

---

# Chapter 8. Design Patterns

## Table of Contents

Objectives .....	1
Introduction to design patterns .....	1
The idea of a pattern .....	1
The origins of design patterns .....	4
Patterns in software design .....	4
Design patterns in object-oriented programming .....	5
Definitions of terms and concepts .....	5
Scope of development activity: applications, toolkits, frameworks .....	6
Pattern classifications and pattern catalogue .....	7
Behavioural patterns .....	8
Creational patterns .....	9
Structural patterns .....	11
How to use a design pattern .....	12
Patterns in Java .....	12
The <i>Observer</i> pattern in Java .....	12
The <i>Model-View-Controller</i> pattern .....	16
<i>Abstract factory</i> facilities in Java .....	16
<i>Composite</i> patterns in Java .....	18
Review .....	19
Questions .....	19
Answers .....	20

## Objectives

At the end of this chapter you should be able to:

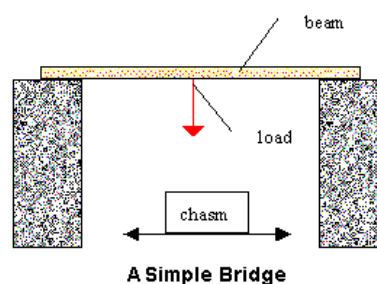
- Describe the provenance of design patterns and explain their potential use in the design process.
- Select a specific design pattern for the solution of a given design problem.
- Create a catalogue entry for a simple design pattern whose purpose and application is understood.

## Introduction to design patterns

### The idea of a pattern

A bridge is a structure used for traversing a chasm. In its basic form it consists of a beam constructed of rigid material, the two ends of the beam fixed at opposite ends of the chasm.

**Figure 8.1. A simple pattern for a bridge**



The bridge will fulfil its function if the rigidity of the beam can support the loads which traverse it. The beam's rigidity depends on the material of its construction and its span. In situations where the heaviness of the load, the length of the span or the material of construction are likely to lead to failure, the design of the bridge needs to be modified. More rigid materials are generally more expensive and/or more difficult to work with, so we shall ignore this possibility. This leaves two possible approaches:

1. Increasing the rigidity.
2. Decreasing the span.

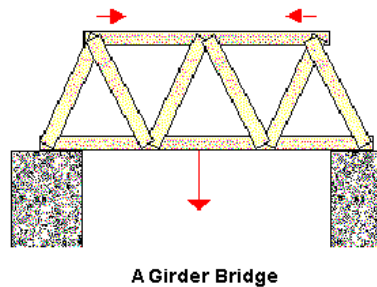
## Increasing the rigidity

The rigidity of the bridge structure can be improved by supporting the beam in a number of ways.

### The girder

We can redistribute the beam's material to improve its rigidity.

**Figure 8.2. The girder**

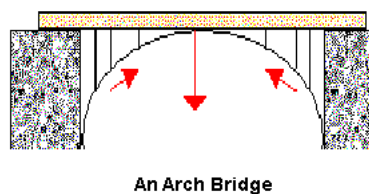


Some of the force of the load on the lower beam is distributed by the cross-members resulting in a compressive force in the upper beam. The (same) material of construction of the upper beam is better able to support compressive forces along its length.

### The arch

Some of the force of the load on the beam is distributed compressively along the material of the arch.

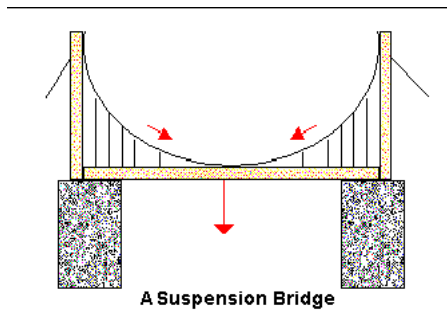
**Figure 8.3. The arch**



The arch must be specially shaped so that the forces remain compressive along the length of the arch. This shape is called a catenary.

### Suspension

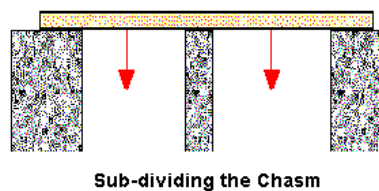
The arch can be replaced by a cable which supplies the same support from above rather than below.

**Figure 8.4. Suspension**

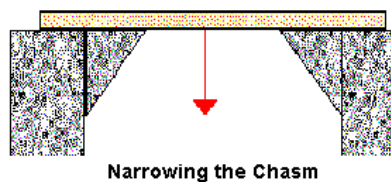
The cable hangs in a catenary shape.

## Decreasing the span

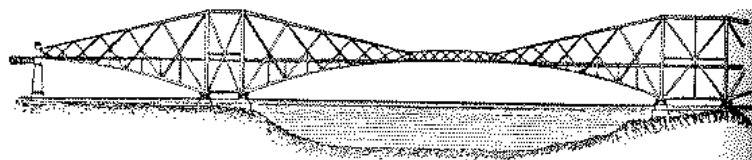
If the chasm is not too deep it may be possible to divide it into two “sub-chasms” by building a support in the middle.

**Figure 8.5. Subdivision**

Alternatively, the edges of the chasm can be extended to reduce the length of the span.

**Figure 8.6. Narrowing**

Some bridges are built by combining several of these approaches into an elegant and functional structure.

**Figure 8.7. The Forth Bridge**

The Forth of Forth Bridge (taken from Encyclopedia Britannica, 1896)

The Forth Bridge is a railway bridge over the Firth of Forth river in Scotland.

## Patterns

These are the *patterns* of bridge design. Even though we have no specialist knowledge of civil engineering, we can see how and why they work, and when we look at a bridge we can see the patterns

which were used to design it because they are built into its structure. Civil engineers, whose work is to build such bridges, learn these patterns along with a great deal of specialised knowledge about construction materials and very sophisticated analytical techniques which help them to make precise predictions about the strength and suitability of different designs and materials before the bridges are built. They use the patterns, either singly or in combination, when thinking about and discussing the design of a particular bridge with other engineers, and they recognise the patterns used in constructing the bridges of their fellow engineers.

## The origins of design patterns

The ideas behind design patterns come not from civil engineers but from architects. Like civil engineers, architects are concerned with designing structures to meet certain functional requirements. Individual buildings need to have a form that helps them to fulfil their purpose. For instance, a dwelling for a single family needs to have a living area, a sleeping area and utility rooms, like a kitchen and a bathroom. A good dwelling design will place these together in such a way that makes family living convenient and pleasant. The sleeping area should be away from the living area so that it is quiet. The bathroom will be directly accessible from many parts of the house, and not, for example, via one of the bedrooms. The kitchen should be accessible from the living area but not from the sleeping area. In addition, the design will make efficient, effective and economic use of the materials of construction, for example roofing, materials and plumbing.

In their working lives architects “solve” these problems over and over again, in slightly different ways and under different circumstances. However, the basic precepts of good dwelling design remains the same, and they are different from the precepts of good office block design, or good hotel design or good hospital design. The design patterns of architects are the solutions they have found to all these various problems. Like civil engineers, architects can see the patterns in the buildings they design and see examples in the designs of other architects every time they walk into a house or drive down a street.

In the 1970's the architect Christopher Alexander wrote a series of books in which he first enunciated the idea of the design pattern, and the ways in which patterns could be used to solve specific architectural problems, and be combined to communicate design ideas amongst architects. He said:

“Each pattern is a rule which describes what you have to do to generate the entity which it defines.”

— Christopher Alexander, *The Timeless Way of Building*

This definition gives a pattern two roles – firstly, as an abstract description of a solution to a particular type of problem; and secondly as a form which we can recognise within an “entity” which solves the problem. Thus a suspension cable is a way of providing support to the beam of a bridge, and when we look at a bridge we can see whether or not it uses a cable for support.

## Patterns in software design

Many aspects of the work of software engineers have parallels with the work of civil engineers and architects. For instance:

- Software engineers design and construct software systems to meet certain functional requirements.
- These software systems may consist of a number of software components which must work together in a structure to deliver the functions.
- The designers must concern themselves with effective, efficient and economic construction and operation.
- There are precepts of good design for various types of software system.
- Software engineers solve variations of particular design problems over and over again.

However, there are also some stark differences.

- Although they may use other engineers' software, software engineers rarely have the opportunity to observe other people's software designs in the same intimate way that an architect can when he or she enters a building.
- Both civil engineering and architecture are very old disciplines stretching back thousands of years. In contrast, software engineering is a new discipline, only a few decades old.

It is not surprising that software engineers would look to more mature design disciplines for some assistance in defining what they were trying to do and in a search for techniques to help them to do it. As explained in ???, the term *software engineering* itself was coined to emphasise the belief that it took more than just skillful programming to produce a good piece of software, and that careful consideration of requirements combined with systematic design and development would help to bring software artefacts up to the same levels of reliability and elegance as well-engineered "hardware".

The work of Alexander was known to two software engineers, Ward Cunningham and Kent Beck, when, in 1987, they visited a client to discuss the design of a user interface. They liked Alexander's idea that a pattern is a symbolic way of describing a solution to a type of problem, and that a set of patterns could provide a language for discussing the problem and considering various solutions. They wanted the users of the proposed system to contribute to the design (another Alexandrian precept), so they invented a small set of user interface patterns for their users. They became convinced of the value of these ideas and their relevance to software engineering when the users produced a very elegant and efficient design using the simple pattern language.

Ward and Cunningham presented their conclusions at a software engineering conference (OOPSLA '87) but few of the delegates were convinced. However, at about the same time, other workers in software engineering were feeling their way towards an architectural view of software engineering and by the 1991 OOPSLA conference the term "design patterns" was in use. A community of software engineers was gradually developing who were taking Alexander's work very seriously. Many of the leading members of this community attended a meeting in 1993. During the meeting, they decided to try designing a building according to Alexander's principles. This included laying out the physical building plan, which they duly did on the side of the hill in Colorado where the meeting was being held. Henceforth, they were known as the *Hillside group*.

Finally, in 1995, four of the main proponents of design patterns, E. Gamma, R. Helm, R. Johnson and J. Vlissides published the first book on design patterns, *Design Patterns: Elements of Reusable Object-Oriented Software*. This is still perhaps the most authoritative book on the subject and much of the ensuing material is drawn from there. The four authors are known in the design pattern community as "The Gang of Four".

## Design patterns in object-oriented programming

### Definitions of terms and concepts

The following is a summary of terms you were already introduced to in the earlier chapters that will be essential for the understanding of design patterns.

#### Object

One of the main tasks of object-oriented design is to identify the classes which make up the software system (see ???).

Not all objects that will be part of a system are identified early on in the development process, for a number of reasons, including the chosen software process (such as incremental processes).

## Interface

The most important aspect of an object is its *interface*. An object's **interface** defines how the object can be used, in other words, to what kind of messages it can respond. The parameters that need to be passed with the message, if any, and the return type are called collectively the operation's signature. The implementation details of these operations do not need to be known to the client.

Many operations with the same name can have different signatures, and many operations with the same signature can have different implementations (using inheritance). These are forms of *polymorphism*. This substitutability — in other words, being able to substitute objects at execution time — is called **dynamic binding**, and is one of the main characteristics of object-oriented software. Objects with identical sets of signatures are said to conform to a common interface.

## Class

A class definition can be used as a basis for defining subclasses by means of inheritance. A subclass possesses all the data and method implementations of the superclass together with additional data and methods pertaining exclusively to objects of the subclass. In some cases, subclass data may shadow superclass data with the same identifiers, or may override methods with the same signature. An abstract class is a class that can have no objects. Its main purpose is to define a common interface shared by its subclasses. Sub-classes specify implementations for these the methods of an abstract class by overriding them.

There is a distinction between *inheritance* and *conformance*. In Java, this is explicitly defined by means of extending a class through inheritance, and by implementing an interface to ensure conformance to certain behaviour. An object's type is defined by its interfaces; this defines the messages to which it can respond or, in other words, how it can be used. A class is a type, but objects of many different classes can have the same type.

## Scope of development activity: applications, toolkits, frameworks

Software developers may find themselves involved in different sorts of software development activities. Most developers work on applications designed to be used by non-specialist computer users to perform tasks relevant to their particular work. However, some developers may be involved in producing specialist software designed to help application software developers in the production of their applications. The products of such developers are variously called *toolkits* or *frameworks*, depending on the scope of their applicability.

When developing an application it is necessary to consider reusing existing software, as well as making sure the newly developed software is easy to maintain and is itself reusable. Maintenance is in itself a form of software reuse.

The smallest unit of reuse in object-oriented software is an object or class. When a class is reused (e.g., refined by means of sub-classing) this is called **white-box reuse**. This is due to visibility: all attributes and methods are normally visible to sub-classes. This type of reuse is considered to be more complex for developers, because it requires an understanding of the implementation details of the existing software. When reuse is by means of object composition, and we are only concerned with the interfaces – how an object can be used — this is called **black-box reuse**, because the internal details of the object are not visible.

Black-box reuse has proved to be much more successful than white-box reuse. It is less complex for developers and does not interfere with the encapsulation of objects and is therefore safer to use.

**Toolkits** are a set of related and reusable classes designed to provide a general purpose functionality. Toolkits help with the development process without imposing too many restrictions on the design. The packages in Java such as `java.net`, `java.util`, and the `java.awt` are examples.

**Frameworks** represent reuse at a much higher level. Frameworks represent design reuse and are partially completed software systems intended for a specific family of applications. One example of a framework is the *Java Collections Framework*.

Patterns, in contrast, are not pieces of software at all. They are more abstract, intended to be used for many types of applications. A **pattern** is a small collection of objects or object classes that co-operate to achieve some desired goal. Each design pattern concentrates on some aspect of a problem and most systems may incorporate many different patterns.

## Pattern classifications and pattern catalogue

Design patterns are based on practical solutions that have been successfully implemented over and over again. Design patterns represent a means of transition from analysis/design to design/implementation.

To help developers to use design patterns, catalogues of patterns have been created. Each catalogue entry for a pattern should contain the following four essential elements:

- *The pattern name*, which identifies a commonly agreed meaning and represents part of the design vocabulary.
- *The problem or family of problems and conditions* to which it may be applied.
- *The solution*, which is a general description of participating classes/objects and interfaces their roles and collaborations.
- *The consequences* — each pattern highlights some aspect of the system and not others, so it is useful to be able to analyse benefits and restrictions.

Gamma et al classify design patterns into three categories according to purpose. The categories are *behavioural*, *creational* and *structural*. Unfortunately the catalogue of patterns is not standardised, which may cause some confusion. The level of granularity and abstraction differs greatly from objects whose only responsibility is to create other objects to those that create entire applications. There is no guarantee that a suitable pattern will always be found. It also may be that several different patterns could be used to solve a specific problem — in other words, a single pattern may not represent the only solution, but a possible solution.

**Table 8.1. Design patterns according to Gamma et. al.**

Behavioural	Creational	Structural
Interpreter	Factory Method	Adaptor (class)
Template Method	Abstract Factory	Adaptor (object)
Chain of Responsibility	Builder	Bridge
Command	Prototype	Composite
Iterator	Singleton	Decorator
Mediator		Facade
Memento		Flyweight
Observer		Proxy
State		
Strategy		
Visitor		

### The Portland Pattern Repository

A large collection of design patterns is available at the Portland Pattern Repository [<http://c2.com/ppr/>]. This repository is hosted by *Cunningham & Cunningham*, the consultancy firm

of Ward Cunningham, one of the Gang of Four. The website is also famous for being the web's first wiki.

## Behavioural patterns

Behavioural patterns are required when the operations that need to be performed cannot be achieved without co-operation. Thus behavioural patterns concentrate on the way in which classes and objects organise responsibilities in order to achieve the required interaction.

The *Observer* pattern is an example of a behavioural pattern that defines some dependency between objects.

### The *Observer* pattern

In some applications, two or more objects that are independent of each other must respond to some event in synchrony. For example any *Graphical User Interface* (GUI) will respond to the click of a mouse button, or keyboard, which will trigger the execution of an application or utility and redraw the screen appropriately. The mouse click event will result in one or more objects responding. Each object is otherwise independent. Each object is able to respond only to certain events.

Other typical examples of applications in which the *Observer* pattern could be used:

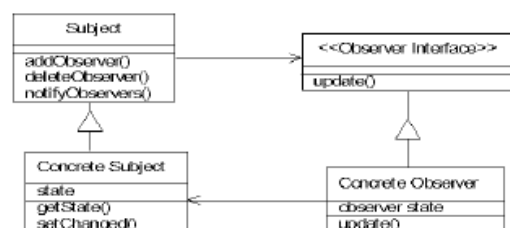
- To integrate tools in a programming environment. For instance, the editor for creating program code may register with the compiler for syntax errors. When the compiler encounters such an error, the editor will be informed and can scroll to the appropriate line of code.
- To ensure consistency constraints, such as referential integrity for database management systems.
- In the design of user interfaces to separate presentation of data from applications that manage the data. For example a spreadsheet object and a chart or a text report may all display the same information resulting from an application's data at the same time in their different forms.

### The problem

The important relationship that needs to be established is between a *subject* and an *observer*. The subject may be observed by any number of observers. The observers should be notified when the subject changes state. Each observer will receive information concerning the subject's state with which to synchronise, and to allow them to respond as required by the application. The following summarises the conditions:

- The subject is independent, and the observers are dependent.
- A change in the subject will trigger changes in observers — of which there may be many.
- The objects that will be notified by the subject are otherwise independent. They only share in some aspect of their behaviour.

**Figure 8.8. Class diagram of the *Observer* pattern**





## The solution

### Participating classes /objects:

In the diagram the subject is shown as a class. *Subject* has methods for attaching and detaching observer objects. The methods are shown on the diagram as `addObserver()`, `deleteObserver()` and `notifyObservers()`.

*Observer* has an updating interface for objects that will be notified of changes in a subject, here shown as an interface with the method `update()`.

*Concrete Subject*, a subclass of *Subject*, contains its state (of interest to the observers), plus operations for changing its state. *Concrete Subject* is able to notify its observers when its state changes. On the diagram the attribute state, and methods `getState()`, `setChanged()` provide this functionality, and the other methods are inherited from *Subject*.

*Concrete Observer* maintains a reference to the *Concrete Subject* object, and a state that needs to be kept consistent with the *Concrete Subject*. It implements the updating interface. On the diagram the *Concrete Observer* is a class that implements the *Observer* interface. It supplies the code for the `update()` method and has `observerState` to denote the data that needs to be kept consistent with the *Concrete Subject* object.

### Collaborations

The *Concrete Observers* objects register with the *Concrete Subject* object, using the `addObserver()` method.

When a *Concrete Subject* changes state it notifies the *Concrete Observer* objects by executing the `notifyObservers()` method.

The *Concrete Observer* object(s) obtain the information about the changed state of the *Concrete Subject* and execute the `update()` method.

### Consequences

The advantage of the pattern is that the *Subject* and *Observer* are independent of each other, and the subject does not need to know anything about the handling of the notification by the observers (i.e., how `update()` works). This means that any type of broadcasting communication could be implemented in this way.

## Creational patterns

Creational patterns handle the process of object creation. These patterns may be used to provide for more reusable designs by placing the emphasis on the interfaces and not the implementation. The abstract factory is an example of a creational pattern that can be used to make objects more adaptable, in other words:

- Less dependent on specific implementations.
- More amenable to change and customisation, easier to change the objects themselves.
- Less necessary to change the applications that use the objects.

### **Abstract factory pattern**

The abstract factory pattern makes the system independent of how objects are created, composed and represented. It should be used whenever the type or specific details of the actual objects that need to

be created cannot be predicted in advance, and therefore must be determined dynamically. Only the required behaviour of the objects is specified in advance. The information that can be used to create the object would be based on data passed at execution time. Examples of applications of the pattern:

- To customise Windows, Fonts, and so on, for the platform on which the application will run so as to ensure appropriate formatting, wherever the application is deployed.
- When the application specifies all the required operations on the objects it will use, but their actual format will be determined dynamically.
- To internationalise user interfaces (e.g., to display all the text in a local language, to customise the date format, to use local monetary symbols).

## The problem

The application should be independent of how its objects are created and represented. It should be possible to configure the application for different products/platforms. The application defines precisely how the objects are used (i.e., their interfaces).

## The solution

### Participating classes/objects

*Abstract Factory* class will contain the definition of the operations required to create the objects, `createProductA()`, `createProductB()`.

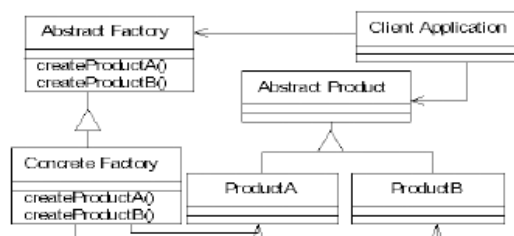
*Concrete Factory* implements the operations `createProductA()`, `createProductB()`. Only one *Concrete Factory* is created at run time which will be used to create the product objects

*AbstractProduct* will declare an interface for the type of product object for example a particular type of GUI object: Label or Window.

*ProductA* will define the object created by the *Concrete Factory 1*, implementing the *Abstract ProductA* interface.

*Client Application* uses only the interfaces from the *Abstract Factory* and *Abstract Product* classes.

**Figure 8.9. Class diagram for the *Abstract Factory* pattern**



### Consequences

Different product configurations can be used by replacing the *Concrete Factory* an application uses. This is a benefit and liability, because for each platform or family of products a new *Concrete Factory* subclass needs to be defined. However, the changes will be broadly restricted to the definition of subclasses of *Abstract Factory* and *Abstract Product*, thus confining the changes to the software to well documented locations.

## Structural patterns

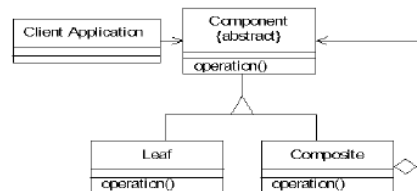
These patterns deal with the composition of complex objects. Similar functionality can be often achieved by using delegation and composition instead of inheritance. An example of a structural pattern is the composite pattern. In the Java API, this pattern is used to organise the GUI using AWT objects and layout managers.

### Composite pattern

The pattern involves the creation of complex objects from simple parts using inheritance and aggregation relationships to form treelike hierarchies.

The diagram shows the composite pattern as a recursive structure where a component can be either a leaf (which has no sub-components of its own) or a composite (which can have any number of child components). The component class defines a uniform interface through which clients can access and manipulate composite structures. In the diagram this is represented by the abstract method `operation()`.

**Figure 8.10. Class diagram of the *Composite* pattern**



### The problem

Complex objects need to be created, but the composition and its constituent parts should be treated uniformly.

### The solution

#### Participating classes/objects

*Component* should declare the interface for the objects in the composition, as well as interfaces for accessing and managing its child components.

*Leaf* represents objects that have no children, and defines behaviour for itself only.

*Composite* will define behaviour for components with children, and implements the child related interfaces.

*Client Application* manipulates objects through the component interface using the `operation()` method. If the object to be manipulated is a *Leaf* it will be handled directly, if it is a *Composite*, the request will be forwarded to the child

#### Consequences

The pattern enables uniform interaction with objects in a composite structure through the *Component* class.

Defines hierarchies consisting of simple objects and composite objects which can themselves be composed and so on.

Makes it easier to add or remove components.

## How to use a design pattern

- Consult design pattern catalogues for information (such as the Portland Pattern Repository, discussed earlier). You may find an example or description that may suggest the pattern is worth considering.
- Try to study the suggested solution in terms of participating objects/classes, conditions, and descriptions of the collaborations.
- If the examples of these patterns are part of a toolkit, it may be useful to examine the available information. `java.util` supports the *Observer* pattern, for example.
- Give participant objects names appropriate for your application context.
- Draw a class diagram showing the classes, their necessary relationships, operations and variables that are needed for the pattern to work.
- Modify the names for the operations and variables appropriately for your application.
- Try out the pattern by testing a skeleton example.
- If successful refine and implement it.
- Consider alternative solutions.

## Patterns in Java

Some design patterns generally recognised as common solutions to specific problems have been adopted as part of the Java JDK and Java API. A sample of these design patterns will be analysed here in greater detail.

### The *Observer* pattern in Java

In Java, the *Observer* pattern is embodied by the *Observer* interface and the *Observable* class which are part of the `java.util` package.

Any object that needs to send a notification to other objects should be sub-classed from class *Observable*, and any objects that need to receive such notifications must implement the interface *Observer*.

**Table 8.2.** *Observable* class methods in `java.util.package`

Observable Methods	Description
<code>addObserver(Observer o)</code>	Add the object passed as an argument to the internal record of observers. Only observer objects in the internal record will be notified when a change in the observable object occurs.
<code>deleteObserver(Observer o)</code>	Deletes the object passed as an argument from the internal record of observers.
<code>deleteObservers()</code>	Deletes all observers from the internal records of observers.
<code>notifyObservers(Object arg)</code>	Call the <code>update()</code> method for all the observer objects in the internal record if the current object has been set as changed. The current object is set

Observable Methods	Description
	as changed by calling the <code>setChanged()</code> method. The current object and the argument passed to the <code>notifyObservers()</code> method will be passed to the <code>update()</code> method for each <i>Observer</i> object.
<code>notifyObservers()</code>	Same but with null argument.
<code>countObservers()</code>	The count of the number of observer object for the current object returned as an int.
<code>setChanged()</code>	Sets the current object as changed. This method must be called before calling the <code>notifyObservers()</code> method.
<code>hasChanged()</code>	Returns true if the object has been set as “changed” and false otherwise.
<code>clearChanged()</code>	Reset the changed status of the current object to unchanged.

`addObserver()` method of the *Observable* class registers the *Observers*.

Each class that implements an *Observer* interface will have to have an `update()` method, and this method will ensure that the objects will respond to the notification of a change in the *Observable* object by executing the `update()` method:

```
public void update(Observable, Object)
```

The *Observable* object can indicate that it has changed, by invoking at any time `notifyObservers()`

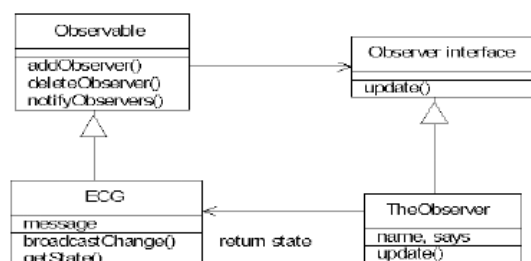
Each *Observer* is then passed the message `update()` where the first argument is the *Observable* that has changed and the second is an optional one provided by the notification.

## Example of *Observer* pattern using `java.util`

An electrocardiogram (ECG) monitor attached to a patient notifies four different devices:

- *Remote Display* for the physician, to allow them to adjust the configuration settings and treatment.
- *Chart Recorder* that will display the waveforms.
- *Remote Display* with a patient alarm.
- *Instruments Monitor* for the service personnel.

**Figure 8.11. The ECG *Observer***



Each of the devices will perform specific operations when the events for which they have registered an interest take place. The ECG attached to the patient will notify them of changes, as they occur.

The skeleton solution presented below will display messages stating the specific tasks performed by each *Observer*.

```
/*===== ECG.java
```

*Definition of class ECG subclassed from Observable from which Observable objects will be created. Demonstrates the use of methods setChanged() which will change the state of the ECG object, and notifyObservers() which will broadcast the event to the registered objects.*

```
=====*/
```

```
import java.util.*;

public class ECG extends Observable
{
    String message = "";

    public void broadcastChange()
    {
        message = "\tHeart Electrical Activity";
        setChanged();
        notifyObservers();
    }

    public String getState()
    {
        return message;
    }
}

//class
```

```
/*=====
```

*TheObserver.java*

*Definition of class TheObserver from which Observer objects will be created.*

*The class implements the Observer interface and thus must define the method update() which will be executed when the objects of class TheObserver are notified.*

```
=====*/
```

```
import java.util.*;
import java.io.*;

public class TheObserver implements Observer
{
    String name;
    String says;

    public TheObserver (String name, String says)
```

```

    {
        this.name = name;
        this.says = says;
    }

    public void update(Observable O, Object o)
    {
        System.out.println(( (ECG)O).getState()+
            "\n\t" + name + " : " + says+ "\n");
    }
}

//class

/*=====

PatternTest.java

Definition of program to test the Observer pattern

An object of class ECG called ecg is created, as well an array
called observers containing the four TheObserver objects.

These objects register with the observable object ecg using method
addObserver.

When the ecg method broadcastChange() is executed the observers
will be notified and update themselves.

=====*/

import java.util.*;

public class PatternTest {

    public static void main(String[] args) {
        ECG ecg = new ECG();
        TheObserver[] observers
        = { new TheObserver("Physician", "Adjust Configuration"),
          new TheObserver("Remote Display","Monitor Details"),
          new TheObserver("Chart Recorder", "Draw ECG WaveForm")
          new TheObserver("Service Personnel", "Monitor Instruments")};

        for (int i = 0; i < observers.length; i++)
            ecg.addObserver(observers[i]);

        ecg.broadcastChange();
    }
}

```

## The Observer pattern as part of the Java API

An example of the application of the *Observer* pattern of the Java API is the Java model of event handling using listeners. *Graphical User Interface* (GUI) components from the Java *Abstract Windowing Toolkit* (AWT) such as buttons, text fields, sliders, check boxes, and so on, are managed in this way. The observer objects implement a listener interface (e.g., the *ActionListener*, *WindowListener*, *KeyListener* etc.). When any of the components changes state, the listeners are

notified that a change has occurred. The listener decides what action should be taken as a result of the change. To tell a button how the events it generates should be responded to, the button's `addActionListener()` method is called, passing a reference to the desired listener. Every component in AWT has one `addXXXListener()` method for each event type that the component generates. When a button with an action listener is clicked, the listener receives an `actionPerformed()` call which will contain the instructions that need to be performed in response to the event. The `actionPerformed()` method must be defined as part of implementing the listener interface

## The *Model-View-Controller* pattern

This pattern is a specialised version of the *Observer* pattern, which was introduced in the *Smalltalk* language as a way of structuring GUI applications by separating the responsibilities of the objects in the system. The user interface consists of a *View*, which defines how the system should present this information to the user, and a *Controller*, which defines how the user interacts with the *Model*, which receives and processes input events. Systems analysis and design concentrates mainly on building the model representing the main classes of the application domain and the information of interest will be internally stored in object of the classes. The *Model-View-Controller* pattern makes it possible to associate the model with many different view/controller pairs in a non-intrusive manner without cluttering the application code. It is introduced here because it can be easily applied to the way in which Java applications using the `java.awt` can be structured. It provides a way for providing the application system model with a user interface. This is achieved by separating the responsibilities as follows:

The responsibilities of the *Model* are:

- To provide methods that enable it to be controlled.
- To provide a method or methods to update itself, and if it is a graphical object, to display itself.

The *Controller* carries out the following series of actions:

- The user causes an event such as clicking the mouse over a button.
- The event is detected.
- A method to handle the event is invoked.
- A message will be sent to the model.
- The update method within the model will change the data within the model.
- A message will be sent to the view update the user interface, to, for example, redraw the image.

The *View* performs the following:

- Initially displays the model when the window is created, and every time it is resized.
- When the event handler detects a change (e.g., responds to a click of a button) it sends a message to redraw the screen.
- The *View* enables the updated information from the model to be displayed.

## **Abstract factory facilities in Java**

To make the creational process more versatile, object-oriented language facilities that provide for customisation should be used. The JDK has facilities to customise graphics and to internationalise programs and applications.

## **Graphics related platform characteristics**

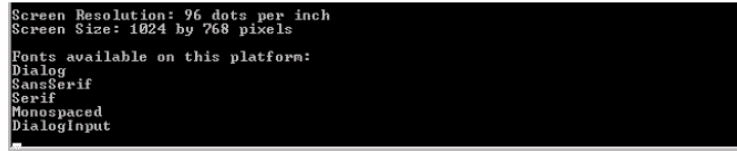
One of the problems of having a portable *Abstract Windowing Toolkit* is that the appearance and positioning of the objects may differ from one platform to another. To ensure that the graphical display is similar wherever the application may be ported, we should be able to adjust the *View* accordingly.



The *Toolkit* and *FontMetrics* classes may be used to help create a *Concrete Factory* that will create the *Concrete Product*:

**Figure 8.12. Java output of screen and font information**

Sysinfo.java output from my machine



```
Screen Resolution: 96 dots per inch
Screen Size: 1024 by 768 pixels

Fonts available on this platform:
Dialog
SansSerif
Serif
Monospaced
DialogInput
```

The *Toolkit* class provides information regarding screen resolution, pixels, available fonts, and *FontMetrics* can help supply information concerning font measurements for every AWT component. Below is a program listing showing how some screen and font information can be obtained. Figure 8.12, “Java output of screen and font information”, shows an example of some output obtained from the program.

```
/*=====

Sysinfo.java

program for finding out details of the display using the
java.awt Toolkit class

The program creates a Toolkit class object theKit, and then uses
the Toolkit methods:

getDefaultToolkit, getScreenResolution, getScreenSize and getFontList.

=====*/

import java.awt.*;

public class Sysinfo
{
    public static void main(String[] args)
    {
        Toolkit theKit = Toolkit.getDefaultToolkit();
        System.out.println("\nScreen Resolution: " +
            theKit.getScreenResolution() + " dots per inch");
        Dimension screenDim = theKit.getScreenSize();

        System.out.println("Screen Size: " + screenDim.width
            + " by " + screenDim.height + " pixels ");

        String myFonts[] = theKit.getFontList();

        System.out.println("\nFonts available on this platform: ");

        for (int i = 0; i < myFonts.length; i++)
            System.out.println(myFonts[i]);

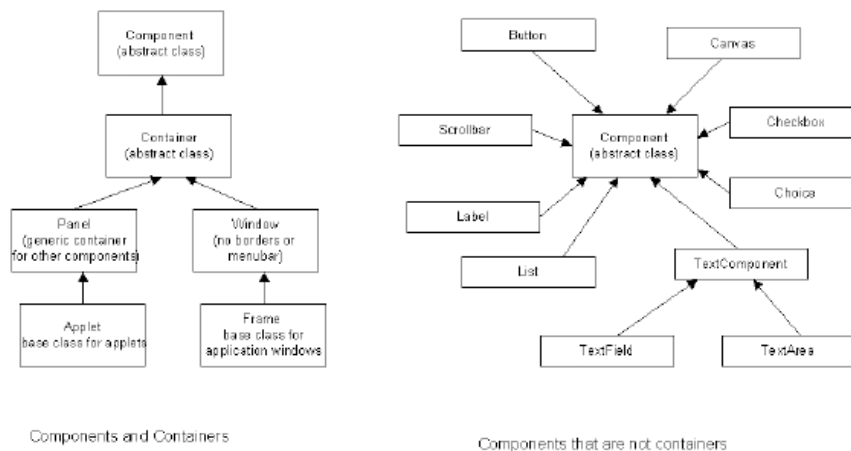
        return;
    }
}
```

Java facilities for internationalisation of applications (displaying all user visible text in the local language, using local customs regarding dates, monetary displays etc.) are available from `java.util` package using the *ResourceBundle* class and its subclasses.

## Composite patterns in Java

HCI classes for creating GUI applications are used in almost every interactive application. Much of these GUI classes adhere to some form of *Composite* pattern, and the Java AWT is no exception. Here, containers are components which can hold other components. Each component knows how to draw itself on the screen; containers, however, will defer some of their drawing functionality to the components that they contain.

**Figure 8.13. java.awt GUI components containers and layout managers**



Layout Managers	Description
FlowLayout	Places components in successive rows in a container, fitting as many on each row as possible, and starting on the next row as soon as a row is full. This works in much the same way as a text processor placing words on a line. Its primary use is for arranging buttons, although it can be used with components. It is the default layout manger for <i>Panel</i> and <i>Applet</i> objects
BorderLayout	Places components against any of the four borders of the borders of the container and in the centre. The component in the centre fills the available space. This layout manger is the default for objects of the <i>Window</i> , <i>Frame</i> , <i>Dialog</i> , and <i>FileDialog</i> classes
CardLayout	Places components in a container, one on top of the other – like a deck of cards. Only the “top” component is visible at any one time.
GridLayout	Places components in the container in a rectangular grid with the number of rows and columns that you specify.
GridBagLayout	This places the components into an arrangement of rows and columns, but the rows and columns can vary in length. This is a complicated layout

Layout Managers	Description
	manager with a lot of flexibility for controlling where components are placed in a container.

In Java, these are embodied by the AWT *Component* and *Container* class, and the layout manager classes. A *Window* can be divided into *Panels*, and each *Panel* can be treated as an individual component within another layout at a higher level.

## Review

### Questions

#### Review Question 1

How many of the different methods of managing heavy loads have been used in constructing the Firth of Forth Bridge?

A discussion of this question can be found at the end of this chapter.

#### Review Question 2

What is the principal difference between the job of a software engineer and those of architects and civil engineers?

A discussion of this question can be found at the end of this chapter.

#### Review Question 3

Explain the role of design patterns in object-oriented software development.

A discussion of this question can be found at the end of this chapter.

#### Review Question 4

Place each of the following patterns in the category it belongs to according to “the gang of four”:

Patterns: Observer, Model-View-Controlled, Abstract Factory, Composite:

Category: Creational, Structural, Behavioural.

A discussion of this question can be found at the end of this chapter.

#### Review Question 5

What are the four essential elements of a design pattern catalogue entry?

A discussion of this question can be found at the end of this chapter.

#### Review Question 6

What is meant by granularity?

A discussion of this question can be found at the end of this chapter.

#### Review Question 7

Give examples of white-box reuse and black-box reuse from the pattern examples.

A discussion of this question can be found at the end of this chapter.

## Review Question 8

Compile ECG.java, TheObserver.java and TestPattern.java, then execute the TestPattern program.

1. Create a new directory in which to store the files. You may call the directory Activity1.
2. Copy all three files ECG.java, TheObservers.java and TestPattern.java
3. Compile the three files using the JDK command javac
4. Once the compilation is successful you should be able to execute the TestPattern application using the command java Testpattern

Does the program output the messages in the order you have expected?

A discussion of this question can be found at the end of this chapter.

## Review Question 9

Design a solution using the Observer pattern for the operation of dispensing cash by an ATM machine. When a bank customer withdraws money from an ATM (Automatic Teller Machine), before the cash is dispensed it is necessary to determine whether there are sufficient funds. If there are, then it is necessary to instruct the machine to dispense the cash, to debit the customer's account and to log the transaction for auditing purposes.

How would you use the Observer pattern to design a solution to this problem?

A discussion of this question can be found at the end of this chapter.

## Review Question 10

Draw a diagram to represent the design pattern of a solution to the ATM operation of dispensing cash using the Observer pattern. You may use a CASE tool for the class diagram and include the outline of the methods, data and messages required to make the pattern work. It would be useful if you put the project in which the class diagram will be placed into a new directory.

A discussion of this question can be found at the end of this chapter.

## Review Question 11

The code you are expected to write will be a skeleton for the Observer pattern with messages instead of complex code to implement the operations. Compile the three parts of the program in the correct order. The Observable class, then the Observer and last the program. You may use any version of Java for this exercise. Follow the instructions similar to the ones given for Activity 1. It would be useful to place all of your files for this exercise in a new directory.

A discussion of this question can be found at the end of this chapter.

## Answers

### Discussion of Review Question 1

The Firth of Forth bridge uses three patterns which can be directly seen. These are:

- girders with cross-members,
- arches in a catenary shape,
- decreasing the span.

In addition some suspension support is provided 'from above' but this is not strictly a catenary shape.

## Discussion of Review Question 2

The main difference is that software engineers do not have the opportunity to see their and other people's designs implemented visually.

## Discussion of Review Question 3

Your answer is expected to include some of the following:

Serve as exemplars to programmers, designers and architects, which they can quickly adapt for use in their projects.

Emphasise solutions: discovering patterns that have been used before rather than inventing them.

Represent codified, distilled wisdom: solutions to recurring problems, if those solutions have well understood properties.

Allow programmers and designers to program and design using bigger chunks; this also eases those aspects that involve understanding an architecture; architectural reviews. Reverse engineering, maintenance and system restructuring.

Aid in communicating among designers, between designers and programmers, and between a project's team members and its non-technical members.

Identify and name abstract, common themes in object-oriented design, themes that have known qualify properties.

Form a documented, reusable base of experience, which would otherwise be learnt only through an informal oral tradition or through trial and error.

Provide a target for reorganisation of software because a designer can attempt to map parts of an existing system to a set of patterns. If this mapping can be done, the complexity of the resulting reorganised system will be less than the original version.

## Discussion of Review Question 4

Behavioural: Observer, M-V-C

Creational: Abstract Factory

Structural: Proxy, Composite

## Discussion of Review Question 5

- The pattern name which identifies a commonly agreed meaning and represents part of the design vocabulary.
- The problem or family of problems and conditions in which it may be applied.
- The solution which is a general description of participating classes/objects and interfaces their roles and collaborations.
- The consequences - each pattern highlights some aspect of the system, and not others so it is useful to be able to analyse benefits and restrictions.

## Discussion of Review Question 6

Granularity usually refers to the size of the components you deal with. In this context it could be from patterns that specify how a single object may be created, to patterns that will specify the structure of a whole application.

## Discussion of Review Question 7

For example, in the Observer pattern:

Concrete Observer is an example of black-box reuse because all the Concrete observer needs to do is to implement the `update()` method.

The Concrete Subject is an example to white-box reuse, because it needs to know the details of its super class `Observable`.

The Concrete Factory of the Abstract Factory pattern is an example of white box reuse. The way in which the Client Application uses the Abstract Factory pattern is an example of black box reuse. Etc.

## Discussion of Review Question 8

Compare this example with the Observer catalogue entry description, and follow the instructions on how to use a design pattern before going on to do the next exercise.

## Discussion of Review Question 9

This exercise will help you test your understanding of the Observer pattern. You are expected to base your answer on the given examples – to literally use the pattern. Using the CASE tool will help you produce the documentation of your design in the standard format.

## Discussion of Review Question 10

The purpose of this exercise is to use the `Observer` class and `Observable` interface from `java.util` and to test some of the methods from the lecture notes. This will consolidate your understanding of the use of this design pattern.

## Discussion of Review Question 11

Writing portable software, and implementing dynamic binding of objects in a distributed environment for which Java was designed requires developers to be aware of the different platforms, and be able to make sure applications take advantage of the facilities in the language that make it possible to port.