues.  **Template Method** can be used in State’s implementation. |
| Code | public interface Strategy {  public void doThis();  }  public class StrategyA implements Strategy {  public void doThis() {  // do something based on strategy  }  }  public class StrategyB implements Strategy {  public void doThis() {  // do something else based on another strategy  }  }  public class Context {  private Strategy strategy;  public void setStrategy(Strategy aStrategy) {  strategy = aStrategy;  }  public void doThis() {  strategy. doThis(this);  }  } |

# Template Method

|  |  |
| --- | --- |
| Issue | Need to redefine certains steps of an algorithm without changing algorithm’s structure. |
| Solution | Define an object, Foo, that implements an algorithm using multiple methods then ties them together in templateMethod(). Objects can extend Foo and override whichever steps deemed necessary. |
| Related | **Strategy** and Template Method both deal with algorithms, but Template Method uses inheritance to change part of the algorithm and Strategy uses delegation to change the entire algorithm. |
| Code | public abstract class Foo {  public abstract void doThis();  public abstract void doThat();  public void templateMethod() {  doThis();  doThat();  }  }  public class FooA extends Foo {  public void doThis() {  // do something  }  public void doThat() {  // do something else  }  } |

# Visitor

|  |  |
| --- | --- |
| Issue | Need to traverse a graph of objects and call functions on them without knowing their type. |
| Solution | Define an object, Visitor, that implements the desired functions per object using visit(). The objects themselves will implement an accept() function that takes a Visitor then call the Visitor’s visit() with themselves passed in as a parameter. This also allows you to create new functions without changing the structure of the objects. |
| Related | **Double Dispatch** is used to implement Visitor in that the visitor calls a different function based on the type of visitor and type of object.  **Interpreter** is a type of Visitor. |
| Code | public interface Foo {  public void accept(Visitor aVisitor);  }  public class FooA implements Foo {  public void accept(Visitor aVisitor) {  aVisitor.visit(this);  }  }  ...  public class Visitor {  public void visit(FooA anA) {  // do something specific to FooA  }  public void visit(FooB aB) {  // do something specific to FooB  }  } |

# Unlisted Patterns

Flyweight, Collecting Parameter, and Interpreter.