```
                Phone: 33445 44-55
Entry number 2 Name: T. Fox
                Address: 123 Houndsditch
                Phone: 0787 909 112
Entry number 3 Name: A. Widdecombe
                Address: Smith Square
                Phone:
```

where database already contained the entry for *A Jennings*.

The *DataItem* class is still used to transfer query results from database interface to GUI, but *Database* class has been entirely replaced by *AddressBookDb*.

Part 4: Design Patterns

Design patterns are standard design structures which have been found to be useful in many different applications. They provide a set of ready-made solutions to design problems which are encountered in software development, and a repertoire of building blocks for software design.

Design patterns are an important technique for organising structure of components (modules or subsystems) in a clear and systematic fashion.

Three main categories: *Creational* patterns address object construction; *Structural* patterns focus on static structure of inheritance and clientship, *Dynamic* patterns deal with object behaviour.

Classification of Design Patterns

		Purpose		
		Creational	**Structural**	**Behavioural**
Scope	**Class**	**Factory Method**	**Adapter (Class)**	**Template Method** **Interpreter**
	Object	**Abstract Factory** **Singleton** **Builder** **Prototype**	**Adapter (Object)** **Facade** **Proxy** **Bridge** **Composite** **Decorator** **Flyweight**	**State** **Observer** **Iterator** **Visitor** **Mediator** **Chain of Responsibility** **Command** **Strategy**

Patterns are means of organising structure of module or subsystem, can be distinguished from related concepts such as *frameworks* by scale of their application:

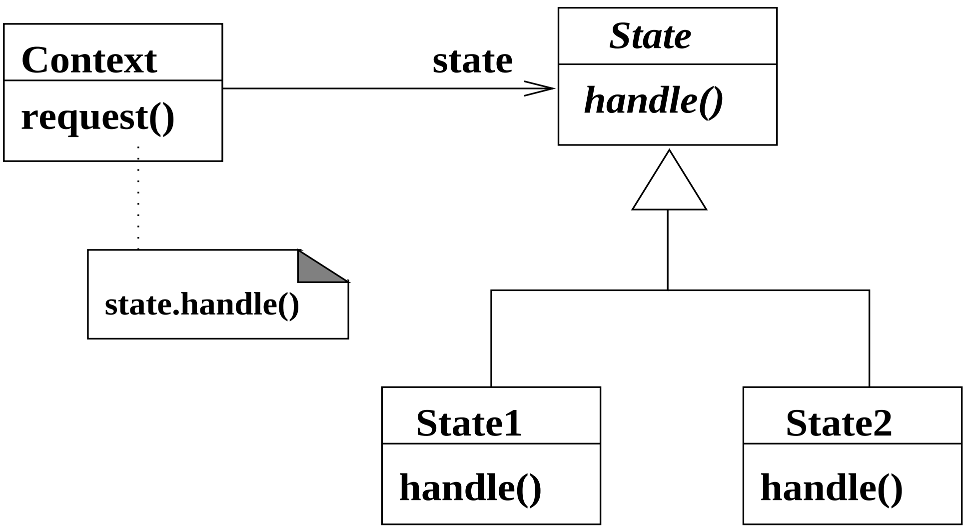
- **Frameworks**: template architectures used to organise complete application, such as J2EE for internet applications.
- **Design patterns**: apply at medium scale, define collaborating group of classes, objects and methods as part of application. Many variations are possible on how a pattern is applied, and they are mainly independent of particular programming languages.
- **Idioms**: characteristic way of using constructs of specific programming language. Eg, using *public static final int* constants in Java to define enumerated types. Unlike patterns, idioms are local in scope, apply at statement level, and limited in variation.

The State Pattern

Objects often have internal modes or states, with different behaviour – responses to messages – in each mode. For example a lift will respond to a request to move to a floor differently depending on whether it is idle or busy, and on whether it is at requested floor or not.

The state pattern introduces explicit subclasses of a class, to represent each different mode that objects of class may be in. In this pattern, choice between different responses to methods is handled by polymorphism of *State* subclasses, not by programmer (so program decisions are not hard-coded in one class, but depend on what subclass an object belongs to at runtime: this is potentially more efficient and maintainable).

General Structure of State Pattern



Structure of State Pattern

Each *Context* has a reference to some *State* object, which may be a member of either the *State*1 or *State*2 subclass at particular points in time.

State could be introduced to replace some original pseudocode:

```
public class Context
{
  public static final int MODE1 = 0;
  public static final int MODE2 = 1;
  private int mode;
  ....

  public void request()
  {
      if (mode == MODE1)
      { Code1 }
```



```
      else if (mode == MODE2)
      { Code2 }
  }
   ....
} // Context
```

Problem with this version is it is difficult to maintain, since any change to set of modes in system will require change to every method that depends on mode.

Applying state pattern gives:

```
public class Context1
{
  private State state;
  ...
  public void request() { state.handle(); }

  ...
```

```
} // Context1

abstract class State
{
  ...
  public abstract void handle();
} // State
```


There are two subclasses of *State*:

```
class State1 extends State
{
  public void handle() { Code1 }
} // State1

class State2 extends State
{
  public void handle() { Code2 }
} // State2
```

Now a change in behaviour of *State* objects in mode n only requires modification of local class *Staten*, not *Context*1. Introducing a new mode only involves creating a new subclass of *State* and possibly modifying *Context*1, but does not affect any other subclasses of *State*.

State pattern should be used when:

- An objects behaviour depends on its mode, and it must change its behaviour at run-time depending on that mode.
- Operations have large multipart conditional statements depending on mode.

The involved classes are:

- *Context*1 – this defines interface for clients; and maintains a *State* instance, corresponding to original *mode* variable.
- *State* – this is abstract superclass of classes representing particular modes.
- *State*1, etc – each subclass of *State* implements behaviour of a particular mode.

The benefits of pattern are that:

- The state pattern puts each mode of behaviour into a separate class, *localising* this behaviour. New states and transitions can be added by defining new *State* subclasses.
- It reduces size and complexity of operations, enhancing code understanding and maintainability.
- It makes state transitions more explicit (they are represented by changes in class of *state* object instead of by assignments of constants to *mode*).
- *State* objects can be shared between contexts if they have no instance variables of their own.

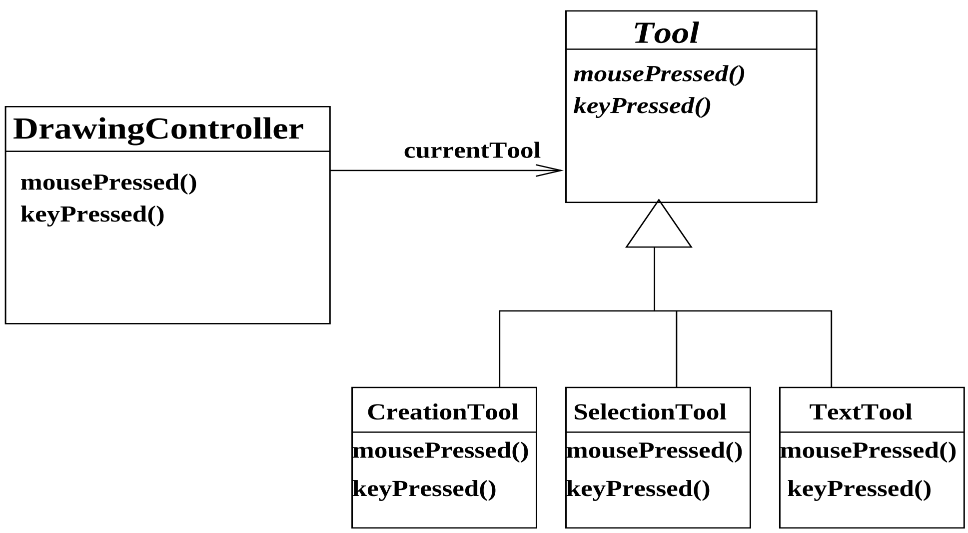
State Pattern Example: Drawing Tool

Common use of state pattern is to organise applications such as drawing tools which have different behaviour in different modes (eg, modes of a drawing tool could be shape creation, text editing or shape selection) but which respond to same basic sets of events (mouse events). In this example *Context* is a *DrawingController* responding to *mousePressed*(), *keyPressed*(), etc.

Role of *State* is played by *Tool* class: the drawing controller has a current drawing tool.

Subclasses of *Tool* are *CreationTool*, *DeletionTool*, *SelectionTool* etc, representing behaviour of drawing tool in corresponding mode: each subclass must handle *mousePressed*(), etc in different ways. For example, pressing mouse in creation tool mode indicates that shape should be created at press location; in selection tool mode it selects shape under this location.

State Pattern Applied to Drawing Package

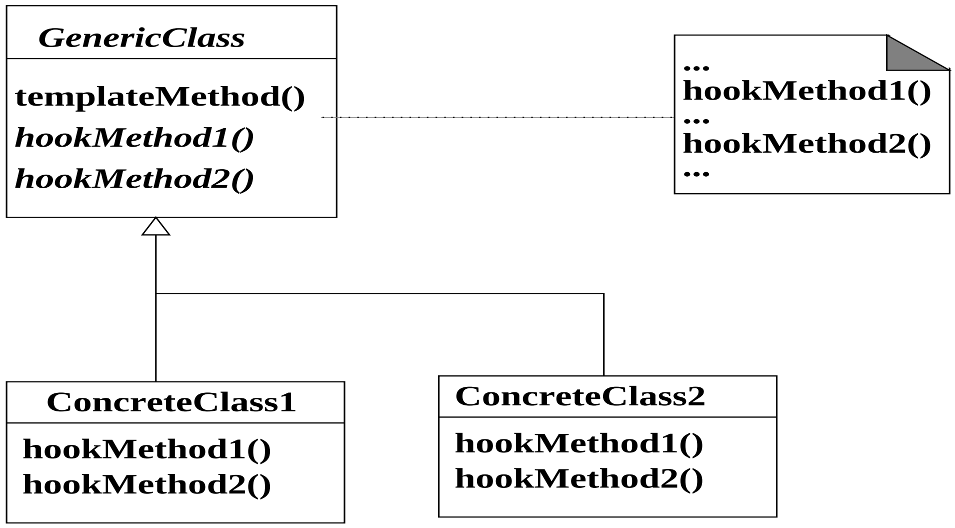


The Template Method Pattern

This pattern defines skeleton of an algorithm in a superclass method, deferring some details (defined only as abstract *hook* methods in superclass) to subclasses, allowing subclasses to redefine certain steps/parts of algorithm whilst leaving other parts unchanged.

This is used to factor out common parts/structure of an algorithm into superclass, so that subclasses only need to implement the parts of algorithm that differ between subclasses, thus avoiding code duplication.

Structure of Template Method Pattern



The classes involved in this pattern are:

- *GenericClass* – this defines template method which consists of a skeleton algorithm calling one or more hook methods. Also defines hook methods that subclasses can override.
- *ConcreteClass* – implements hook methods.

Hook methods may be defined as abstract in *GenericClass*, or given default implementations there.

Template Method differs from State pattern in that superclass is used to define common algorithms, the template methods, and subclasses only redefine parts of these algorithms, the hook methods. In state pattern, superclass declares common set of methods which system may respond to in any mode, all subclasses usually redefine each of these methods.

Template Method Pattern Example: Student Pass Marks

Problem in this example is:

> A student database has to manage two kinds of student, MSc and BSc students, which have common attributes *int studentId* and *int numberPassed*: latter attribute records number of courses passed by student so far. There is common method *addMark*(*int m*) which inputs mark given for a course.
>
> For MSc students *addMark*(m) increments number of courses passed, *numberPassed*, only if $m \geq 50$, the pass mark for MSc students. Similarly for BSc students, except that pass mark is 40 in this case.

Template Method can be applied, with *addMark* defined in a generic way using a hook method *isPass*, which has different definitions in the two subclasses.

The resulting classes are:

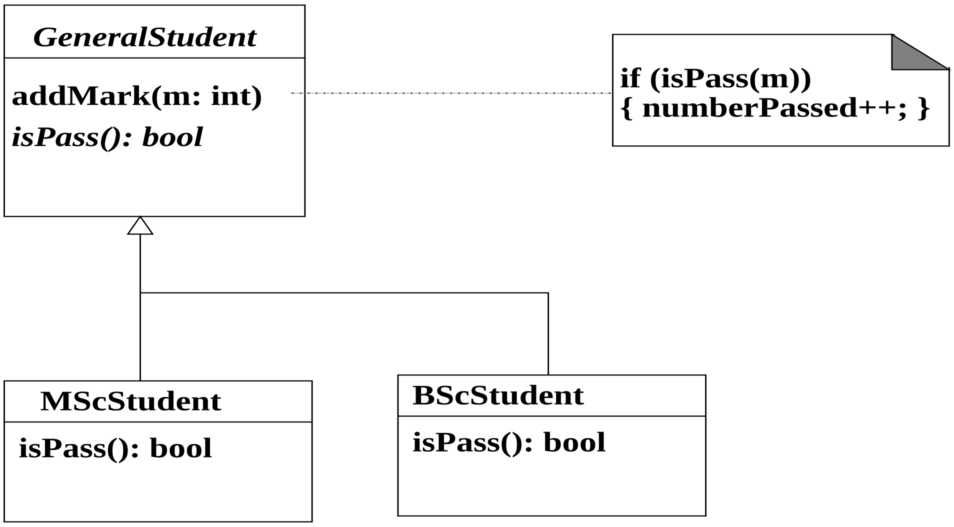
```
abstract class GeneralStudent
{ private int studentId = 0;
  private int numberPassed = 0;

  public GeneralStudent(int id)
  { studentId = id; }

  abstract protected boolean isPass(int mark);  /* Hook method */

  public void addMark(int m)           /* Template method */
  { if (isPass(m))
    { numberPassed++; }
  }
}

class MScStudent extends GeneralStudent
```



```
{ public MScStudent(int id)
  { super(id); }

  protected boolean isPass(int m)
  { return (m >= 50); }
}

class BScStudent extends GeneralStudent
{ public BScStudent(int id)
  { super(id); }

  protected boolean isPass(int m)
  { return (m >= 40); }
}
```

Notice that *MScStudent* and *BScStudent* *only* redefine hook method *isPass*, since the template method *addMark* is unchanged from superclass definition.

Template Method Example



If pattern was not used, would have to define *addMark* in both subclasses:

```
public void addMark(int m)
{ if (m >= 50)
  { numberPassed++; }
}
```

in *MScStudent* and

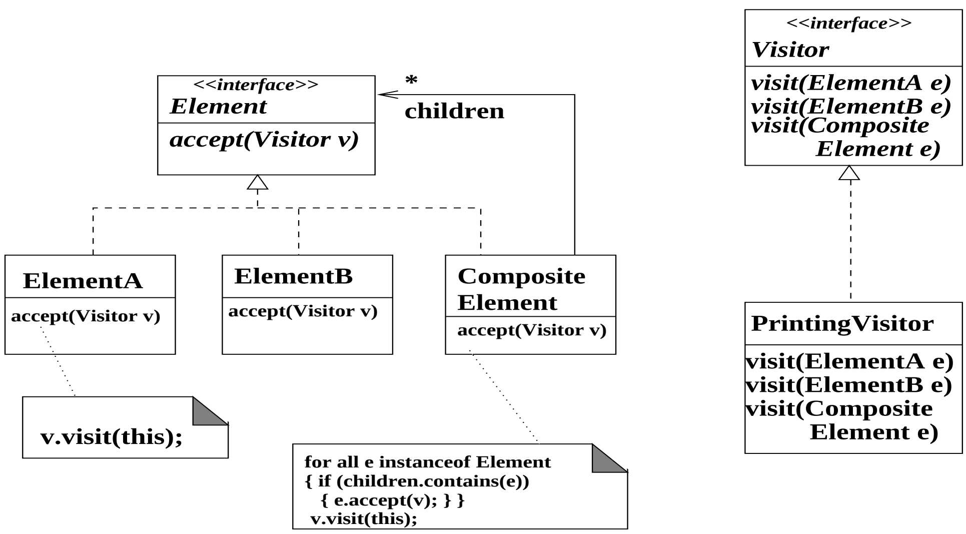
```
public void addMark(int m)
{ if (m >= 40)
  { numberPassed++; }
}
```

in *BScStudent*, even though definitions are almost identical. *numberPassed* would also have to have *protected* visibility instead of *private*.

The Visitor Pattern – defines algorithms over common data structure. The involved classes are:

- *Element* – superclass of all classes of objects found in structure to be iterated over. It has an abstract *accept* method to process general visitors.
- *ConcreteElement* – specific classes of elements in structure, such as *ElementA* and *CompositeElement*. Each class defines *accept* for *Visitor* objects, specifying in what order visitor should traverse any subelements and element itself.
- *Visitor* – defines abstract *visit*(*ConcreteElement ce*) methods for each concrete element class.
- *ConcreteVisitor* – encapsulates particular algorithm for data structure, defines *visit* methods giving actions to be taken when visiting each type of element in structure.

Example Structure of Visitor Pattern



Visitor Pattern Example: Parse-Tree Processing

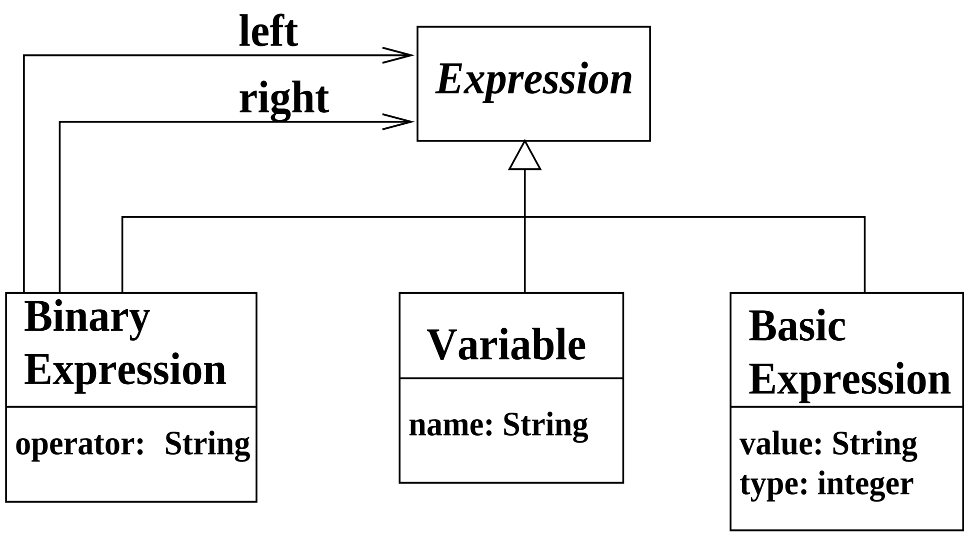
A parse tree for a simple programming language is represented in the classes:

1. *BasicExpression*s, representing numbers or string values. These have a *String value* attribute to hold value, and *int type* to hold type (0 for numbers, 1 for strings).

2. *Variable*s, representing variables, with a single *String name* attribute to hold variable name.

3. *BinaryExpression*s, representing expressions $e = f$, $e < f$, $e > f$, etc. Operator is held in *String operator* attribute, and arguments in *Expression left* and *Expression right* attributes.

Classes for Parse Tree Representation



Visitor Example: Parse-tree Operations

Several functions are to be implemented to extract properties of and locally modify elements of such trees:

- An operation to compute list of all variable names (without duplicates) used in a parse tree of above form.
- An operation to perform a simultaneous renaming of all variables in a parse tree of above form, ie, given a *Map replacement* from *String* to *String*, if s is name of variable v, given by $v.getName()$, and s occurs in domain of *replacement*, execute $v.setName((String)\ \ replacement.get(s))$.

A suitable visitor framework over parse trees can be defined as:

```
abstract class Visitor
{ public abstract void visit(Variable v);

  public abstract void visit(BasicExpression be);

  public abstract void visit(BinaryExpression be);
}

abstract class Expression
{ public abstract void accept(Visitor v); }

class BinaryExpression extends Expression
{ private Expression left;
  private Expression right;
  private String operator;
```

```
  public void accept(Visitor v) // Visit left subtree,
  { left.accept(v);             // then right, then
    right.accept(v);            // myself.
    v.visit(this);
  }
  ...
}

class BasicExpression extends Expression
{ private String value;
  private int type;

  public void accept(Visitor v)
  { v.visit(this); }
}

class Variable extends Expression
{ private String name;
```



```
  public void setName(String nme)
  { name = nme; }

  public String getName()
  { return name; }

  public void accept(Visitor v)
  { v.visit(this); }
}
```

This definition specifies a post-order traversal over expression tree structures, ie, visitor processing is carried out on left argument of a binary expression first, then on right argument, and then on expression itself.

First function can be achieved by a *GetUsesVisitor*:

```
import java.util.Vector;

public class GetUsesVisitor extends Visitor
{ private Vector vars = new Vector();

  public void visit(Variable v)
  { if (vars.contains(v.getName())) {}
    else
    { vars.add(v.getName()); }
  }

  public void visit(BasicExpression be) {}
```



```
  public void visit(BinaryExpression be) {}
}
```

This accumulates all variable names in a list as it traverses expression tree, carrying out a skip action when visiting any other kind of expression.

The renaming operation can be achieved by a *RenamesVisitor*:

```
import java.util.Map;
import java.util.HashMap;

public class RenamesVisitor extends Visitor
{ private Map replacements = new HashMap();

  public void setReplacement(String old, String rep)
  { replacements.put(old,rep); }

  public void visit(Variable v)
```

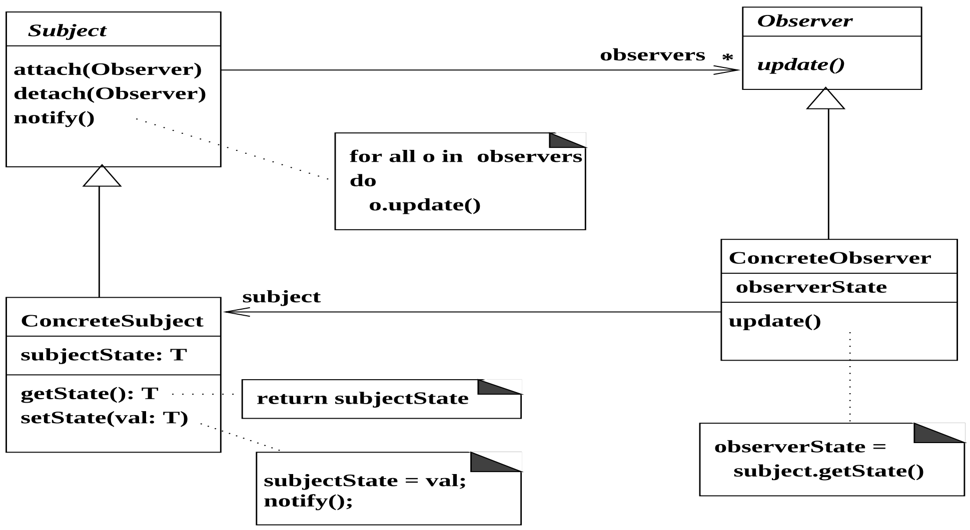
```
  { String s = (String) replacements.get(v.getName());
    if (s != null)
    { v.setName(s); }
  }

  public void visit(BasicExpression be) {}

  public void visit(BinaryExpression be) {}
}
```


The Observer Pattern

This pattern is intended for situations where there are multiple views or presentations of a set of data, such as alternative graphical views (eg, pie charts and bar charts of sales figures).

In general it defines a one-to-many dependency between objects so that when one object (the data) changes state, all its dependants (the views) are notified and update themselves automatically.

Structure of Observer Pattern



The classes in this structure have roles:

- *Subject* – abstract superclass of all classes containing observed data. Provides methods *attach* and *detach* to add and remove observers of a subject, and *notify* to inform all observers that a significant state change has occurred in the observable and that they may need to update their presentation of it.
- *ConcreteSubject* – specific data, any updater method of this class may need to call *notify* on completion.
- *Observer* – abstract superclass of observers of subjects. Declares an *update* method whose function is to adjust observer's presentation on any subject state change.
- *ConcreteObserver* – class defining a specific view, such as a bar chart.

Positive consequences of using the pattern are:

- Modularity: subject and observers may vary independently.
- Extensibility: we can define and add any number of observers to a given subject.
- Customisability: different observers provide different views of a subject.

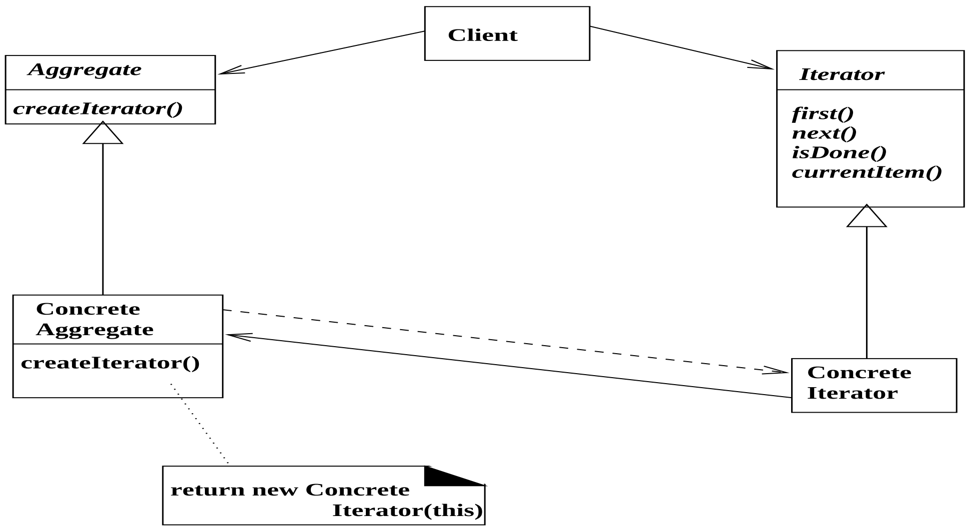
This pattern is widely used in commercial languages and libraries, for example, in Smalltalk Model-View-Controller (MVC) paradigm. In Java the Swing event model is directly related to Observer, with *Listener* objects playing role of *Observer* for event source (*Subject*) objects. Java Beans and Jini also use a similar event notification model.

The Iterator Pattern

Purpose of this behavioural pattern is to support access to elements of an aggregate data structure (such as a tree or array) sequentially. It is applicable for:

- multiple traversal algorithms over an aggregate;
- a uniform traversal interface over different aggregates;
- or when aggregate classes and traversal algorithm must vary independently.

Structure of Iterator Pattern



An iterator object acts like a cursor or pointer into a structure, indicating a current location within this structure and providing operations to move this cursor forwards or backwards in structure.

Classes involved are:

- *Aggregate* – class defining a general composite data structure such as a list or tree.
- *ConcreteAggregate* – specific subclass defining particular data structure such as a linked list or binary search tree.
- *Iterator* – interface for general iteration operations such as accessing first element in collection, stepping through collection, etc.
- *ConcreteIterator* – iterator subclass specific to particular data structure. *createIterator*() method of data structure returns a *ConcreteIterator* instance for structure.

Possible to have multiple iterators acting on same aggregate object simultaneously, possibly with different traversal algorithms.

If iterator pattern was not used, would require direct access to the private parts of data structures being iterated over.

Some examples of iterators in Java are:

- Simple integer indexing into an array:

```
for (int i = 0; i < a.length; ++i)
{ x += a[i]; }
```

- Java Collections iteration over an *ArrayList*:

```
for (Iterator i = a.iterator(); i.hasNext(); )
{ x += ((Integer) i.next()).intValue(); }
```

- Java JGL iteration over an *Array*:

```
for (ForwardIterator i = a.start(); !i.atEnd(); i.advance())
{ x += ((Integer) i.get()).intValue(); }
```


Iterators in Java

Java Collections library of data structures, distributed with JDK, has *Iterator* interface with operations *hasNext*() and *next*(). The first returns *false* once iterator has reached end of its iteration, second returns current element that iterator points to, and moves forward one position. Each container class has a method *iterator*() which returns an *Iterator* object positioned at start of container data.

In JGL library (http://www.objectspace.com) in contrast there are several kinds of iterator:

- *InputIterator*: reads items in forward direction.
- *OutputIterator*: writes items in forward direction.
- *ForwardIterator*: reads/writes in forward direction.
- *BidirectionalIterator*: a *ForwardIterator* that can also

read/write items in backwards direction.

- *RandomAccessIterator*: a *BidirectionalIterator* that supports relative position comparisons within the container (between two *RandomAccessIterator* objects).

In JGL creation of an iterator for a container is carried out by *start* method of container class. The test for end of an iteration is the *atEnd*() method (inverse of *hasNext*()). Access to current element is by the *get*() method, which is separate to movement operations *advance*(), *retreat*(), etc.

Typical implementation of iterators in Java uses concrete iterator classes defined as inner classes of container class they are for, which implement *Iterator* interface of Collections:

```
import java.util.Iterator;

public class MyLinkedList
{ protected Node head;

  protected class Node
  { Object element;
    Node next;
    .... }

  public Iterator iterator()
  { return new LinkedListIterator(); }

  private class LinkedListIterator implements Iterator
```

```
  { private MyLinkedList.Node cur;

    LinkedListIterator() { cur = head; }

    public boolean hasNext()
    { return cur != null; }

    public Object next()
    { Object obj = null;
      if (cur != null)
      { obj = cur.element;
        cur = cur.next;
      }
      return obj;
    }

    public void remove()
    { throw new UnsupportedOperationException(); }
```



```
  }
}
```

Iterator and visitor patterns are related – both provide frameworks for navigating through data structures.

Iterator provides mechanisms for accessing individual objects of a collection and navigating over structure. Does not define what processing is performed on elements, this is outside the pattern.

With Visitor, navigation strategy is built-in to collection classes and cannot be varied: all visitor algorithms for structure must use this strategy. With Iterator, navigation strategy is defined within different iterator classes, can be several of these for same structure.

With Visitor what may vary is function applied to each collection element, with Iterator what may vary is navigation strategy to traverse a structure and access its elements.

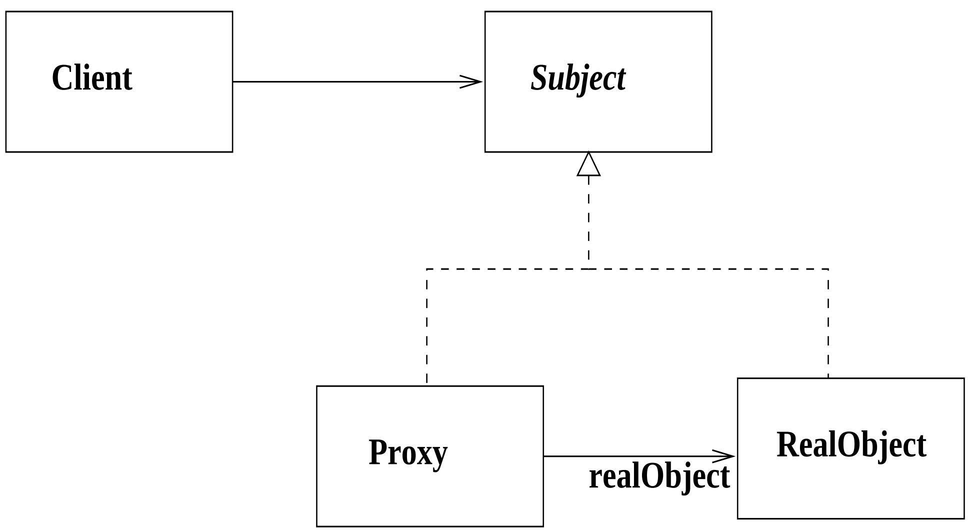
The Proxy Pattern

This provides surrogate or placeholder ('proxy' object) that represents another ('real') object. It is applied when there is a need for a more versatile or sophisticated interface to an object than a simple direct reference.

Examples include:

- *remote proxy*: eg, stub in Java RMI, providing local representative for a server object residing on remote host;
- *virtual proxy*: creating space/time consuming objects only on demand (eg, database connections);
- *protection proxy*: controlling access to real object to provide different levels of access rights.

Structure of Proxy Design Pattern



Classes involved in pattern are:

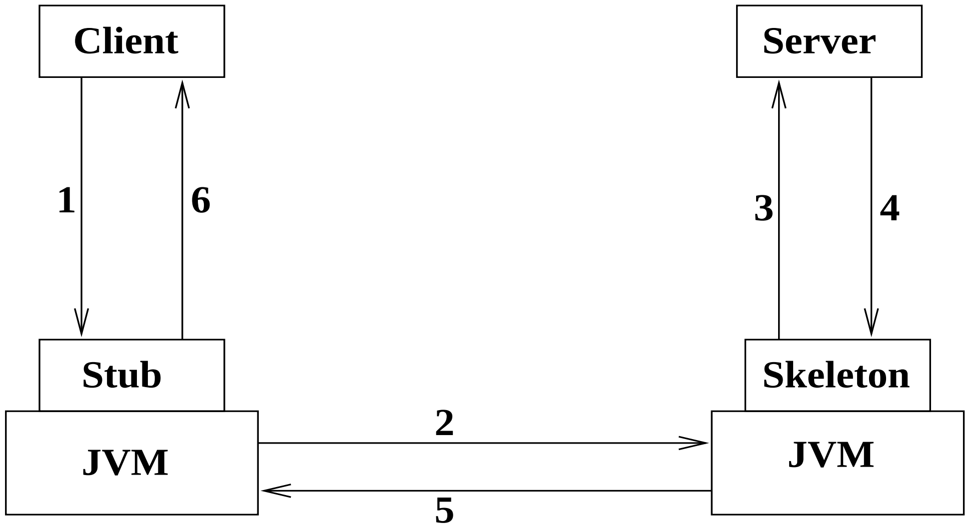
- *Proxy*: implements *Subject* interface, so that proxy can be substituted for real object. It has a reference to real object, or other means for forwarding commands to it.
- *Subject*: defines common interface of *Proxy* and *RealObject* so that a *Proxy* can be used anywhere that *RealObject* is expected.
- *RealObject*: defines real object that proxy represents.

Proxy Pattern Example: Java RMI

RMI (remote method invocation) is mechanism for distributed processing allowing method invocations to be made on remote objects (ie, objects running on different host machine to caller). It involves:

- *Server*: an object providing services to other objects, possibly on different hosts.
- *Service contract*: an interface declaring services provided by server objects.
- *Client*: an object using services of remote servers.
- *Stub*: object on same host as client (ie: a *local* object), serving as proxy for remote server object.
- *Skeleton*: object on same host as server, which receives requests from stubs and dispatches requests to server.

Steps of RMI



These steps are:

1. a remote invocation $server.m(p)$ by client is carried out as invocation of a method of stub: $stub.m(p)$;
2. Stub marshals parameter values p (ie, puts them into a form that can be transmitted over network), then sends values and call information to skeleton on server host;
3. Skeleton unmarshals arguments and call information, then invokes server method $server.m(p)$ with these arguments;
4. Server object executes the method, and returns result to the skeleton;
5. Skeleton marshals result, sends this back to the stub;
6. Stub unmarshals result, and returns this to the client.

Using Java RMI

The *RMI registry* maps program names to actual programs on the server host. If *name* is registered, then URL $rmi://host:port/name$ identifies server program, where *host* is name/IP address of host running the registry, *port* is registry port number.

java.rmi package provides support for RMI classes: *Naming* class in this package encapsulates the RMI registry operations, and provides following static methods:

Methods of Naming class

$bind(name, ob)$	binds object *ob* to *name*
$rebind(name, ob)$	binds *ob* to *name*: replaces the old binding of *name*
$unbind(name, ob)$	removes the binding of *ob* to *name*
$lookup(url)$	returns the object *ob* named by *url*
$list(url)$	lists the registered bindings of the RMI registry running on *url*

To use RMI to manage communication between a local and remote program:

1. Define interface of the remote object, eg:

```
public interface Contract extends Remote
{
  public void aService(...) throws RemoteException;
```

```
  /* other services */ ...
}
```

Types of parameters and return types must be serialisable, so they can be marshalled and transmitted over network.

2. Define a server implementing *Contract*:

```
public class ServiceProvider extends UnicastRemoteObject
       implements Contract
{
  public void aService(...) throws RemoteException
  { /* implementation */  }
  /* implementation of other services */ ...
}
```

3. Create instance of this server class and register it:

```
Contract server = new ServiceProvider(...);
Naming.rebind(name,server);
```

4. Generate stub *ServiceProvider_Stub* and skeleton

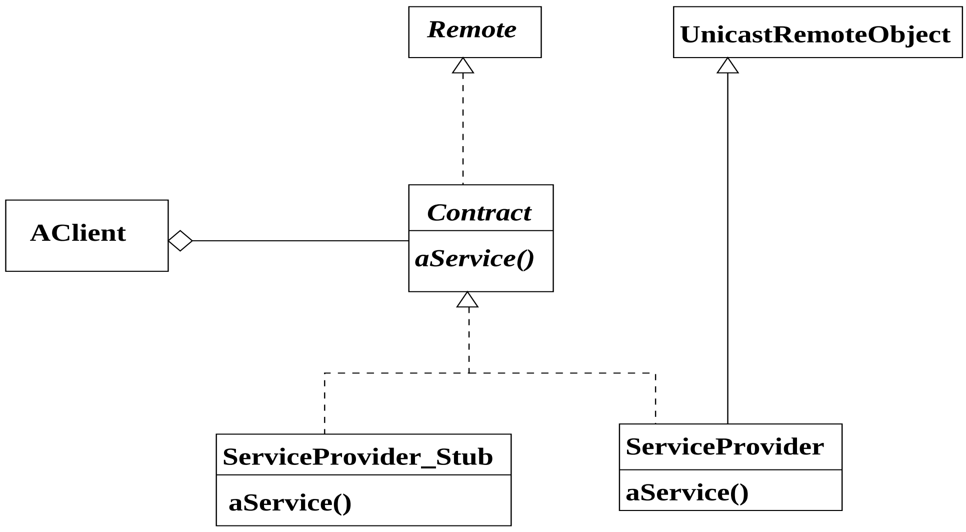
ServiceProvider_Skel: *javac ServiceProvider.java*, then run RMI compiler *rmic* on *ServiceProvider*.

5. Each client object must locate the remote server object, then invoke methods on it:

```
class AClient
{ ...
  Remote remoteObj = Naming.lookup("rmi://host:port/name");
  Contract serverObj = (Contract) remoteObj;
  ...
  serverObj.aService(...);   // remote invocation
  ...
}
```

remoteObj is an instance of the stub.

Typical RMI Application Structure



Correspondence with proxy pattern is:

- *Contract* corresponds to *Subject*.
- *AClient* corresponds to *Client*.
- *ServiceProvider_Stub* corresponds to *Proxy*.
- *ServiceProvider* corresponds to *RealObject*.

RMI is a good candidate for communication subsystem in restaurant management system: notification of new dishes and drinks for preparation can be sent to kitchen and bar interfaces by invoking operations on remote objects that manage these interfaces.

Creational Patterns

These patterns provide ways of managing object creation. They try to abstract from creation of specific instances of classes, to avoid dependence by one part of a system on exact classes provided by another.

Some examples are:

- Abstract Factory.
- Builder.
- Factory Method.
- Prototype.
- Singleton.

Structural Patterns

These patterns are concerned with composition of classes or objects into larger structures. Examples include:

- Adapter.
- Bridge.
- Decorator.
- Facade.
- Flyweight.
- Proxy.

Adapter and Facade, like Proxy, involve definition of an intermediary class between a client and supplier, in order to adapt services provided by supplier to those required by client (Adapter), and to reduce dependencies between client and its suppliers (Facade).

Behavioural Patterns

These patterns are concerned with organising patterns of communication between objects, algorithms and functional responsibilities of objects. Some examples of behavioural patterns are:

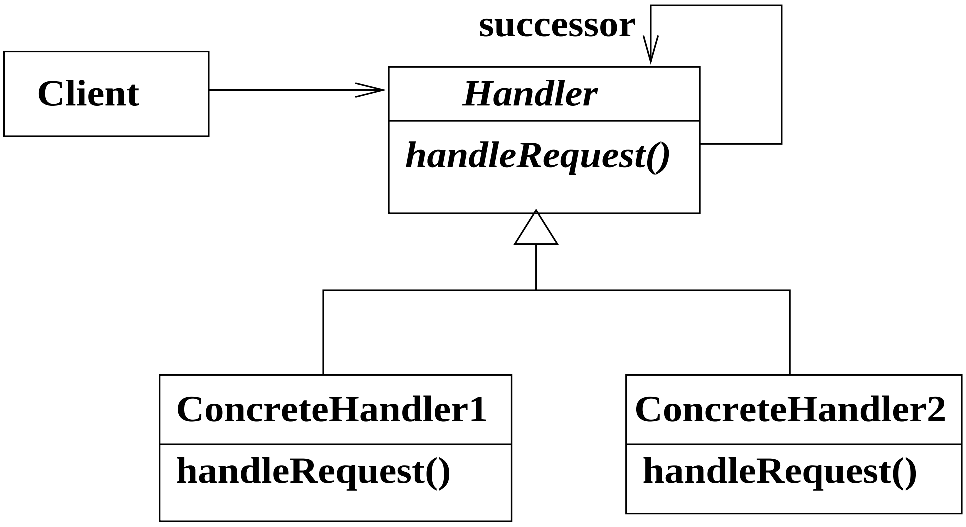
- Chain of Responsibility.
- Command.
- Observer.
- State.
- Mediator.
- Template Method.
- Visitor.

The Chain of Responsibility Pattern

This is an object-oriented analogue of the procedural programming pattern 'filter' or cascade of conditional tests: the chain of responsibility pattern enables a request to be passed down a chain of objects until one is able to respond to it.

Examples could include exception handling routines, utility cost calculation routines, etc.

Chain of Responsibility Class Structure



It is used:

- When more than one object may handle a request, and handler isn't known *a priori* to client.
- When a request should be issued to several objects, but you need to avoid explicitly specifying which ones (eg: in a dynamic network, a node may not be aware of what are all existing nodes).

The classes involved are:

- *Handler* – this defines interface for handling requests, and a successor link.
- *ConcreteHandler* – this handles those requests it is responsible for, otherwise it forwards request to its successor object.
- *Client* – initiates request to a *ConcreteHandler* object in the chain.

The benefits include:

- Reduced coupling – a client doesn't have to know what object will actually respond to its request.
- Flexibility in assigning responsibilities to objects – chain can be dynamically modified, unlike a static conditional cascade approach.

But need to ensure that always *some* object will eventually be reached that can handle each request.

Following is small example of applying pattern. Before pattern is used, we could have hard-coded structure of tests:

```
void Op()
{ if (C1)
  { Code1 }
  else if (C2)
  { Code2 }
  else
  { Code3 }
}
```

This code is directly written in the *Client*.

Using chain of responsibility pattern could replace this by:

```
abstract class Handler
{ protected Handler successor;

  public abstract void handleRequest();
```

```
}

class C1Handler extends Handler
{
  public void handleRequest()
  { if (C1) { Code1 }
    else
    { successor.handleRequest(); }
  }
}
```

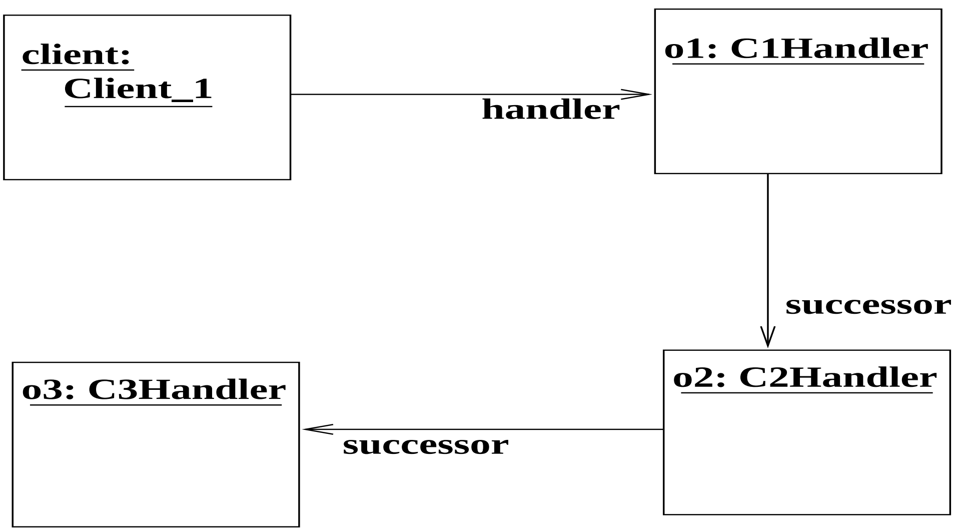
and similarly for $C2Handler$ and $C3Handler$, and we set up a chain of objects $o1 : C1Handler$, $o2 : C2Handler$ and $o3 : C3Handler$ with:

$$o1.successor = o2$$

and

$$o2.successor = o3$$

Object Structure of Chain of Responsibility



The client operation is:

```
void  Op() { handler.handleRequest(); }
```

where $handler : C1Handler$.

Part 5: Distributed and Internet Applications

This part describes design techniques for systems which use internet to support distributed processing, such as online banking applications, e-commerce, web-based teaching applications, etc.

- Architectures for Internet Applications
- Internet Application Programming Languages
- JavaScript
- JSP
- Servlets