|  |
| --- |
|  |

|  |
| --- |
| **ISYS2102 - Software Engineering 2** |
| Pattern report |
|  |
|  |
|  |
| **ABBC Team** |
| **11/24/2012** |
|  |

Table of Contents

1. Bridge design pattern

[I. Bridge design patter 4](#_Toc341529700)

[II. Purposes of Bridge design pattern 4](#_Toc341529701)

[III. Example of Bridge pattern 5](#_Toc341529702)

[1. Inheritance vs. Bridge pattern 5](#_Toc341529703)

[2. Code example 6](#_Toc341529704)

[IV. Advantages and disadvantages of Bridge pattern 7](#_Toc341529705)

[1. Advantages in using Bridge pattern 7](#_Toc341529706)

[2. Disadvantages in using Bridge pattern 8](#_Toc341529707)

[V. When should we use Bridge pattern? 8](#_Toc341529708)

[VI. Similarity with Adapter pattern 8](#_Toc341529709)

[1. Similarity: 8](#_Toc341529710)

[2. Differences: 8](#_Toc341529711)

1. Facade design pattern

[I. Facade design pattern 10](#_Toc341529712)

[II. Motivation 10](#_Toc341529713)

[III. Example of Facade pattern 10](#_Toc341529714)

[IV. Benefits of Façade pattern 11](#_Toc341529716)

[V. When should the Façade pattern be applied? 11](#_Toc341529717)

[VI. Related Patterns 11](#_Toc341529718)

[3. Abstract Factory: 11](#_Toc341529719)

[4. Mediator: 11](#_Toc341529720)

1. Factory design pattern

[I. Overview 13](#_Toc341529721)

[II. Reason for applying Factory method pattern, its advantages and disadvantages 14](#_Toc341529722)

[1. Application: 14](#_Toc341529723)

[2. Advantages: 14](#_Toc341529724)

[3. Disadvantages: 14](#_Toc341529725)

[III. Example of applying Factory method 14](#_Toc341529726)

[IV. Alternative Choice 17](#_Toc341529727)

1. Visitor design pattern

[I. Intent 19](#_Toc341529728)

[II. Implementation 20](#_Toc341529729)

[Example: - Car Application. 21](#_Toc341529730)

[III. Advantages 24](#_Toc341529731)

[IV. Disadvantage 24](#_Toc341529732)

[V. Alternative 24](#_Toc341529733)

[1. Differences: 24](#_Toc341529734)

[2. Similarities: 24](#_Toc341529735)

[3. When use Stategy Pattern: 24](#_Toc341529736)

[4. When use Visitor Pattern: 25](#_Toc341529737)

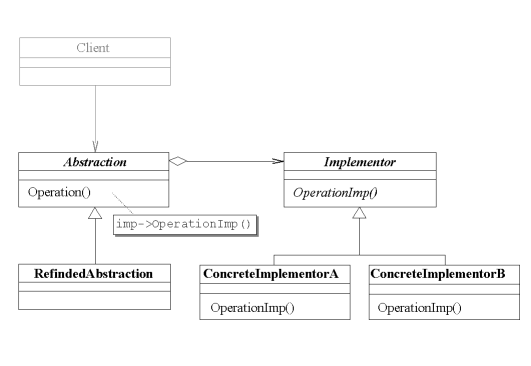
1. References

[References 26](#_Toc341529738)

Bridge Design Pattern

# Bridge design patter

Bridge design pattern is a complex pattern which is categorized as Structural Design. Bridge design pattern is intended to decouple an abstraction from its implementation so that the two can vary independently (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1994).

Bridge pattern structure

# Purposes of Bridge design pattern

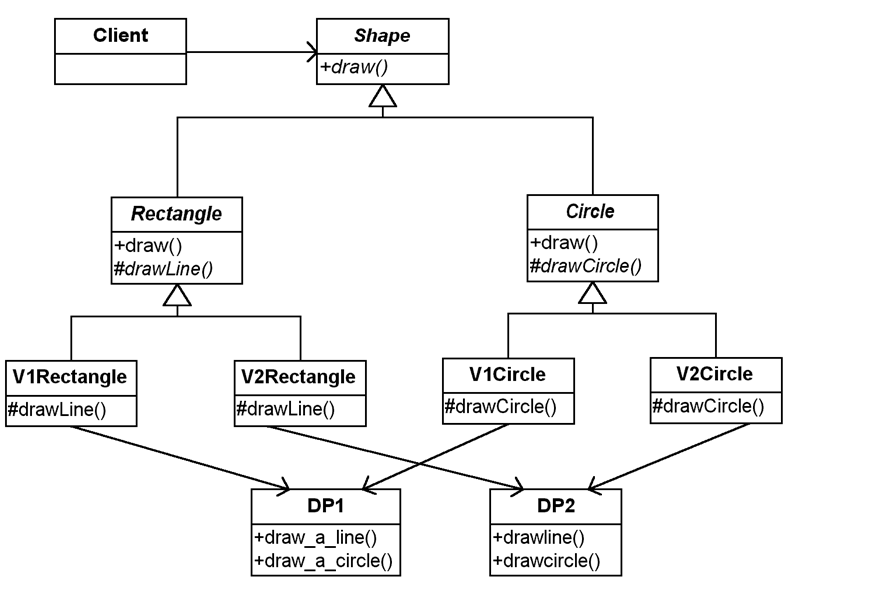
Normally, an abstraction has several possible implementations we use inheritance to accommodate them. Abstract class defines the interface to abstraction and concrete subclasses implement in different ways. According to (Alan Shalloway, James Trott, James Trott, 2002), the above methodology is not flexible in some cases due to the permanent binding of inheritance on implementation. This leads to the difficulty for modification, extensibility, reusability of abstractions and implementation.

Instead of combining the abstraction and the implementation, Bridge pattern separate them in different class hierarchies. One class hierarchy is for interfaces and the other is for specific implementations. Example and comparison between raw use of inheritance and Bridge pattern will be presented in the next section.

# Example of Bridge pattern

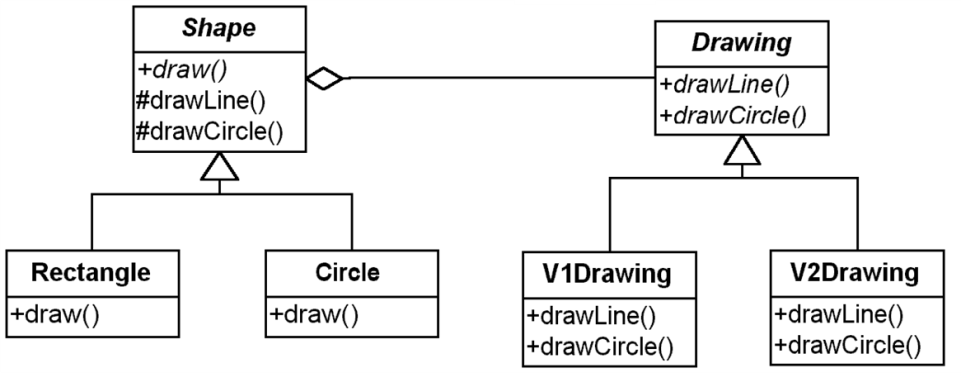
In this section, the comparison between raw inheritance and Bridge design pattern will be presented and discussed based on the example. Some other example of Bridge pattern will also be presented.

## Inheritance vs. Bridge pattern

Inheritance [[](http://www.jedeksvault.nl/wp-content/uploads/2011/10/solution-without-bridge.png)](http://www.jedeksvault.nl/wp-content/uploads/2011/10/solution-without-bridge.png)

In this case, both DP1 and DP2 draw line and circle but the handle is different. The above design results in tight coupling which reduce the flexibility dramatically. For example, when we want to add another shape this design will show its problem. Now we will look at Bridge pattern based design.

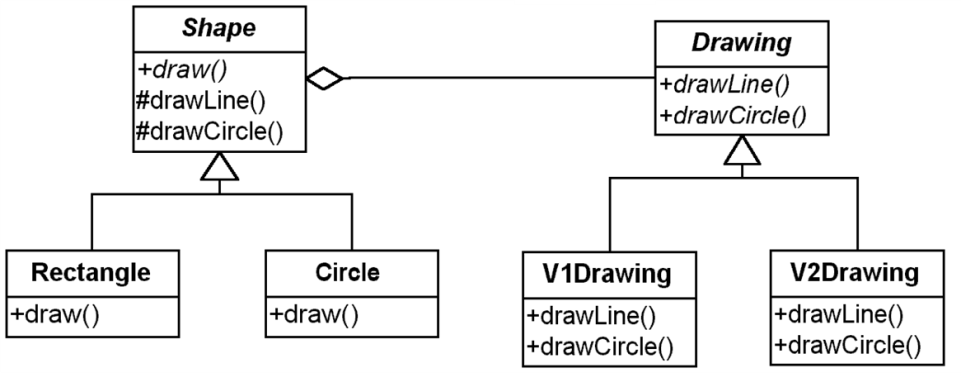
Bridge pattern

[](http://www.jedeksvault.nl/wp-content/uploads/2011/10/bridge-design-pattern.png)

This design separate the shapes and drawing into their own classes. As we can see clearly, compare to raw inheritance which binds implementation to abstraction permanently, Bridge pattern offers more flexibility in term of extensibility and reusability of implementation as it decouples the shapes and drawing. It will be convenient in case we want to add new drawings or new shapes.

## Code example

This section will provide the code for the Bridge pattern based design above.

[](http://www.jedeksvault.nl/wp-content/uploads/2011/10/bridge-design-pattern.png)

Drawing:

public interface Drawing

{

public void drawLine();

public void drawCircle();

}

//V1Drawing

public class V1Drawing implements Drawing

{

public void drawLine()

{

//V1Drawing specific action

}

public void drawCircle()

{

//V1Drawing specific action

}

}

//V2Drawing

public class V2Drawing implements Drawing

{

public void drawLine()

{

//V2Drawing specific action

}

public void drawCircle()

{

//V2Drawing specific action

}

}

Shape:

//Shape

public abstract class Shape

{

private Drawing drawProgram;

Shape(Drawing draw)

{

drawProgram = draw;

}

abstract void draw();

void drawLine()

{

drawProgram.drawLine();

}

void drawCircle()

{

drawProgram.drawCircle();

}

}

// Circle

public class Circle extends Shape

{

Circle(Drawing draw)

{

super(draw);

}

public void draw()

{

drawCircle();

}

}

// Rectangle

public class Rectangle extends Shape

{

Rectangle(Drawing draw)

{

super(draw);

}

public void draw()

{

drawLine();

drawLine();

drawLine();

drawLine();

}

}

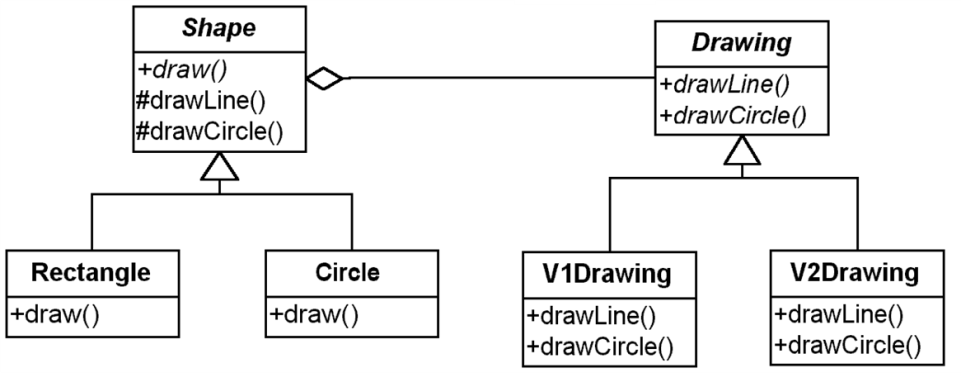
# Advantages and disadvantages of Bridge pattern

## Advantages in using Bridge pattern

* Avoid the binding between abstraction and implementation and select to implementation at the run time.
* Reduce number of sub classes.
* Clear code and reduction in executable size (Lasater, 2010).
* Interface and implementation can be varied independently.
* Abstraction and implementation can be extended independently.

## Disadvantages in using Bridge pattern

* According to (Lasater, 2010), Bridge pattern results in double indirection. In the example below, the Shape class must delegate message to Drawing subclass which implements appropriate method. This will impact the performance slightly.

[](http://www.jedeksvault.nl/wp-content/uploads/2011/10/bridge-design-pattern.png)

* Multiple inheritances.

# When should we use Bridge pattern?

* When we want to avoid a permanent binding between abstraction and its implementation. For example, when the Implementation must be selected or switched at runtime.
* When both abstraction and its implementation need to be extensible by subclass-ing as Bridge pattern allow we to combine different abstractions and implementations and extend them independently.
* When changes in the implementation of abstraction have no impact on clients.
* In C++, when we need to hide the implementation from the clients as in C++ the representation of class is visible in the class interface (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1994).
* When we want to share an implementation among multiple objects.

# Similarity with Adapter pattern

## Similarity:

Both used to hide the details of the underlying implementation.

## Differences:

The adapter pattern is geared towards making unrelated components work together.

A bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.

Bridge can abstract a complex entity from its implementation while Adapter only abstracts a single interface.

Facade Design Pattern

# Facade design pattern

Façade design pattern is an object structural pattern. Façade pattern provides a unified interface to a sex of interfaces in the subsystem. Façade is used to simplify the subsystem. (Lasater, 2010)

Façade structure



# Motivation

Façade pattern structures a system into subsystems helps reduce complexity. The subsystems are groups of classes, or groups of classes and other subsystems. The interface exposed by the classes in a subsystem or set of subsystems can become quite complex. One way to reduce this complexity is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.

# Example of Facade pattern

For our project, as we apply MVC structure, we use Façade pattern to provide the interface for model subsystem to communicate with the client (in this case is controller).

## C:\Users\Johansen\Desktop\rmit-se2-2012C-ABC\DevFortress\dist\javadoc\overview0W0H0.pngC:\Users\Johansen\Desktop\rmit-se2-2012C-ABC\DevFortress\dist\javadoc\overview0W1H0.png

# Benefits of Façade pattern

* It provides interface for clients to communicate with subsystem therefore reducing the number of objects the clients deal with.
* Demonstrate weak coupling between subsystem and the clients.
* Reducing compilation dependencies in large software system.
* Façade doesn’t prevent applications from using subsystem class which provide both ease of use and generality.

# When should the Façade pattern be applied?

* When we want to reduce complexities of a system.
* When we need to decouple subsystems, its dependencies or improve portability.
* When we need an entry to the subsystems.
* Minimize the communication between subsystems.
* Security and performance consideration.

# Related Patterns

## Abstract Factory:

Abstract Factory can be used with Façade to provide an interface for creating subsystem objects in sub-system-independent way. Abstract Factory can also be alternative to Façade in case we need to hide platform- specific classes.

## Mediator:

Mediator is similar to Façade as it abstract functionalities of existing classes. However, Mediator’s motivation is to abstract arbitrary communication between colleague objects (centralize functionality that doesn’t belong in any of them) (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1994).

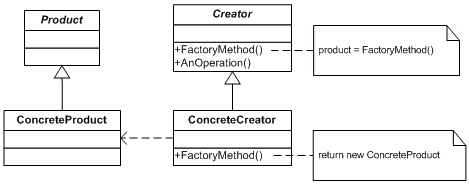
A bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.

Bridge can abstract a complex entity from its implementation while Adapter only abstracts a single interface.

Factory Design Pattern

# Overview

Factory method design pattern is applied for the intention of define an interface that creates an object without specifying the exact class of that object. The returned product type will be defined depending on the subclass of the interface.

Factory method structure:

# Reason for applying Factory method pattern, its advantages and disadvantages

## Application:

* Applications having class that cannot anticipate the class of object it should create.
* A class wants it subclass to decide the object it creates.

## Advantages:

* Since **new()** operator is considered harmful because it creates new object without encapsulation object creation. So, we don’t want to call it when not really necessary. To avoid calling **new()** by mistake when we have to create many objects, we can use Factory method to return the needed object without retyping **new()** all the time thus significantly reducing duplication of creation code.
* Using Factory method will enforce encapsulation thus we can hide the concrete class from client.
* Factory method centralizes object creation into a single method of class. This will extremely helpful to maintain a big application and make it more flexible since when we need to extend or change the type of the object we don’t need to go through the whole implemented code to find and change them but only need to go to the Factory and do the job.
* With Factory method we don’t need to know exactly which class of object we will return as we coding to an interface.
* Low coupling.

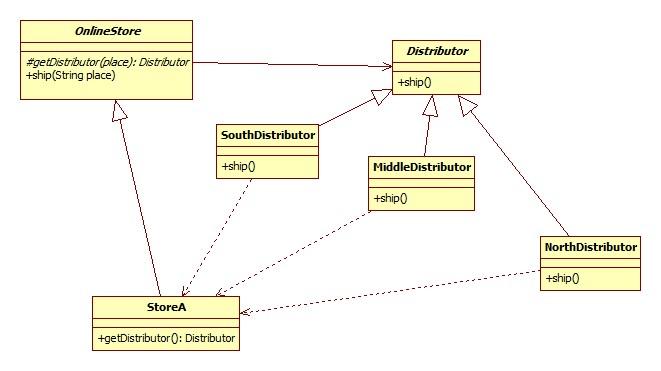
## Disadvantages:

* When we want to change the interface Factory method (such as adding parameter to or removing parameter from the method) we need to go through all the derived classes and change the implementation of the method.

# Example of applying Factory method

- Factory method should be used in Applications that defines an interface to create object but allow subclasses decide which class to instantiate.

**Description**: we open an online shop that sells and ships product bought by online customers. As an online store, we can display our products and describe those product information via a website so we don’t really need to stock up our products. This will lower the fund for our shop and reduce the risk of storing too them long without any customer to buy. When customers want to buy anything they request the product, pay money via our website and wait for it to be shipped to their place. When we get request and money, we just make another request to the distributor and they will send the product to our customer’s house. The job is easy but since customers might live all around the country, we need to find distributors who are near them to lower the shipping fee. Customers will not be concerned about those distributors, they just get what they request from our website. So, we’ll build an application that will request the product from appropriate distributor with Factory Method pattern



**Code implementation**

// OnlineStore

public abstract class OnlineStore {

public void ship(String place) {

Distributor d = getDistributor(place);

d.ship();

}

protected abstract Distributor getDistributor(String place);

}

// StoreA

public class StoreA extends OnlineStore() {

Distributor d;

protected Distributor getDistributor(String place) {

if(place == “North”) {

d = new NorthDistributor();

}

else if(place == “South”) {

d = new SouthDistributor();

}

else {

d = new MiddleDistributor();

}

return d;

}

}

//Distributor

public abstract class Distributor() {

String place;

void ship(){

System.out.println(“Ship to ”+ place);

}

}

//NorthDistributor

public class NorthDistributor extends Distributor() {

public NorthDistributor(){

place = “North”;

{

}

//SouthDistributor

public class SouthDistributor extends Distributor() {

public SouthDistributor() {

place = “South”;

{

}

//MiddleDistributor

public class MiddleDistributor extends Distributor() {

public MiddleDistributor(){

place = “Middle”;

{

}

# Alternative Choice

\* **Prototype Pattern** will be considered an alternative choice for Factory Method.

**- Similarities**

+ Allow object to create other objects without knowing the exact class or details of how to create them.

**- Differences**

+ While Factory Method creates new object as invoking the constructor of a concrete implementation class Prototype creates new object as a copy of the prototype object.

+ Prototype doesn’t require subclassing but it requires an initialize operation. On the other hand, Factory Method require subclassing but initialization is not needed.

+ Factory Method is the creation through inheritance. Prototype is the creation through delegation.

**\* When we should choose Prototype Pattern over Factory Method**

When we only need the copy of target object. For example, there is a report object in which data is store in some specific order, now we want other copies of that reports object which contain data in different order. In that case, we only need copies of that object rather than create a bunch of new objects. With Prototype Pattern we can increase performance of the product as cloning is much faster and less expensive than creating new object.

**\*** **Disadvantage of using Prototype Pattern**

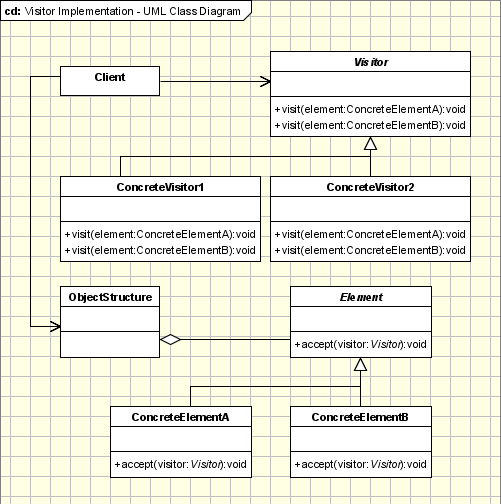
Despite the improvement on performance, Prototype pattern can get you in trouble if you don’t think carefully enough where to apply the pattern. For example, for an object that creates database connection, cloning will give better performance than creating new. The problem is that the clone and the original object will have a reference to the same database connection. This will cause a lot of troubles.

Visitor Design Pattern

# Intent

* Performed an operation on the elements of an object structure.
* Visitor lets you define a new operation without changing the definition of the elements on which it operates.

# Implementation

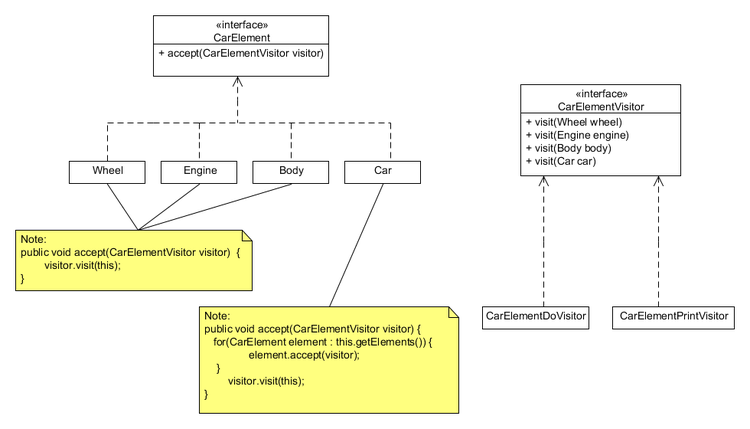


The participants classes in this pattern are:

* **Visitor** - This is an interface or an abstract class used to declare the visit operations for all the types of visitable classes. It also define visit() method. Each Visitor can have many visit() method and they are separate by different signature.
* **ConcreteVisitor** - Each Visitor will invoke different operations. When a new visitor is added, it must be passed to the object structure.
* **Element**- is an interface which declares the accept operation.
* **ConcreteVisitable** - Implements the Visitable interface and defines the accept operation. Visitor objects is passed to this object by accept method.

### Example: - Car Application.

The following example shows how the contents can be printed. Instead of creating "print" methods for each subclasses, we use the Visitor Pattern. CarElementDoVisitor invokes different actions based on the class of the argument passed to it.



|  |
| --- |
| **interface** CarElementVisitor {  **void** visit(Wheel wheel);  **void** visit(Engine engine);  **void** visit(Body body);  **void** visit(Car car); }   **interface** CarElement {  **void** accept(CarElementVisitor visitor); *// CarElements have to provide accept().* }   **class** Wheel **implements** CarElement {  **private** String name;    **public** Wheel(String name) {  **this**.name = name;  }    **public** String getName() {  **return** **this**.name;  }    **public** **void** accept(CarElementVisitor visitor) {  */\*  \* accept(CarElementVisitor) in Wheel implements  \* accept(CarElementVisitor) in CarElement, so the call  \* to accept is bound at run time. This can be considered  \* the first dispatch. However, the decision to call  \* visit(Wheel) (as opposed to visit(Engine) etc.) can be  \* made during compile time since 'this' is known at compile  \* time to be a Wheel. Moreover, each implementation of  \* CarElementVisitor implements the visit(Wheel), which is  \* another decision that is made at run time. This can be  \* considered the second dispatch.  \*/*   visitor.visit(**this**);  } }   **class** Engine **implements** CarElement {  **public** **void** accept(CarElementVisitor visitor) {  visitor.visit(**this**);  } }   **class** Body **implements** CarElement {  **public** **void** accept(CarElementVisitor visitor) {  visitor.visit(**this**);  } }   **class** Car **implements** CarElement {  CarElement[] elements;    **public** Car() {  *//create new Array of elements*  **this**.elements = **new** CarElement[] { **new** Wheel("front left"),   **new** Wheel("front right"), **new** Wheel("back left") ,   **new** Wheel("back right"), **new** Body(), **new** Engine() };  }    **public** **void** accept(CarElementVisitor visitor) {   **for**(CarElement elem : elements) {  elem.accept(visitor);  }  visitor.visit(**this**);   } }   **class** CarElementPrintVisitor **implements** CarElementVisitor {  **public** **void** visit(Wheel wheel) {   System.out.println("Visiting " + wheel.getName() + " wheel");  }    **public** **void** visit(Engine engine) {  System.out.println("Visiting engine");  }    **public** **void** visit(Body body) {  System.out.println("Visiting body");  }    **public** **void** visit(Car car) {   System.out.println("Visiting car");  } }   **class** CarElementDoVisitor **implements** CarElementVisitor {  **public** **void** visit(Wheel wheel) {  System.out.println("Kicking my " + wheel.getName() + " wheel");  }    **public** **void** visit(Engine engine) {  System.out.println("Starting my engine");  }    **public** **void** visit(Body body) {  System.out.println("Moving my body");  }    **public** **void** visit(Car car) {  System.out.println("Starting my car");  } }   **public** **class** VisitorDemo {  **static** **public** **void** main(String[] args) {  Car car = **new** Car();  car.accept(**new** CarElementPrintVisitor());  car.accept(**new** CarElementDoVisitor());  } } |

# Advantages

* Easily add functions to class libraries even when you do not know the source.
* Obtain data from a disparate collection of unrelated classes without difficulty.
* Collaborate with the Composite pattern.

# Disadvantage

* Not suitable for unstable visited classes, Visitors have to be amended whenever new Composite derived classes are added.
* In classic implementation of the Visitor pattern the type of visitor methods has to be known in advance.

# Alternative

## Differences:

* The Strategy pattern is designed to have operation decided at runtime and interchangeable. While Visitor patterns have a Composite Association between the Host and the Visitor and therefore the Visitor containing the operations has the same life cycle as the Host.
* The Visitor must know all Hosts and provide operations for each. The Visitor controls the operations run by the Host.
* Strategy Pattern provides only the required algorithms, and the decision over which operation is executed is handled by the Context.

## Similarities:

* Allow separation between concreted object with operation, therefore user can add new operation without any amendment in derived classes.

## When use Stategy Pattern:

* A few algorithms will be used by many different classes.
* Different algorithms may be used at different times.
* Operation requires data that the Object shouldn't know about.
* Classes are using multiple conditional statements. These can be moved to an implementation of the Strategy class.
* An object structure is likely to change often.

## When use Visitor Pattern:

* A stable object structure and operations on those structures will be amended often.
* Programmer want to encapsulate functionality for each concrete class
* Operation requires data that the Object shouldn't know about.
* Programmer want a maintainable state within operations across multiple objects.

# References

Alan Shalloway, James Trott, James Trott. (2002). *Design Patterns Explained: A New Perspective on Object-Oriented Design.* Boston: Addison-Wesley Professional.

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Indianapolis: Pearson.

Lasater, C. (2010). *Design Patterns.* Plano: Wordware Publishing, Inc.

Sourcemaking**.** Factory Method Design Pattern. *Sourcemaking.* [Online] Sourcemaking. [Cited: November 23, 2012.] http://sourcemaking.com/design\_patterns/factory\_method.

**—.** Prototype Design Pattern. *Sourcemaking.* [Online] Sourcemaking. [Cited: November 23, 2012.] http://sourcemaking.com/design\_patterns/prototype.

StackOverflow**.** Questions about the Prototype Pattern. *StackOverflow.* [Online] [Cited: November 22, 2012.] http://stackoverflow.com/questions/5739240/questions-about-the-prototype-pattern.