PATTERNS UNIT -7 and 8 OF OOMD

By Dr. Radhika K R
Asst Prof, Dept of ISE,
B M S College of Engineering

(a) (b) (c)

(d) Figure 1

[courtesy: Internet pictures]

The above figures from (a) – (d) depict some of the patterns of real world. Figure (a) depicts a pattern to provide natural lighting in the room. Figure (b) depicts some number patterns kept on table, which has to be identified by kid to match the same with patterns lying down on the ground. Figure (c) represents another architectural marvel of lighting pattern. Figure (d) represents structural patterns for a shirt to be stitched. The basic component of the costume is represented. This forms a stitching pattern. The new shirts can be certain variations of

the pattern. With all the above examples, it is clear that we try to recall a similar problem which is already solved and reuse the essence of its solution to solve the new problem. It is a natural way of replicating with any kind of problem. The patterns can be used in architecture, economics, software engineering etc. Therefore we can infer the following characteristics of the pattern:

- Distilling out the common factors.
- Families of similar problems.

Alexander depicted the major architectural patterns and expressed it as “Timeless way of building” [1]. He stressed on the factor that the main problem lies in separating activities surrounding analysis and synthesis rather than recognizing their duality. Analysis is learning from decomposing the whole to sub-problems. Synthesis is learning about whole by analyzing sub-problems. These are methods to identify the common characteristics of problems. Like an organism, a building is more than a realization of a design or a development process.

Alexander's patterns are both a description of a recurring pattern of architectural elements and a rule for how and when to create that pattern. They are the recurring decisions made by experts, written so that those less skilled can use them. They describe more of the "why" of design than a simple description of a set of relationships between objects. Alexander's patterns that describe when a pattern should be applied are called "generative patterns".

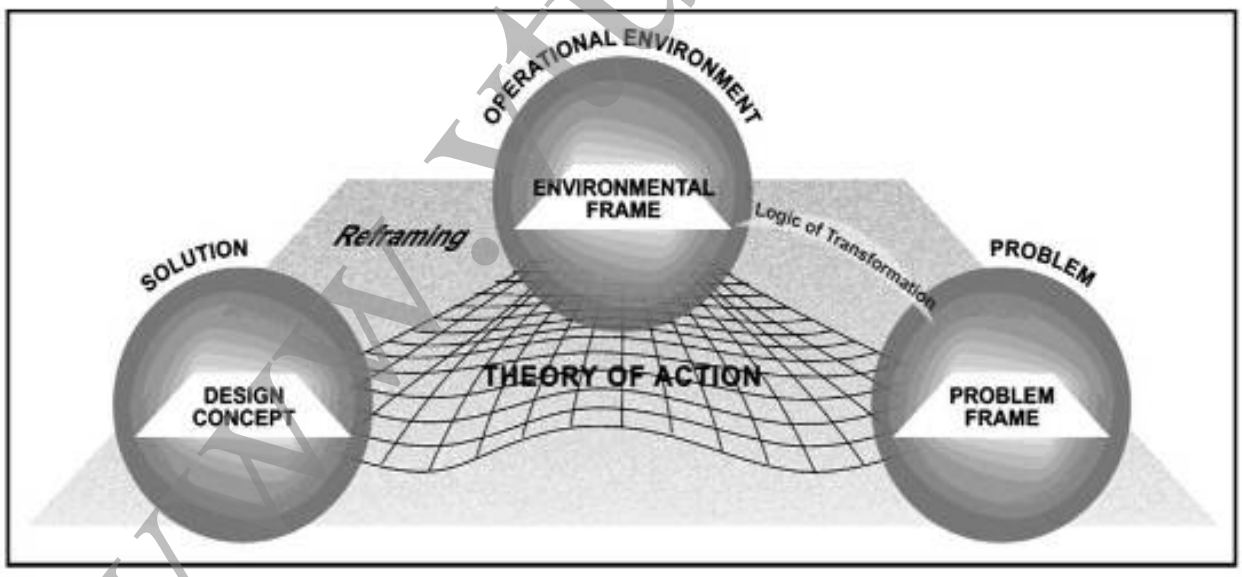
Generative patterns share many advantages:

They provide a language for designers that makes it easier to plan, talk about and document designs. They have the added advantage of being easier for non-experts to use and providing a rationale for a design.
A more detailed aspect of the theory is problem solvers utilize means-ends analysis (Newell & Simon, 1972) [2]. Search methods involve a combination of selecting differences between the desired and current

states. Selecting operators that will reduce the chosen differences and applying the operators. Here, we are trying to provide similar characteristics of the problem. Creating sub-problems to transform the current states to desired states are achieved. The operators are tools to distill the similar concepts. Chaining backward from aspects of the goal state with relevant operators to determine useful sub-goals. The backward chaining is analysis. But with experience, the strategy can be replaced by experts with forward chaining that leads directly to the goal.

Context-Problem-Solution

Context : A situation giving rise to a problem.
Problem : Recurring problem arising in context.
Solution: A proven resolution of the problem.



[courtesy: Internet pictures] Figure : 2

It is common understanding that if in a situation we arrive at a problem which occurred previously, we try to provide the same solution, which we provided earlier. It is a concept of re-usability which is depicted in Figure

2. In software using the set of library functions is an example of recurring problem.

Definitions-PATTERN

Approach - 1

❖Each pattern is a three-part rule.

- A relation between a certain context.
- A problem.
- A solution.

Approach - 2

❖Each pattern is a relationship.

- A context.
- A system of forces which occurs repeatedly in that context. Force denote any aspect of the problem that should be considered when solving it.
- A spatial configuration which allows these forces to resolve themselves.

Approach - 3

❖Each pattern is an instruction which helps for re-usability.

- A relevant context.
- To resolve system of forces.

Approach - 4

❖Each pattern is both a process and thing.

- A description of thing.
- A description of the process which will generate the thing.

Let us consider software with a human-computer user interface. The interface is prone to change. To provide the functionality extension will be challenging. Usually user interface will be adapted for specific customers. The interface will look different when posted to another platform. When upgrading an operating system the parameters to provide

user interface may change. Keeping track of the above points, if the flexibility is provided, it will increase the cost of development and copies of different implementations have to be saved in memory. There exist certain scenarios where non-graphical user interface has to be re-modified for graphical interface or vice-versa. The user-interface can also be dependent on the input device. Therefore we can make the functional core to be independent from user interface.

The important aspects of user-interface development are:

1. Changes to user interface should be easy and possible at run time.
2. Adapting or porting the user interface should not impact code in the functional core of the application.

Display should be separately designable/evolvable.

Design: Model-view-controller triad

Processing	Model	Core data and functionality
Output	View	Display to user
Input	Controller component	Mouse/keyboard events

Basic parts of any application:

1. Data being manipulated by the input events.
2. A user-interface through which data manipulation occurs.
3. The data is logically independent from how it is displayed to the user.

The internal representation of data is different.

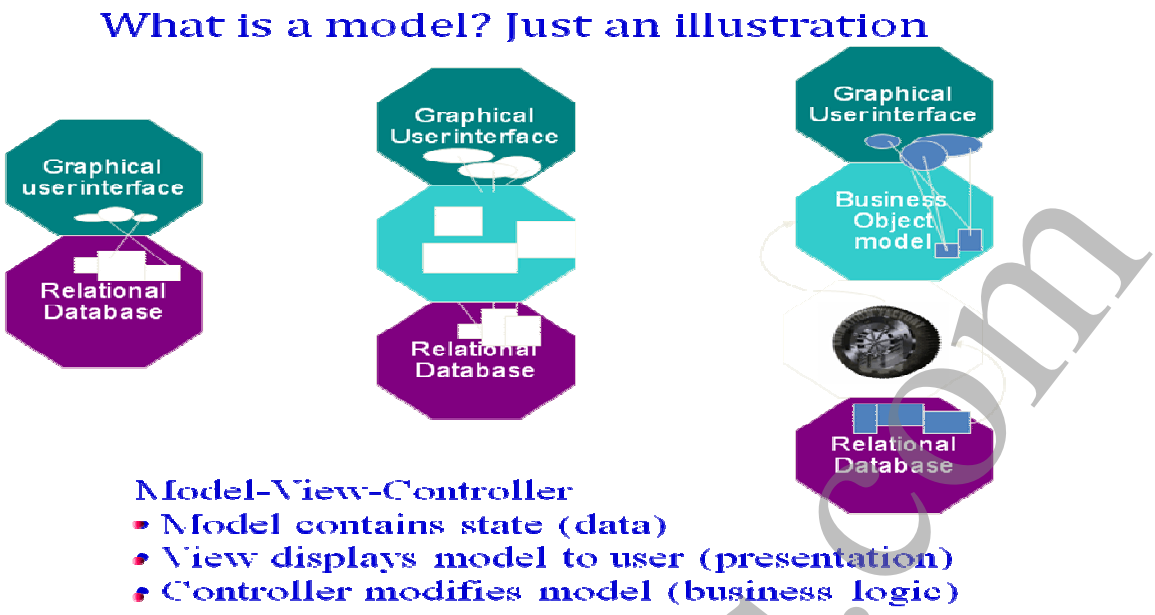


Figure: 3

Figure 3 represent the evolution of controller. Let us consider the example of retrieving the relational data represented. There exist a logic to retrieve/update/input the data. This we call it as business logic. The graphical user interface portrays the input process. This is the point of input. Relational data base is the model to save the input in the memory.

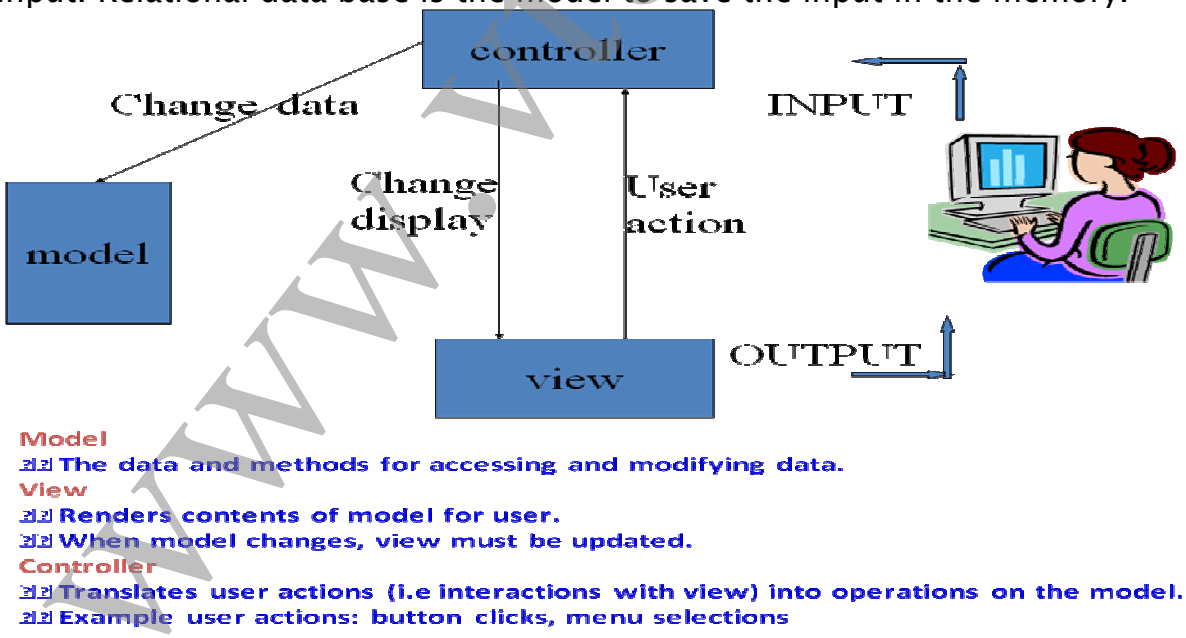


Figure 4

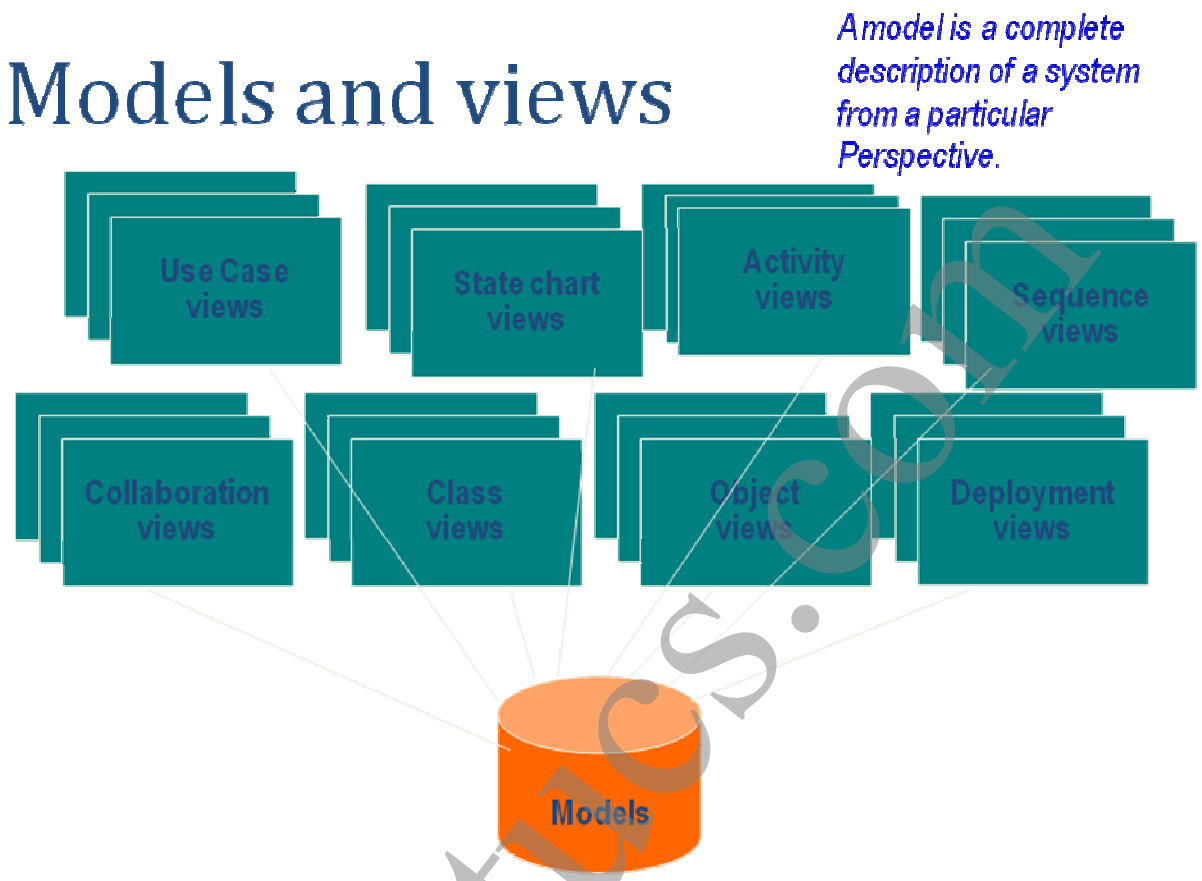


Figure 5

The figure 5 represents many views for a model. This is due to the reason that many types of users exist. Therefore many views. User interface is many things to many different interested parties such as end-user, customer, project manager, system engineer, software developer, tester, model developer, integrator of modules. If there are multiple stakeholders, then multiple views of same model exist.

The salient features of models:

1) Models are the language of designer, in many disciplines.
2) Models are representations of the system to-be-built or as-built.
3) Models are vehicle for communications with various stakeholders.
4) Visual models also exist.
5) Scaling of data will be allowed in a model.
6) Models allow reasoning about characteristic of the real system.

To achieve clear understanding of model-view-controller, the set-up sequence can be analyzed. The model is initialized. After this the view is initialized. The reference to a controller is made. The controller is the middle component, therefore controller references to both model and view. The updations are only through controller, hence controller registers with view.

At execution level, view recognizes input event. Input event is data entry or menu selection. View procedure calls appropriate method on controller. Controller procedure accesses model and updates it. If model has been changed, view is updated (via the controller).

With this set-up functional core independency is achieved. We can change from a non-graphical to a graphical user interface, without modifying the model subsystem. (Independent of specific 'look and feel'). We can add a support for a new input device without affecting information display or the functional core.

The following points summarize the properties of patterns:

1) Addresses a recurring design problem that arises in specific design situations and presents a solution to it.

Figure 6

Example: Supporting variability in user interface as shown in Figure 6.

This problem may arise when developing software systems with human-computer interaction.

2) Document existing, well-proven design experience.

Patterns are not invented or created artificially. They distill and provide a means to reuse the design knowledge gained by experienced

practitioners. Those familiar with an adequate set of patterns can apply them immediately to design problems without having to rediscover them.

3) Identify and specify abstractions that are above the level of single classes and instances or of components.

A pattern describes several components, classes, objects. Patterns portray details, responsibilities, relationships and co-operation. All components together solve the problem that the pattern addresses. (more effectively than a single component).

4) Provide a common vocabulary and understanding for design principles.

Pattern names if chosen carefully, become part of widespread design language. They facilitate effective discussion to design problems and solutions. They remove the need to explain a solution to a particular problem with a lengthy and complicated description.

5) Means of documenting software architectures.

Patterns can describe the vision of the originator when designing a software system. This helps others to avoid violating this vision when extending and modifying the original architecture.

6) Support the construction of software with defined properties.

Patterns provide a skeleton of functional behavior and therefore help to implement the functionality of the application. patterns also provide explicitly non-functional requirements for software systems such as reliability, testability and reusability.

7) Patterns help to build complex and heterogeneous software architectures.

Using predefined design artifacts supports the speed and quality of design. Understanding and applying well written patterns saves the time when compared to searching for the solutions. Not to say “better than own solutions”, but at least provides a platform to evaluate and assess

design alternatives. Patterns may not provide complete solution for new scenario and can be extended.

8) Help to manage software complexity.

Every pattern describes a proven way to handle the problem it addresses on basis of the kinds of components needed, component roles, details that should be hidden, abstractions that should be visible and working parameters.

Summary

A pattern for software architecture describes a particular recurring design that arises in specific design context and represents a well proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships and the way in which they collaborate.

Summary – what makes a pattern

The context extends the plain problem-solution dichotomy by describing situations in which the problem occurs. The context of a pattern may be fairly general. Problem that arises repeatedly in the given context. The general essence is identified with the concrete design issue that must be solved.

Aspects in solution: The following equation illustrates the static and dynamic structure of solution to certain extent. Static structure deals with the component structure and its relationship with other components. dynamic structure deals with processing details at run-time. The collaboration of components and variable values updation between the components.

Static →

Structure = Components + Relationship

Dynamic →

Run-time behavior = Collaborate + Organize + Communicate

Pattern categories depend on various range of scale and abstraction. They help in structuring a software system into subsystem with refinement of components. Particular design aspects in a specific programming language will lead to a pattern. Domain independent decoupling of interacting components lead to sub patterns.

Types of patterns:

Type	Description	Examples
Architectural patterns	Fundamental structural organization is expressed for software systems that provide a set of predefined subsystems. Relationships are specified. These patterns include the rules and guidelines for organizing the relationships.	MVC
Design patterns	Capture the static – dynamic roles & relationships in solutions that occur repeatedly.	Observer Publisher-subscriber
Idioms	Restricted to a particular language.	Counted Body

MVC

Model-view-controller is a pattern used to isolate business logic from the user interface. The Model represents the information of the application and the business rules used to manipulate the data. The View corresponds to user interface such as text and checkbox items. The Controller manages the communication between the model and view. The controller handles user actions such as keystrokes and mouse movements. User actions are sent to the model or view as required.

Example 1: Calculator

1) The Form → view
2) Events → controller
3) Controller will call methods from model → ADD/Subtract/…

The model takes care of all the components and it maintains the current state of the calculator.

Example 2: Web usage

View → The actual HTML page,

Controller is the code that gathers dynamic data and generates the content within the HTML. Model is the actual content stored in a database. Business rules that transform the content based on user actions also reside in model. The user interacts with the user interface.

He/she position the mouse to select or press a key on keyboard. A controller registers the above. The controller notifies the model of the user action. For example, change or updates to user's shopping cart is notified. A view uses the model (indirectly) to generate the screen listing the shopping cart contents. The model has no direct information of the view. The user interface waits for further user interactions that lead to new transaction.

Observer

The Observer design pattern has subject + observer. The relationship between subject and observer is one-to-many. In order to reuse subject and observer independently, decoupling is a must. For example in a graphical interface toolkit, there exist presentational aspect and application data. The presentation is the observer. The application data is the subject part. The different ways of presenting the data, lead to many observers for a single subject. Let us consider the excel sheet data. The data can be used by a formula, to display the analyzed data. Suppose the excel sheet consists of class marks in a subject, we can put a formula to depict how many students have > 40 marks. It can be Boolean operator to depict TRUE or FALSE. The same can be expressed as a bar-graph chart display. It can also be represented as a pie-chart display of marks. The formula, bar graph and pie-chart form the different observers for the

mark list. Here the subject is marks list in the particular subject as shown in Figure 7.

Example for observer:

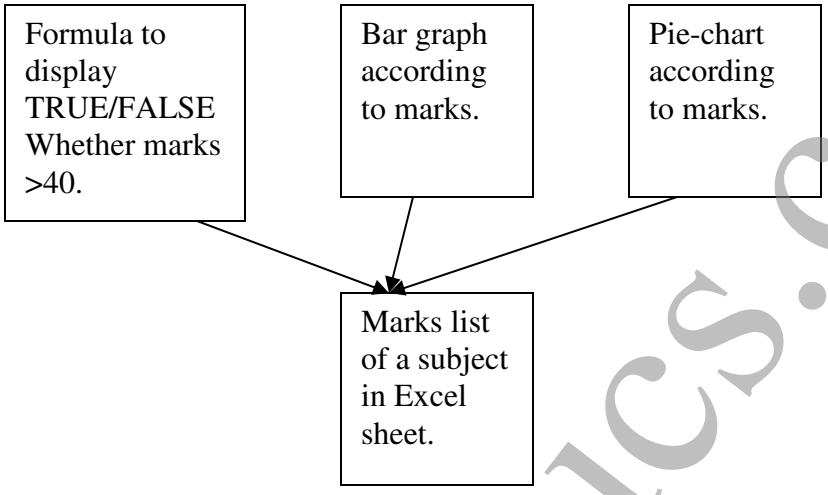


Figure 7

The Spreadsheet data object notifies its observers whenever a data changes. Subject state will be inconsistent if the observers are not updated.

When to apply Observer pattern???

1) When the abstraction has two aspects with one dependent on the other. Separate objects will increase the chance to reuse independently.
2) When the subject doesn't know exactly how many observers exist .
3) When the subject object should be able to notify it's observers without knowing who they are.

Observer

Context – A component uses data or information provided by another component.

Problem – The components should be loosely coupled. The information provider should not depend on details of its collaborators. The components that depend on the information provider are not known a priori.

Solution – change propagation mechanism between subject and observer. Subject is information provider. Observer is a component dependent on subject. Observers can register and unregister. Whenever the subject changes its state, change-propagation mechanism will be started to restore consistency with all registered observers. Changes are propagated by invoking a special update function common to all observers.

The principle behind reference counting is to keep a count of an object. When it falls to zero the object is unused. Used to simplify the memory management for dynamically allocated objects. To keep a count of the number of references held to that object. When the count reaches zero the object is deleted. This is used for garbage collection in C++.

When to be careful with observer pattern???

- Abstract coupling where subject and observer are extended and reused individually. Sometimes decoupling should allow platform independency.
- Dynamic relationship between subject and observer will be established at run time.
- Support for broadcast communication. The notification is broadcast automatically to all interested objects that subscribed to it.
- Observers have no knowledge of each other. With the dynamic relationship between subject and observers, the update dependency is difficult to track.

Counted body (Language –Smalltalk)
Special pointer and reference types are avoided.
Context –
The interface of a class is separated from its implementation. The two classes are created. The handle class and body class. The user interface is taken care by handle class. The body class deals with the implementation of the object. The handle forwards member function invocations to the body class.
Problem –
The problem forces to be addressed are : Copying of bodies is expensive and cleaning up of the object should achieved at run-time (built-in types and user-defined types). Sharing bodies on assignment is semantically incorrect if the shared body is modified through one of the handles.
Solution –
The memory management is done by handle class and reference count is achieved by body class. It is responsibility of any operation that modifies the state of the body to break the sharing of the body by making its own copy, decrementing the reference count of the original body. Sharing is broken, if body state is modified by any handle. Sharing is preserved if common case of parameter passing is achieved.

Relationships between patterns can be single components or relationships inside a particular pattern described by smaller patterns. Relationship can be achieved by all smaller patterns that are integrated by the larger pattern in which they are contained.

Example 1: Refinement of the model-view-controller pattern.
The architectural pattern is converted to design pattern. The MVC pattern is represented as observer- subject pattern. The model is subject pattern. The views and controllers form observer pattern. whenever the state of the model changes, we must update all its dependent views and controllers.

Example 2: Document-view variant of the MVC pattern.
Ability to change input and output functionality dependently. For instance the text selection and making it bold. The view and controller component form a single component called as view.

Inter Process Communication

Figure 8

Example 3: Transparent peer-to-peer inter-process communication.
The three problem forces are time should not be spent for searching remote servers, independent from particular IPC mechanism and client /server together should be depicted as single process. The first two problem forces can be solved by forwarder-receiver pattern. The third problem force can be resolved by a proxy pattern.

Forwarder-receiver pattern offers a general interface for sending and receiving messages and data across process boundaries, as shown in figure 8. Forward-receiver pattern hides the details of the concrete inter-process communication mechanism. Forward-receiver pattern provides name-to-address mapping for servers. Proxy acts as representative of server for client. Proxy knows server name and forwards every request of the client to it as shown in figure 9. The optional process boundary is created.

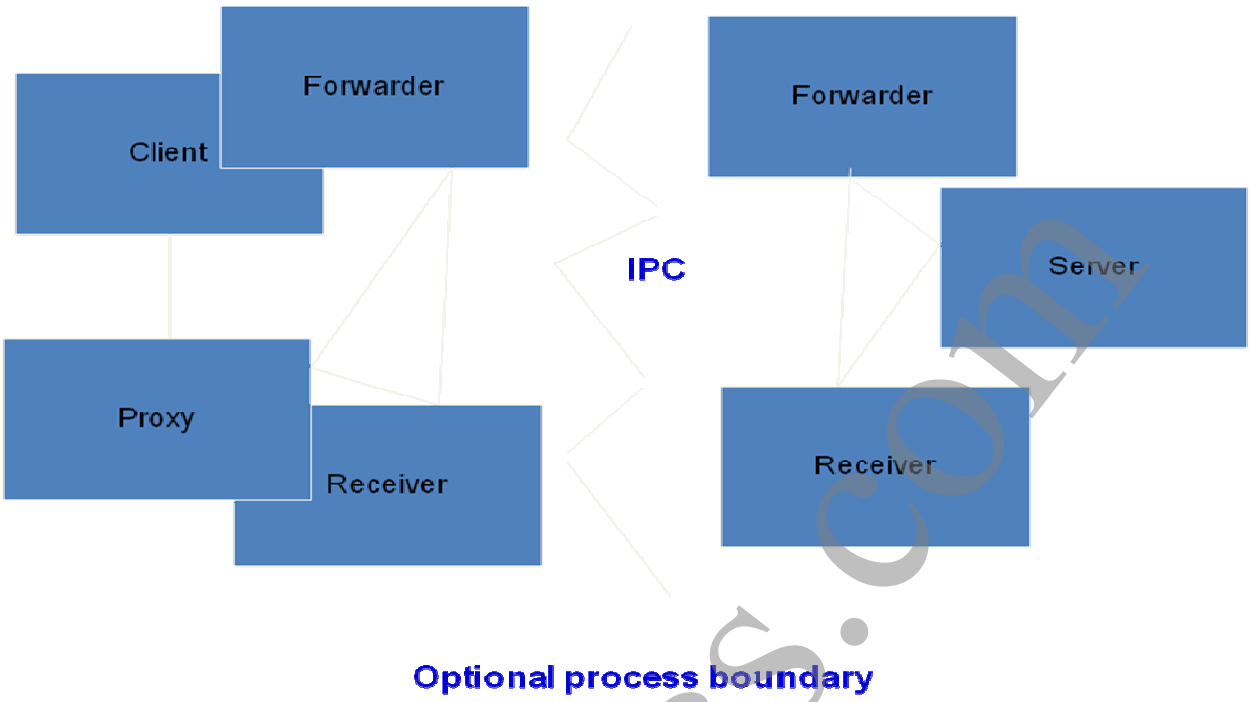


Figure 9

Pattern description

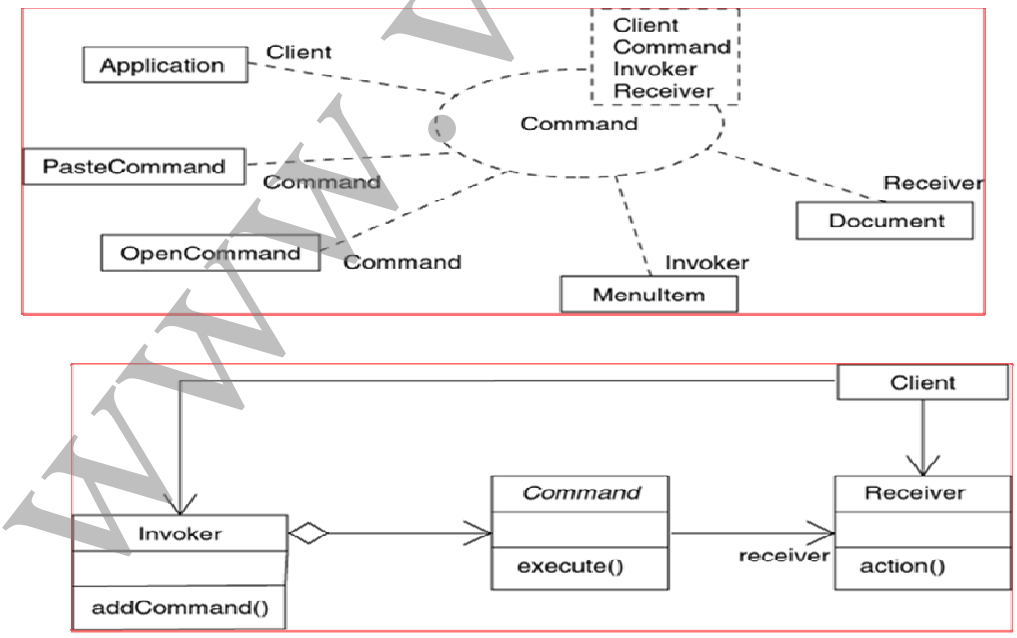


Figure 10

Courtesy : Internet figures

In figure 10, the two ways of describing the problem is provided. Both represent how a particular command pattern works. The pattern description provide all the details necessary to implement a pattern and to consider the consequence of its application. i.e all the details should be provided for proper usage. The foremost thing is, pattern must be named intuitively to aid sharing. An introductory example to a pattern to explain the problem and forces can be given. By this a non-expert can also understand the context and problem. The solution provided can later be re-used. Therefore implementation guidelines for the pattern should be provided. The previous solution platforms or successful implementations can be stated. List of successful applications of the pattern will enhance its credibility. The variations on the existing patterns can also be listed. Variants patterns or alternative solutions are listed. Benefits and potential liabilities of a pattern will aid in solution decision. In the previous section it was mentioned that patterns can be refined or combined to form new patterns. The cross references are provided to other related patterns which are refined current pattern or similar problem addressing patterns.

Pattern description template should provide:

- Name
- Other names
- Real-world example
- Context
- Problem
- Solution
- Structural aspects
- Run-time behavior
- Implementation guidelines
- Resolved example
- Variant
- Examples of uses
- Consequences (benefits and liabilities)
- Cross reference

we like a pattern which gives us something to solve the problem besides only read-through. So somebody should have described the pattern correctly.

Communication patterns

Important aspects of communication patterns are:

Encapsulation of communication facilities.

I. Hiding the details of the underlying communication mechanism from the users.
II. Abstract programming interface is provided on top of the low-level communication facilities.

Location Transparency.

I. Applications are allowed to access remote components without any knowledge of their physical location.

Three major patterns

Patterns	Problem force addressed
Forward-Receiver	Encapsulation of communication facilities.
Client-Dispatcher-Server	Location Transparency.
Publisher-subscriber	Consistency b/w operating components

Forwarder-Receiver design pattern Provides transparent inter process communication for software systems with peer-to-peer interaction model.It introduces forwarders and receivers to decouple peers from the underlying communication mechanisms.

Example

The company DwarfWare offers applications for the management of the computer networks. System consists of agent processes written in Java

that run on each available network node. These agents are responsible for observing and monitoring the resources. Routing tables get modified. Each agent is connected to remote agents in a peer-to-peer fashion, acting as client or server as required. As the infrastructure needs to support a wide variety of different hardware and software systems, the communication between the peers must not depend on a particular mechanism for inter-process communication.

Context – peer-to-peer communication
Problem forces –

1.1 The system should allow the exchangeability of the communication mechanisms.
1.2 The co-operation of components follows a peer-to-peer model, in which a sender only needs to know names of its receivers.
1.3 The communication between peers should not have a major impact on performance.

Solution – peers may act as clients or servers. Therefore the details of the underlying IPC mechanisms for sending or receiving messages are hidden from peers by encapsulating all system-specific functionality into separate components. system specific functionalities are the mapping of names to physical locations, the establishment of communication channels and marshaling and unmarshaling messages.

Structure – Forwarders, receivers and peers are agents. Each peer knows the names of the remote peers with which it needs to communicate. It uses a forwarder to send a message to other peers and a receiver to receive the messages from other peers. The different types of messages that can be passed between the two peers are command messages, information messages and response messages. Command messages are related to implementation process. For instance, changing of routing tables of host machine is a command message. The payload i.e data on the network is communicated via the information messages. For instance, data on network resources and network events form information messages. Response messages deals with acknowledgement details such as, the arrival of a message.

Peer pattern

1) Continuously monitor network events and resources.
2) Listen for incoming messages from remote agents.
3) Each agent may connect to any other agent to exchange information and requests.
4) The network management infrastructure connects the network administrator's console with all other agents.
5) Administrators may send requests to network agents or retrieve messages from them by using available network administration tools.

Forwarder pattern

1) Sends messages across process boundaries.
2) Provides a general interface that is an abstraction of a particular IPC mechanism.
3) Includes functionality for marshaling and delivery of messages.
4) Contains a mapping from names to physical addresses.
5) Determines the physical location of the recipient by using its name-to-addresses mapping.
6) In the transmitted message, the forwarder specifies its own peer, so that remote peer is able to send a response to the message originator.

Receiver pattern

1) Wait for incoming messages on behalf of their agent process.
2) As soon as the message arrives, they convert the received data stream into a general message format and forward the message to their agent process.

Class *peer* *Responsibility* • *Provides application services.* • *Communicates with other peers.*	*Collaborators* • *Forwarder* • *Receiver*

Class *Forwarder* *Responsibility* • *Provides general interface for sending messages.* • *Marshals and delivers messages to remote receivers.* • *Maps names to physical addresses.*	*Collaborators* • *Receiver*

Class *Receiver* *Responsibility* • *Provides general interface for receiving messages.* • *Receives and unmarshals messages from remote forwarders.*	*Collaborators* • *Forwarder*

The forwarder pattern and receiver pattern are the collaborators for peer class. i.e peer co-exist with the two communication patterns. The functional core of the application will be represented be peer pattern. The communication mechanism overheads are handled by forwarder and receiver patterns. The application code is decoupled from the network communication code. The forwarder will be the starting point of communication channel at source. The receiver pattern are will be the ending point of communication channel at destination. The two peers can be residing on same machine and sharing the memory or the two peers can be two different systems distributed over a network with different communication mechanisms. Remote procedural call is a mechanism of communication, when peers share local memory. TCP/IP or UDP will be communication mechanisms for distributed systems. The communication mechanism is related to operating system also.

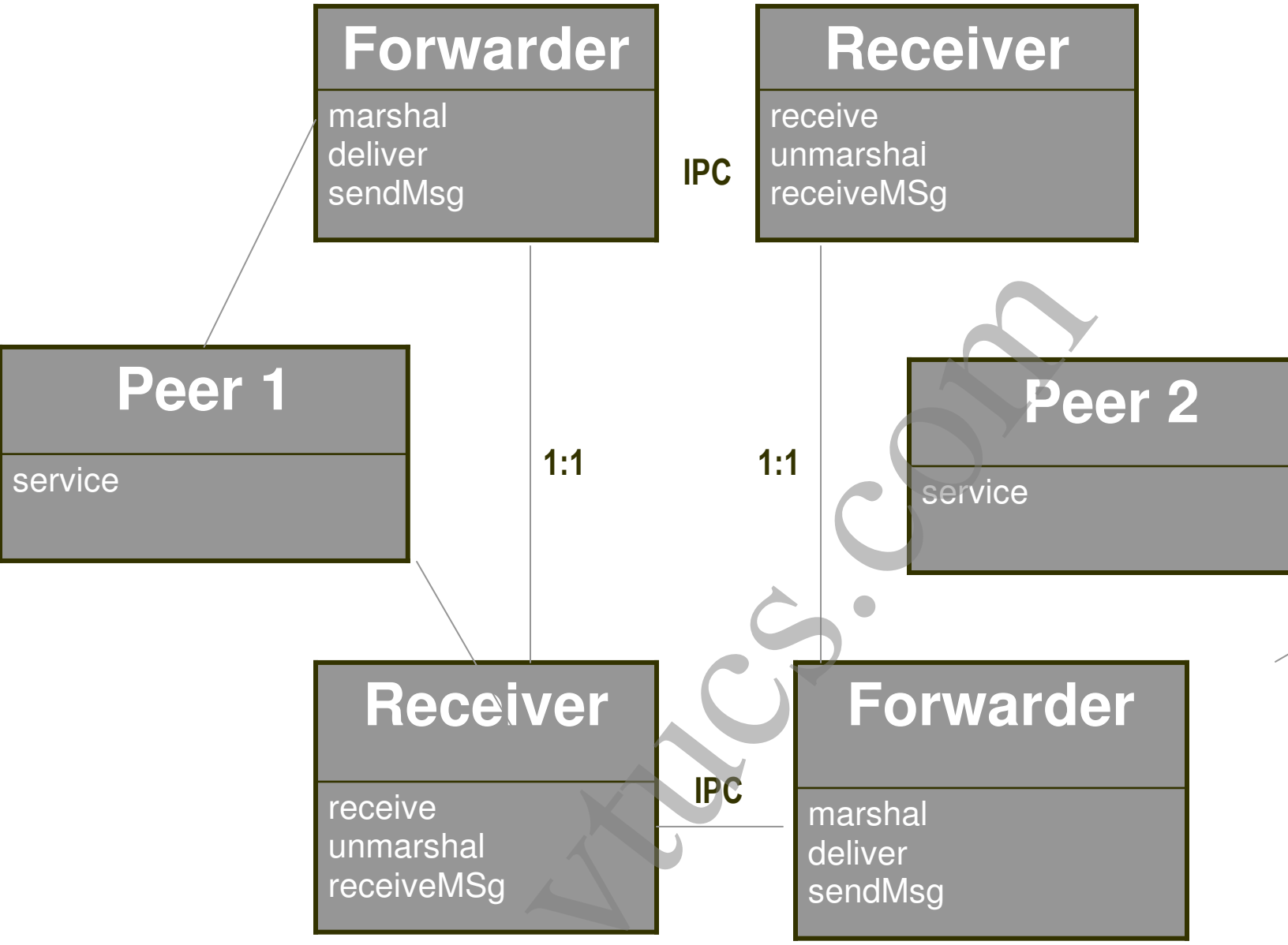


Figure 11

Dynamics

Two peers p1 and p2 communicate with each other as shown in figure 11. P1 uses a forwarder Forw1 and receiver Recv1. P2 handles all message transfers with a forwarder Forw2 and a receiver Recv2.

1) P1 requests a service from a peer P2. It sends the request to its forwarder Forw1 and specifies the name of the recipient.
2) Forw1 determines the physical location of the remote peer and marshals the message.

3) Forw1 delivers the message to the remote receiver Recv2.
4) P2 would have requested its receiver Recv2 to wait for an incoming request. Recv2 receives the message arriving from Forw1.
5) Recv2 unmarshals the message and forwards it to its peer P2.
6) P1 calls its receiver Recv1 to wait for response.
7) P2 performs the requested service and sends the result and the name of the recipient P1 to the forwarder Forw2. The forwarder marshals the result and deliver it to Recv1.
8) Recv1 receives the response from P2, unmarshals it and delivers it to P1.

Implementation

The first step is to specify a name-to-address mapping catalog. The name of the system can be maintained with a convention. The actual physical address of the peer has to be mapped to name of the peer. Peers reference other peers by name, namespace should be created. A namespace defines rules and constraints to which names must confirm in a given context. A name does not necessarily refer to single address. It may refer to a group of addresses. For instance, name can be stated as peerVideoServer. The naming convention is of 15characters and starts with capital letter. The alternative naming convention can be structure names as pathnames. For instance, '/Server/VideoServer/AVIServer'

What does the name reflect??? A name refer to a single address or a name refer to a group of addresses or a name represents a group of remote peers or a name represents a group which is a member of another group.

After specifying name-to-address mapping catalog, the message protocols to be used between peers and forwarders should be specified. Protocol defines the detailed structure of message data a forwarder receives from its peer. Data is partitioned into multiple packets. Messages contain sender and data. Messages do not contain the name of

the recipient. (This allows to send to more than one recipient) The name of the recipient is passed by the sender as an extra argument to its forwarder. Time-out values provided to forwarders and receivers will not block the system when failure of receiving responses arise. In case of communication failure, forwarders and receivers shall report an exception.

The communication mechanism is selected. The operating system provides the communication mechanism. Low-level mechanisms (TCP/IP) are efficient and flexible in communication protocols. Low-level mechanisms require more programming and they are platform dependent. High-level mechanisms such as socket programming are platform independent. Sockets are available on most of the operating systems.

The forwarder is implemented. The forwarder provides its functionality through a public interface and encapsulates the details of a particular IPC mechanism. The mapping repository is created between names and physical addresses. The mapping repository may be static or dynamic. The dynamic repository will provide the updations such as migration at run-time. Forwarder uses this repository to establish the communication link to the remote peer.

Forwarder can use private repository or common (global) repository. The repository uses hash table to manage all address mappings. The hash table will have a mechanism to generate the physical address once the name of the peer is given. It is not searching through the list. A black box which takes name as input and provides physical address as output. The collisions have to be resolved by the formula or black box, not to give same physical address for two different names.

The local repository can be implemented, scope of which is within the intranet. Many local repositories can have different names for same system. These name collisions have to be rectified, if it is local repository.

The global repository will have one entry of physical address for a peer. It just acts as a look-up table.

The physical address structure is determined by the IPC mechanism used. The following specifies the two instances.

physical address = Internet address + socket port
physical address = target machine name + socket port

```
Class Registry {
 private Hashtable htable = new Hashtable();
 public void put(String thekey, Entry theEntry) {
 htable.put (theKey, theEntry);
}
Public Entry get(String aKey)
{
Return (Entry) htable.get(theKey);
}
}
```

Looking at the code above, we can understand that, there exist an entry in the hash table for each peer. The 'put' method will hash the incoming conventional name to a physical address. The 'thekey' here depicts the name of the peer. The 'Entry' provides the actual physical address. Whole-Part design pattern can be used for following forwarder responsibilities such as marshaling, message delivery and repository.

The receiver is implemented. Receivers run asynchronously. Decision should be made whether the receivers should block until a message arrives. Receivers may wait for an incoming message. The control is returned to peer when it receives a message. OR Non-blocking receivers can be implemented by specifying the time out values. More than one communication channels within receiver can be used. Multiple threads within the receiver can be created. Each thread will be responsible for a

particular communication channel. Internal message queues should be used for buffering messages.

The two important aspects of receiver are :

1) After receiving the socket port number from global repository, receiver opens a server socket and waits for connection attempts from remote peers.
2) As soon as a connection is established with a second socket, the incoming message and its size are read from the communication channel.

The peers of application are implemented. Peer acting as clients, send a message to a remote peer and waits for the response. After receiving the response, it continues with its task. Peers acting as servers continuously wait for incoming messages. When such a message arrives, they execute a service that depends on the message they received and send a response back to the originator of the request.
For asynchronous communication – one way communication can be implemented. In one-way communication peer sends a message without requiring a response. For synchronous communication – two way communication can be implemented. In two-way communication peer sends a message and waits for response.

A start-up configuration is implemented.
1) A start up routine should create a repository of name-address pairs.
2) Repository can allow different peers to have different name-to-address mapping.
3) Forwarders and receivers must be initialized with a valid name-to-address mapping.
4) The repository configuration can change dynamically i.e at run-time.

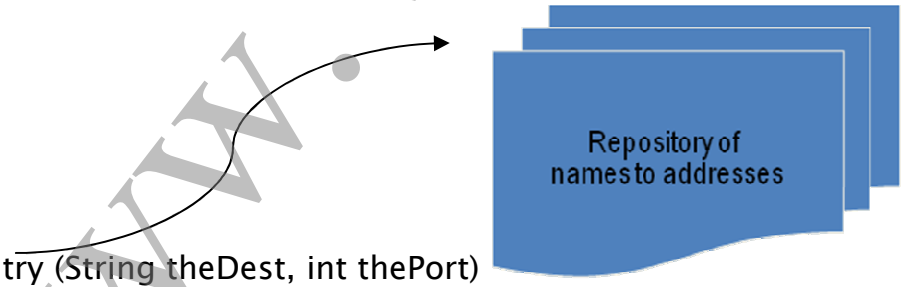
Forwarder-receiver pattern

```
class Message{
public String sender;
public String data;
public message (String thesender,
String rawdata)
{
Sender = theSender;
Data = rawData;
}
}
```

- Peers use objects of the class Message when they invoke their forwarder.
- A receiver returns a message to its peer when it receives a message.
- Messages only contain the sender and the message data.
- Name of the recipient is passed to the forwarder.

Forwarder-receiver pattern

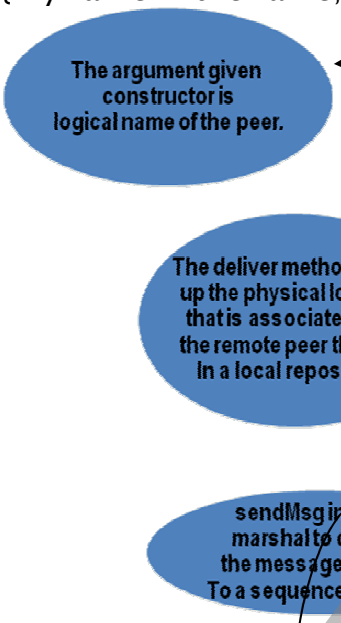
```
class Entry{
public String destinationID;
private int portNr;
```




```
public Entry (String theDest, int thePort)
{
destinationId = theDest;
portNr = thePort;
}
public String dest(){
return destinationId;
}
public int port() {
```

return portNr;
}
}
The repository maps names (strings) to the instances of the class Entry.

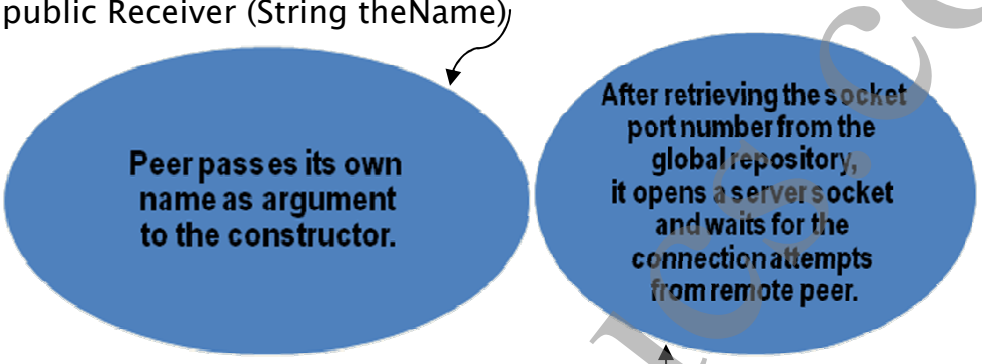
```
class Forwarder  {
private socket s;
private OutputStream ostr;
private String myname;
public Forwarder(String theName)
{myName = theName;
```




```
private void deliver(String theDest, byte [ ] data)
{
Try { Entry entry = fr.reg.get(theDest);
S = new Socket(entry.dest(),  entry.port()};
oStr = s.getOutputStream();
oStr.write(data);
}
catch(…)
Public void sendMsg(String theDest, Message theMsg)
{
```

```
deliver(theDest, marshal(theMsg));
}
}

class Receiver {
private Serversocket srvs;
private Socket s;
private Inputstream istr;
private String myName;
public Receiver (String theName)
```




```
{myName = theName;}
private Message unmarshal(..){…}
private byte[ ] receive( ) {
Entry entry = fr.reg.get(myName);
srvS = new ServerSocket(entry.port(),1000);
s = srvS.accept();
iStr = s.getInputstream();
val = iStr.read();….}
public Message receiveMsg()
{
return unmarshal(receive());
}  }
```

Example of peer acting as a server:

```
class Server extends Thread {
Receiver r;
Forwarder f;
```

```
public void run() {
Message result = null;
r = new Receiver("Server");
Result = r.receiveMsg();
f = new Forwarder("Server");
Message msg = new Message("Server is listening.");
f.sendMsg(result.sender, msg);
}
}
```

Significant Liability of forwarder-receiver design pattern is, there is no support for flexible reconfiguration of components. i.e Forwarder-Receiver cannot adapt if distribution of peers change at run-time. The change affects all peers collaborating with the migrated peer.
The problem can be solved by adding central dispatcher component for forwarder-Receiver pattern.

Client-dispatcher-server pattern

Provides transparent inter-process communication for software systems in which the distribution of components is not known at compile time i.e may vary dynamically at run-time. This pattern allows peer to migrate to other locations at run-time by unregistering and re-registering with the dispatcher.
This pattern provides an intermediate layer between clients and servers called as dispatcher. The pattern provides location transparency by means of a name service. The pattern hides the details of the establishment of the communication connection between clients and servers.

Example

To develop a software system for the retrieval of new scientific information. The information providers are both on our local network and distributed over the world. To access an individual information provider, it is necessary to specify its location and the service to be executed.

When an information provider receives a require a request from client application, it runs the appropriate service and returns the requested information to the client.

Context: A software system integrating a set of distributed servers, with the servers running locally or distributed over a network.

Problem forces:

- A component should be able to use a service independent of the location of the service provider.
- The code implementing the functional core of a service consumer should be separate from the code used to establish a connection with service providers.

Solution

Dispatcher implements a name service to allow clients to refer the servers by names instead of physical location. Dispatcher is responsible for establishing a communication channel between a client and a server.

Each server is uniquely identified by name and is connected to client by dispatcher. Clients rely on dispatcher to locate a particular server and connect. The roles of the server and client can change dynamically.

Structure

1> Before sending a request to a server, the client gets information from dispatcher for communication channel.

2> Servers provide set of operations to clients.

3> Server registers itself OR is registered with the dispatcher by its name and address.

4> The server component can be located on same computer as a client or may be reachable via a network.

5> The dispatcher establishes a communication link to server using available communication mechanism and returns a communication handle to the client.

6> Dispatcher implements registering of servers.

Class *client* *Responsibility* ● *Implements a system task.* ● *Requests server connection from dispatcher.* ● *Invokes services of server.*	*Collaborators* ● *Dispatcher* ● *Server*

Class *server* *Responsibility* ● *Provides services to clients.* ● *Registers itself with the dispatcher.*	*Collaborators* ● *Client* ● *Dispatcher*

Class *Dispatcher* *Responsibility* ● *Establishes communication channels between clients and servers.* ● *Locates servers.* ● *(un-) registers servers.* ● *Maintains a map of server locations*	*Collaborators* ● *Client* ● *Server*

The collaborators of client pattern are dispatcher and server patterns. Similarly the collaborators of server pattern are dispatcher and client patterns.

Dynamics

- A server registers itself with the dispatcher component.

- Client requests the dispatcher for a communication channel specified by the client in its registry.
- The dispatcher establishes a communication link to server.
- The client uses the communication channel to send a request directly to the server.
- After recognizing the incoming request, the server executes the appropriate service.
- When the service execution is completed, the server sends the results back to the client.

Implementation

Step 1: The applications are separated into client and servers. The sending and receiving transaction logic code are defined.

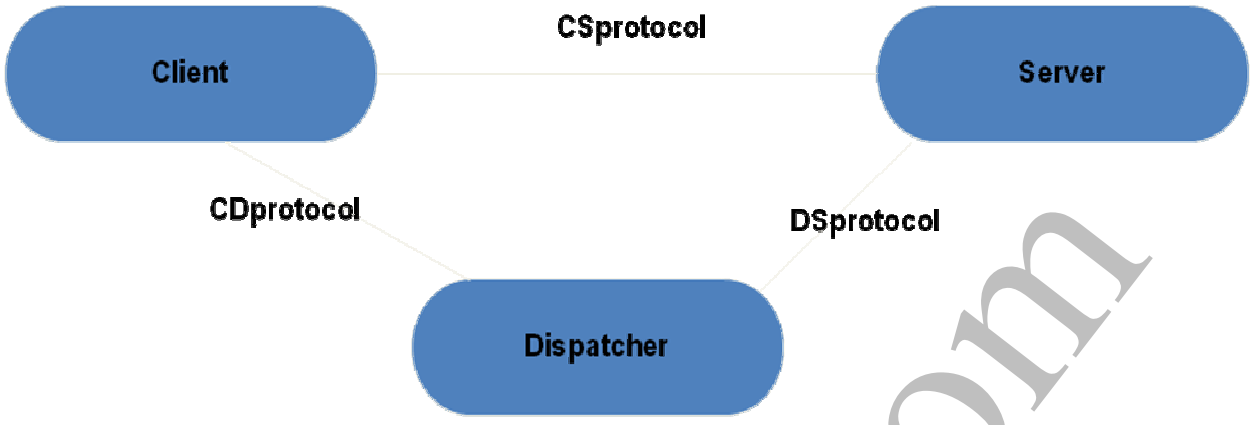
1. Partitioning applications into clients and server is predefined.
2. Clients may act as servers (vice versa), the roles of clients and servers are not predefined and may change at run-time.

Step 2: communication facilities that are required is decided.
Communication facilities are required for following interactions:

1. Interactions between clients and the dispatcher.
2. Interactions between servers and dispatcher.
3. Interactions between the clients and servers.

Different communication mechanism can be used for different interactions. Dispatcher and client will work shared memory if they reside on same machine. Servers and clients get connected using sockets if they reside on different machines. Servers and dispatcher get connected using sockets if they reside on distributed machines.
Same communication for all interactions will reduce the complexity of implementation but performance may decrease.

Step 3: Interaction protocols between components are specified.



The activities specified by protocol are initializing and maintaining a communication channel, structure of messages being transmitted and structure of data being transmitted.

DSprotocol specifies the following:

- How servers register with the dispatcher?
- The activities that are necessary to establish the communication channel to the server.

CDprotocol specifies the following:

- Interaction that occurs when a client requests dispatcher to establish a connection to server.
- On the failure of communication link, the dispatcher informs client.
- Dispatcher may try to establish a communication link several times before it reports an error.

CSprotocol specifies the following:

- The client sends a message to the server using the communication channel previously established between them.
- Both need to share common knowledge of syntax and semantics of messages they send and receive.

- The server receives the message, interprets it and invokes one of its services. After the service is completed, the server sends a return message to the client.
- The client extracts the service result from the message and continues with its task.

Naming of servers should follow some conventions. For instance, the names should uniquely identify the servers. The names should not carry any location information. Example: ID_SERVER_X OR ServerX. The location independent names are mapped to physical locations by the dispatcher.

Step 5: Dispatcher is designed and implemented.

Dispatcher is located within the address space of the client. Local procedure calls should be used for CDprotocol. For CSprotocol and DSprotocol TCP ports, sockets or shared memory should be used.

Constraints on communication mechanisms:

- The number of socket descriptors is constrained by the size of descriptor tables in the operating system.
- Each server may allocate its own socket. (limit for possible servers)
- Dispatcher should maintain internal message queues to save client requests.
- On the arrival of service requests, server opens a new socket and passes new socket descriptor to dispatcher.
- Dispatcher uses the socket descriptor and forwards information to client.
- After the interaction is completed, server closes socket descriptor.

The detailed structure of requests, the detailed structure of responses, the detailed structure of error messages, identification scheme for servers and pool of threads of dispatcher should be defined. The pool of threads to dispatcher will provide many clients to access many servers

using one dispatcher. It is a mechanism to handle many requests in parallel.

Step 6: The client and server components are implemented .

In one mechanism servers may register with dispatcher and in another servers dynamically register and unregister themselves.

Example resolved

TCP port number and the Internet address of the host machine are combined uniquely to identify the server. For instance clients requests as 'ABC/HTSERVER'. Message header is of fixed size and random amount of data is appended. Size and format of data, sender and receiver is provided in the message header. Messages are tagged with sequence numbers to enable to enable receiver of the message to recombine the incoming packets into correct order. Message includes service of request like : 'HTSERVER_DOC_RECEIVE, ANC.jpg'. Server determines the availability and sends message containing the ANC.jpg.

Variants

- Distributed dispatchers
- Broker architectural pattern
- Client-dispatcher-server with communication managed by clients
- Client-dispatcher-server with heterogeneous communication.
- Client-dispatcher-service

Distributed dispatchers can be used instead of single dispatcher.

A dispatcher receives a client request on a remote machine. Connection is established with remote dispatcher on the target node. Remote dispatcher initiates a connection with the requested server and sends the communication channel back to the first dispatcher (via this to client).
Clients can also communicate with directly with the dispatcher on the remote machine.

Client-dispatcher-server with communication managed by clients.

Instead of establishing a communication channel to servers, a dispatcher may only return the physical server location to the client. It is the responsibility of the client to manage all communication activities with the server.

Client-dispatcher-server with heterogeneous communication.

More than one communication mechanism between server and client can exist such as sockets or named pipes. Dispatcher is capable of supporting more than one communication mechanism. Server register itself with dispatcher and specifies the communication mechanism it supports.

Client -dispatcher -service

In this pattern clients address services and not servers. On the request, the dispatcher looks up which servers provide the specified service in its repository. If connection fails dispatcher access another server which provides the same servers.

The benefits achieved by client-dispatcher-server pattern are the following:

- Exchangeability of servers
- Change servers and add new servers without modifications to the dispatcher component. (un-register, register)
- Location and migration transparency
- Servers may be dynamically migrated to other machines, it does not imply any changes to clients.
- Re-configurations
- Start-up time and run-time can be specified.
- Fault tolerance
- On failure, new servers can be activated at a different network node without any impact on clients.

The bottle necks may be faced using client-dispatcher-server pattern are following:

- Lower efficiency is obtained through indirection and explicit connection establishment.
- Dispatcher's activity in locating and registering servers and explicitly establishing the connection will lead to overhead. The alternative is to get rid of dispatcher by hard-coding server location into the client. But we will be loosing the exchangeability of servers.
- Sensitivity to change in the interface of the dispatcher component, as the dispatcher plays the central role, the software system is sensitive to changes in the interface of the dispatcher.

Patterns	Addresses
Forward-Receiver	Encapsulation of communication facilities.
Client-Dispatcher-Server	Location Transparency.
Publisher-subscriber	Consistency between operating components

Client-dispatcher-service

```
class Dispatcher {
Hashtable registry = new Hashtable();
Random rnd = new Random(999);
public void register (String svc, Service obj)
{
Vector v = (Vector) registry.get(svc);
v.addElement(obj);
}
```

```
public Service locate(String svc){
Vector v = (Vector) registry.get(svc);
int I = rnd.nextInt() %v.size();
return (Service) v.elementAt(i);
}
}
```

All clients, servers and the dispatcher exist in same address space. An entry in the hash table is available for each service name. Each entry consists of the vector of all servers providing same kind of service. A server registers with the dispatcher by specifying a service name and new server instance. When a client asks the dispatcher for specific service, the dispatcher looks up all available servers in its repository. It randomly selects one of them and returns the server reference to the client.

```
abstract class Service {
String nameOfService;
String nameOf Server;
public Service(String avc, String srv){
nameOfService = svc;
nameOfServer = srv;
CDS.disp.register(nameOfservice, this);
}
abstract public void service();
}
```

The abstract class Service represents the available server objects. It registers server objects with the dispatcher automatically when the constructor is executed. Concrete server classes are derived from the abstract class Service. The derived Server classes have to implement the abstract method service.

```
class Client {
public  void doTask()
```


```
{
Service s;
try { s= CDS.disp.locate("….");
s.service();
}
```


```
catch  ….
}
```

Clients request the dispatcher for object references, then use these references to invoke the appropriate method implementation.

```
public class CDS {
public static Dispatcher disp = new Dispatcher();
public static void main (String args[ ]) {
Client client = new Client();
client.doTask();
}
}
```

The class CDS defines the main program of the application. Dispatcher is instantiated. Event loop of client is invoked.

Publisher-subscriber model

To keep the state of co-operating components synchronized. To achieve one-way propagation of changes. One publisher notifies any number of subscribers about changes to its state. One or more components must be notified about state changes in a particular component. The number and identities of dependent components is not known a priori or may even change over time. Explicit polling by dependents for new information is

not feasible. The publisher and its dependents should not be tightly coupled when introducing a change-propagation mechanism.

Solution-Publisher is a dedicated component. Subscribers are the components dependent on changes in the publisher. Publisher maintains the registry of subscribers. Publishers changed state is notified to all subscribers. Subscribers retrieve the changed data.

Implementation

- Publishers and subscribers shall be depicted by abstract base classes.
- Decision of which internal state to be notified for subscribers is made.
- An object can be a subscriber to many publishers.
- An object can take roles of a publisher and subscriber.
- Subscription and the ensuing notification is differentiated according to the event type. This allows subscribers to get needed messages
- The publisher send selected details of data change when it notifies its subscribers or can just send a notification and give subscribers the responsibility to find out what changed.

Push-model	Pull model
Publisher sends all changed data when it notifies the subscriber.	Publisher only sends the minimal information when sending a change notification.
The subscribers have no choice about if and when they want to retrieve the data.	Subscribers are responsible for retrieving the data they need.
Very rigid dynamic behavior. Model is not suitable for complex data changes.	Offers more flexibility by more messages between publisher and subscribers.

Even pushing a package description is a overhead.	Data changes are found. The process of finding data changes is organized as a decision tree.
This model is best suited when subscribers need the published information most of the time.	The model is used when only the individual subscribers can decide if and when they need a specific piece of information.

The above table depicts the two different implementations of publisher and subscriber patterns. The suitability has to be understood.
The variants of publisher – subscriber patterns are:

1. Gatekeeper
2. Event channel
3. Producer-consumer

Gatekeeper

This pattern is applied for distributed systems. Publisher instance in one process notifies remote subscribers. Publisher may alternatively spread over two processes. In the receiving process gatekeeper demultiplexes the messages by surveying the entry points to the process. Gatekeeper notifies subscribers when events for which they registered occur.

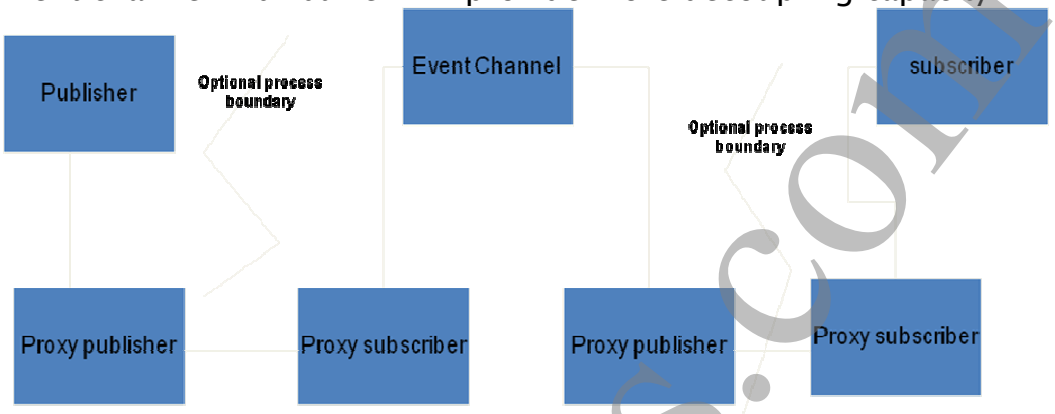
Event Channel

This pattern strongly decouples publishers and subscribers. It is to be noted that there can be more than one publisher. Subscribers wish to be notified about the occurrences of changes and not about identity of publisher. Publishers are not interested in which components are subscribing. An event channel is created and placed between the publisher and the subscribers. The event channel appears as subscriber for publisher (vice-versa).

Implementation

- A subscriber registers with event channel.
- Administration instance with proxy publisher is created.
- Proxy subscriber is created between a publisher and an event channel as shown in the figure 13 below.

- Connection is made over a process boundary with local proxy subscriber.
- Publisher, event channel and subscriber all exist in different processes.
- Event channel with buffer will provide more decoupling capacity.



-

Figure 13

When messages from a publisher arrive, the event channel does not have to notify the subscribers immediately. Several notification policies can be implemented. Several event channels can be chained. The event channel provide capabilities for quality-of-service such as filtering an event, sorting an event internally for a fixed period and sending event to all components that subscribe during that period. A chain assemble all the capabilities necessary for a system. A chain sums the capability of the individual event channels of which it is composed. The event channel variant is powerful enough to allow multiple publishers and typed events. Example: Two UNIX pipes.

Producer-consumer

- Producer supplies information.
- Consumer accepts information for further processing.
- Producer and consumer are strongly decoupled.
- A buffer is placed between them.
- The producer writes to the buffer without any regard for the consumer.
- The consumer reads data from the buffer at its own discretion.

- Synchronization is achieved by stating buffer overflow and underflow.
- The producer is suspended when the buffer is full.
- The consumer waits if the buffer is empty.

It should be noted that publisher-subscriber pattern provide many - many relationship for number of components. The producer-consumer pattern provide one-one relationship for number of components.

Event-channel + producer-consumer provide more than one producer or consumer. Several producers can provide data by only allowing them to write to the buffer in series.

If more than one consumer we can use iterators. When one consumer reads data from the buffer, the event -channel does not delete that data from the buffer, but only marks it as read by the consumer. The consumer is given the illusion that the data is consumed, and hence deleted, while other consumers will be given the illusion that the data is still present and unread. Each consumer has its own iterator on the buffer. The position of iterator on the buffer reflects how far the corresponding consumer has read the buffer. The data in the buffer is purged behind the lagging iterator, as all reads on it have been completed.

Idioms

Idioms represent low-level patterns. These are used to solve implementation specific problems in a programming language such as memory management in C++, object creation and use of library components. The collection of related idioms defines a programming style. The instances of programming style are type of loop statements, program element naming, source text formatting and choosing return values. We should understand that idioms ease communication among developers. Idioms speed up software development and maintenance process. These patterns provide us the library of styles used within the

organization. The global idioms are also available. Idioms define a programming style. The following table lists the salient features of the design patterns and idioms. This table is not to compare the two types of patterns. It provides us a clear understanding that, design patterns are application specific and Idioms are programming language specific.

Design patterns	Idioms
Medium-scale patterns. (smaller architectural units)	Low-level patterns.
Capture the static – dynamic roles & relationships in solutions that occur repeatedly . Address general structural principles.	Describe how to solve implementation-specific problems in a programming language.
The application of a design pattern has no effect on the fundamental structure of a software system.	Directly address implementation of a specific design pattern.
Independent of particular programming language. Portable between the programming languages.	Problems related to language. Less portable between the programming languages.

Idioms can address low-level problems related to the use of a language. Some of the low-level problems are naming program elements, source text formatting, choosing the return values, kind of looping statements, naming of the program elements and formatting of the source code. The literal and variable naming schemes can be with a particular style. The formatting specifications for coding with particular language can be defined. The best method of coding representation for decision statements can be provided.

Idioms approach or overlap areas that are typically addressed by programming guidelines. Idioms demonstrate competent use of programming language features. Idioms can therefore also support the teaching of a programming language.

The summary of what can idioms provide?

1. A collection of related Idioms defines a programming style.
2. There are always many ways to solve a particular programming problem with a given language.
3. Some might be considered better style to make better use of the available language features.
4. We have to know and understand the little tricks and unspoken rules that will make us productive and our code of high quality for a particular problem.

String copy function for C-style strings

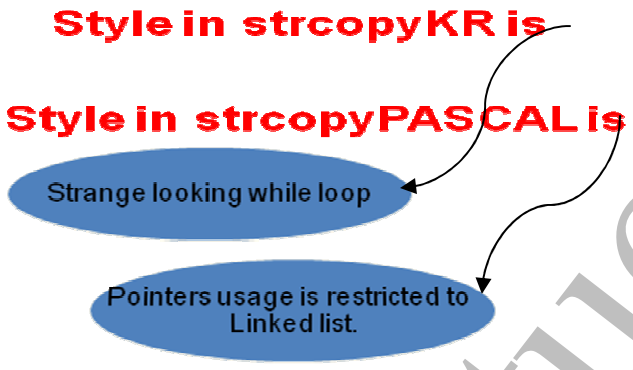
```
void strcopyKR(char *d, const char *s)
    {
      while(*d++ =  *s++);
    }
```

The above code in c-style depicts the copying mechanism from one array to another. C allows pointers to do the job. The while loop initialization and decision are done on same line of code. This is optimized Idiom for string copy. For any C programmer, doing the above function with pointers is very common. Array names act as pointers.

```
void strcopyPascal(char d[ ], const char s[ ])
    {
      int I; for (i=0; s[i] != '\0' ; i=i+1)
      {
            d[i] = s[i];
      }
    d[i] = '\0'; }
```

The above code in pascal depicts the copying mechanism from one array to another. In pascal, it is a practice that, pointers are used to implement dynamic structures only. The for loop initialization and decision are done on same line of code. This is optimized Idiom for string copy. For any pascal programmer, doing the above function with arrays is very common.

Even for an experienced programmer it is difficult to follow a consistent style. For instance a team should agree on a single coding style for programs.



Corporate style guides is an approach to achieve a consistent style throughout programs developed by teams. These use dictatorial rules like 'all comments must start on a separate line'. They give solutions or rules without stating the problem. Corporate style guides seldom give concrete advice to a programmer about how to solve frequently occurring coding problems. Style guides that contain collected idioms, not only give rules, but provide insight into the problem solved by a rule. Guide name the idioms and thus allow them to be communicated. For eg: Say "Use intention revealing selector here" and not "Apply rule 42"

An example of Style Guide idiom

Name: Indented Control Flow
Problem: How do you indent messages?
Solution:

Put zero or one argument messages on the same lines as their receiver.

```
foo isNil
2 + 3
a<b ifTrue: [...]
```

Put the keyword/argument pairs of messages with two or more keywords each on its own line, indented one tab

```
a<b
    ifTrue: [...]
    ifFalse: [...]
```

The following characteristics of Idioms provide us the tangible understanding of this interesting pattern:

- Different sets of idioms are appropriate for different domains.
- A single style guide is unsuitable for companies that employ many teams to develop applications in different domains.
- A style guide cannot and should not cover a variety of styles.
- A coherent set of idioms leads to a consistent style in programs- speeds up development and makes programs easier to understand.

Idioms are found as collection in language introduction. Some design patterns that address programming problem in a general way are provided in guides. Embedded idioms are formed by the perspective of a specific language.

Singleton design pattern : C++

Name: Singleton (C++)

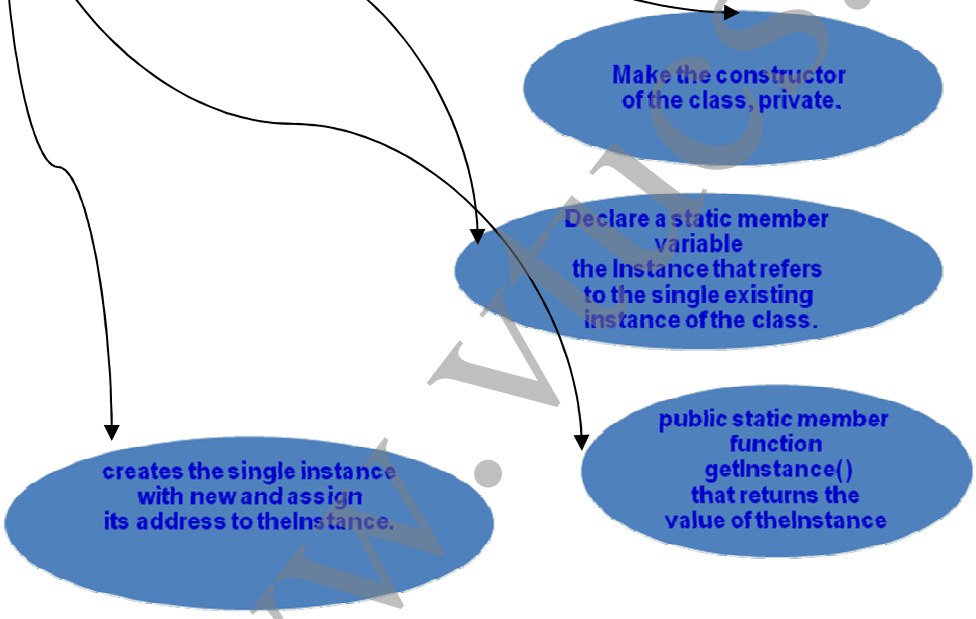
Problem: To ensure that exactly one instance of a class exists at run-time.

Solution:

- Make the constructor of the class private.
- Declare a static member variable the Instance that refers to the single existing instance of the class. Initialize this pointer to zero.
- Define a public static member function getInstance( ) that returns the value of theInstance.

- The first time getInstance( ) is called, which creates the single instance with new and assign its address to theInstance.

```
class Singleton {
Static Singleton *theInstance;
Singleton();
public:
Static singleton *getInstance() {
If (! theInstance)
theInstance = new Singleton;
return theInstance;
```




```
};
//…
Singleton*  Singleton :: theInstance = 0;

}
```

Smalltalk

Name: Singleton (smalltalk)

Problem: To ensure that exactly one instance of a class exists at run-time.

Solution:

- Override the class method new to raise an error.
- Add a class variable TheInstance that holds the single instance.
- Implement a class method getInstance that returns TheInstance. The first time getInstance is called.
- It will create the single instance with super new and assign it to theInstance.

New self error: 'cannot create new object'
getInstance
theInstance isNil ifTrue: [theInstance := super new].
^ theInstance
By now , it must be very clear that, we have achieved pattern mining, style guide concept and pattern language understanding.

Counted pointer

This pattern makes memory management of dynamically allocated shared objects in C++. Reference counter is updated in body class by handle objects. Clients access body class objects only through handles via the overloaded operator → ().

Under memory management issue, the shared object between clients lead to following problems: A client may delete the object while another client still holds a reference to it or all clients may not use the object, without the object being deleted. Once the objects goes out of scope, it should be deleted. The objects here is service provided by the server. The same service can be accessed by many clients. clients may forget their reference to service object, and garbage gets collected.

Context: memory management of dynamically allocated instances of a class.

Without references or pointers to service objects, the objects can be used with pass objects by value. In this compiler will destroy value objects that go out of scope. If objects are large, copying each time is expensive at run-time and memory consumption is also more. Passing objects by values in applications involving dynamic structures such as trees will not be suitable. By storing an object in several collections deliberately required.

Following are the problem forces:

- Passing objects by value is inappropriate for a class.
- Several clients may need to chare the same object.
- Reference to an object that has been deleted should be avoided. (Dangling pointer)
- If a shared object is no longer needed, it should be destroyed to conserve memory.
- The memory should be released to other resources. Solution should not require too much additional code within each client.

Solution

The class of the shared objects is called Body. The body is extended with a reference counter. To keep track of references used, a second class handle is the allowed to hold references to body objects. Handle class deals with Body object's reference counter. The handle objects is used syntactically to the pointers of Body objects by overloading operator → ( ) in the handle class.

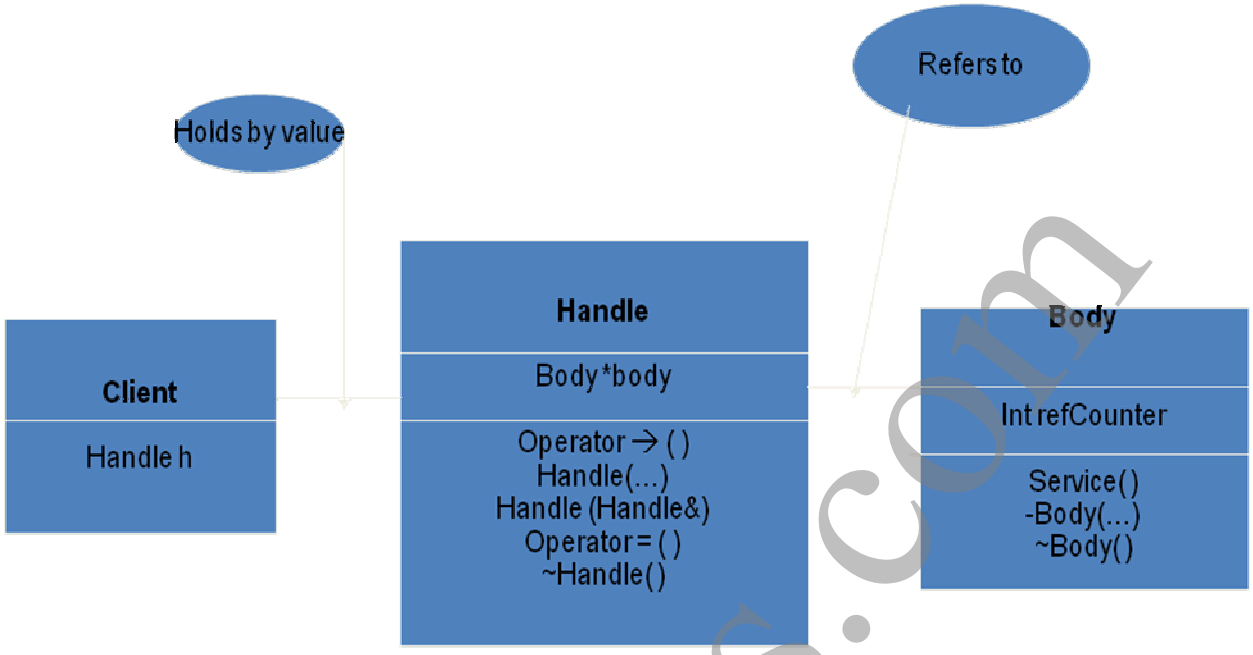


Figure 14

In the above figure 14, it can be noted that the service object is accessed via the object reference created in the handle class. But the handle object is created by pass by value in client. Once the handle object is out of scope, the reference or the dangling pointer to the service object is also deleted.

Implementation

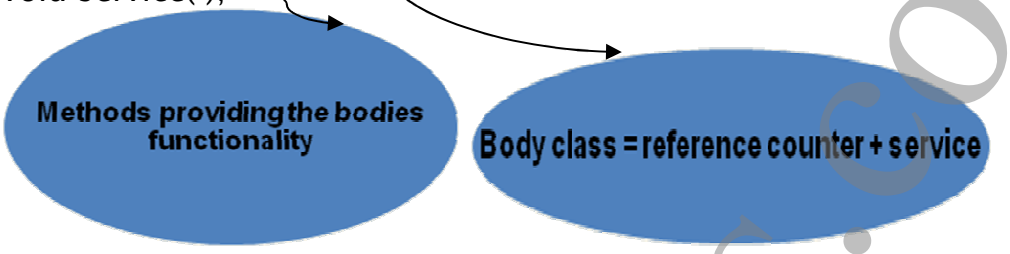
- The constructors and destructor of the body class is declared private (or protected) to prohibit its uncontrolled instantiation and deletion.
- The Handle class is made a friend to Body class to provide Handle class with access to Body's internals.
- The Body class is extended with a reference counter.
- A single data member to the Handle class that points to the Body object is added.

The Handle class copy constructor and its assignment operator is implemented by copying the Body object pointer and incrementing the reference counter of the shared Body object. The destructor of the Handle class will decrement the reference counter and delete the Body object when the counter reaches zero. The following public member function in the Handle class is implemented:

Body & operator → ( ) const {return body;}

The Handle class is extended with one or several constructors that create the initial Body instance to which it refers. Each of these constructors initialises the reference counter to one.

```
class Body {
public:
void service( );
```

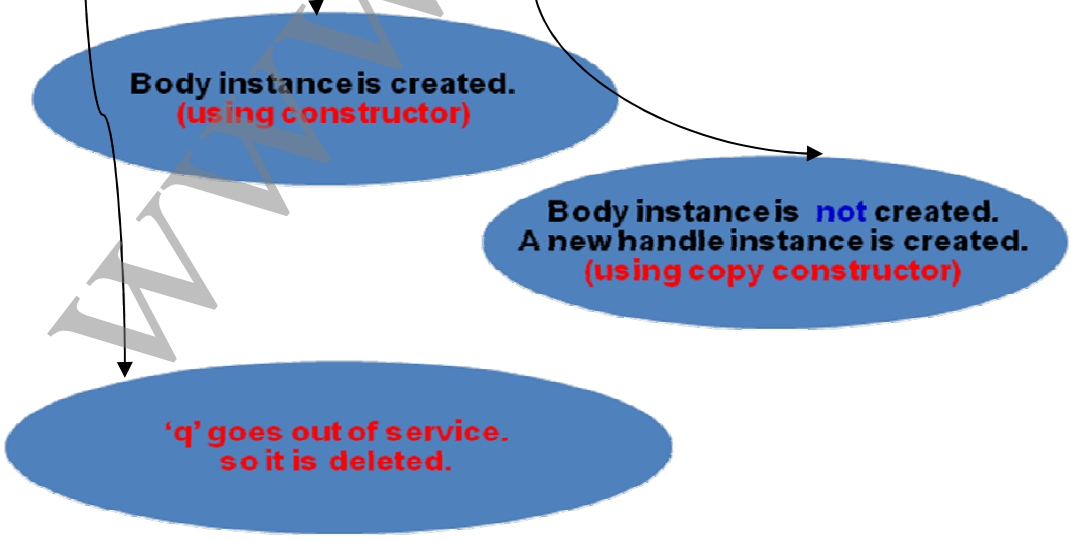



```
private:
friend class Handle;
Body (…){…}
~Body ( ) {…}
int refcounter;
};
```

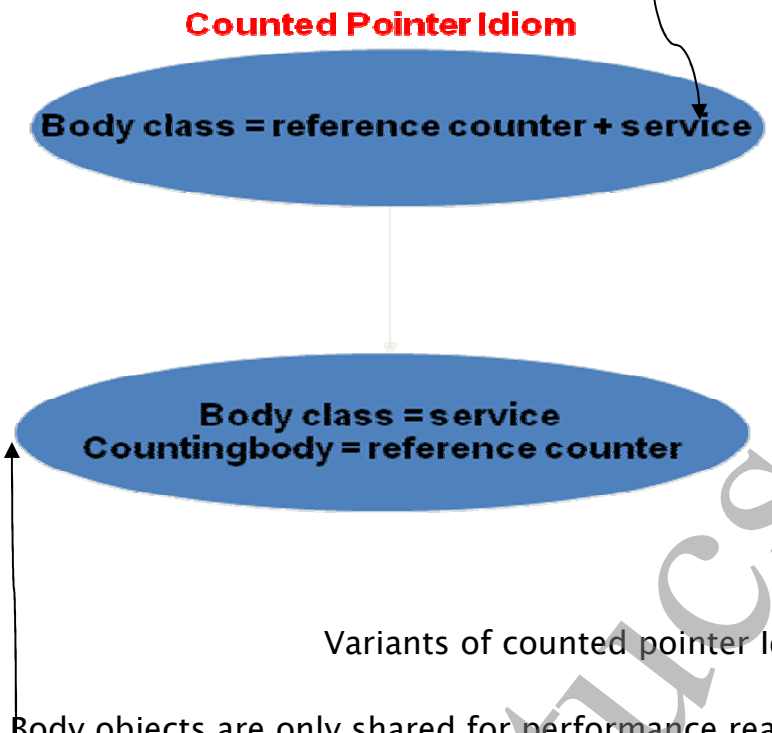
In the above pictorial representation it should be clear that the service object and reference counter are declared in the Body class only. This should be noted to understand the variants further.

```
class Handle {
public:
Handle(…){
body = new Body(…);
body->refCounter = 1;
}
Handle(const Handle &h)
{
body = h.body;
body →refCounter++;
}
```

```
Handle &operator = (const Handle &h)
{
h.Body → refCounter++;
if(--body →refCounter) <=0)
delete body;
body = h.body;
return *this;
}
~Handle( ){
if (--body→refCounter <= 0)
   delete body;
}
Body* operator →( )
{return body;}
Private:
Body *body;
};
Handle h(...);
{
 Handle q(h);  h->service();   q->service(); }
h->service;
```



Drawback of counted Pointer Idiom is that we need to change the Body class to introduce the reference counter.



Variants of counted pointer Idiom

Body objects are only shared for performance reasons.

1. The client has the illusion of using its own Body object, even if it is shared with other clients.
2. Any change in the shared Body object, the Handle creates a new Body instance and uses this copy for all further processing.
3. It is not sufficient to just overload operator → ( ). The interface of the Body class is duplicated by the Handle class.
4. Methods that would change the Body object create a new copy of it, if other clients share this Body object.

James Coplien variant

In cases where the Body class is not intended to have derived classes, it is possible to embed it in the Handle class using this variant. To wrap the existing classes with a reference counter class. This wrapper class then forms the Body class of the Counted Pointer Idiom. Additional level of indirection is required when clients access Body object as shown in figure 15.

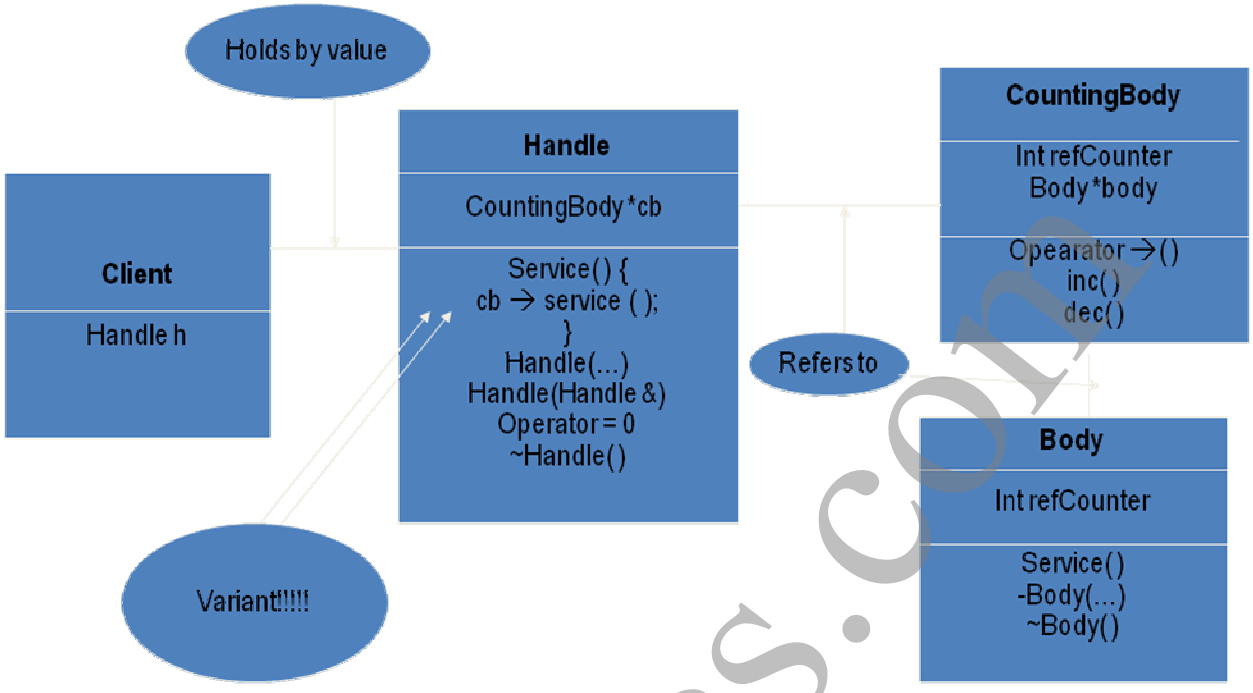


Figure 15

Andrew Koenig variant.

Separate abstraction for use counts are created. The Handle holds the two pointers : To the body object and to use count object. The use count class can be used to implement handles of variety of body classes. The Handle objects require twice the space of the other counted pointer variants. The access is as direct as with a change to the Body class as shown in Figure 16.

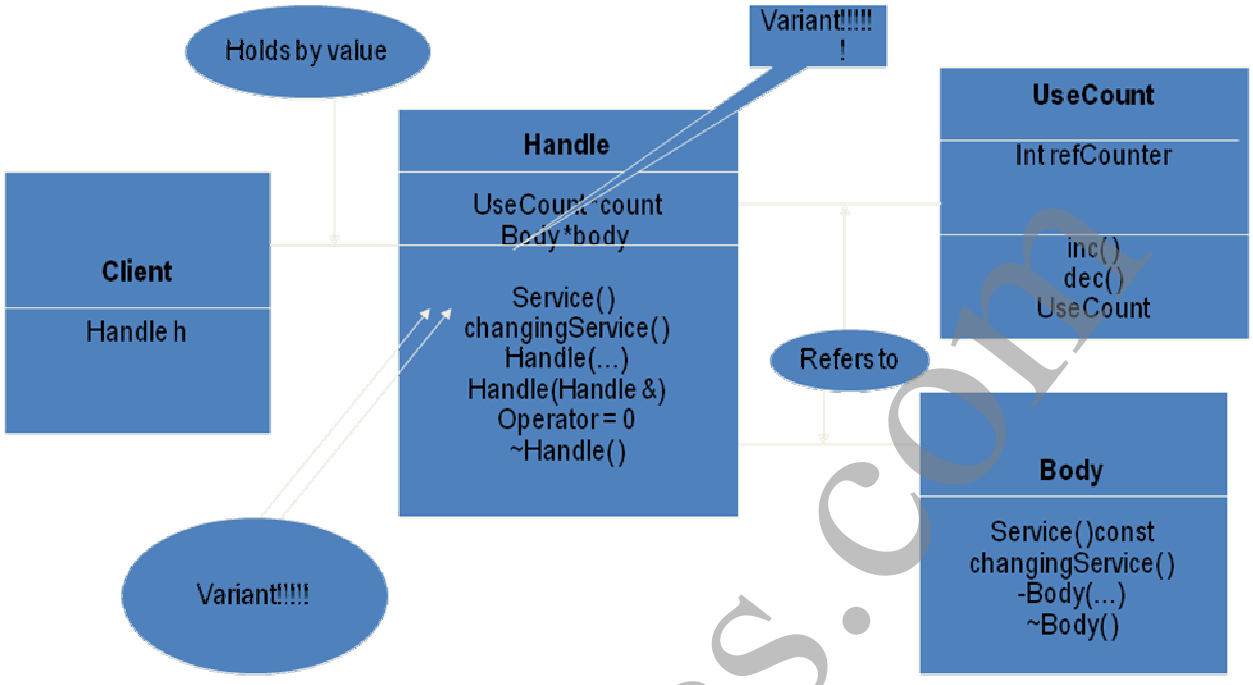


Figure 16

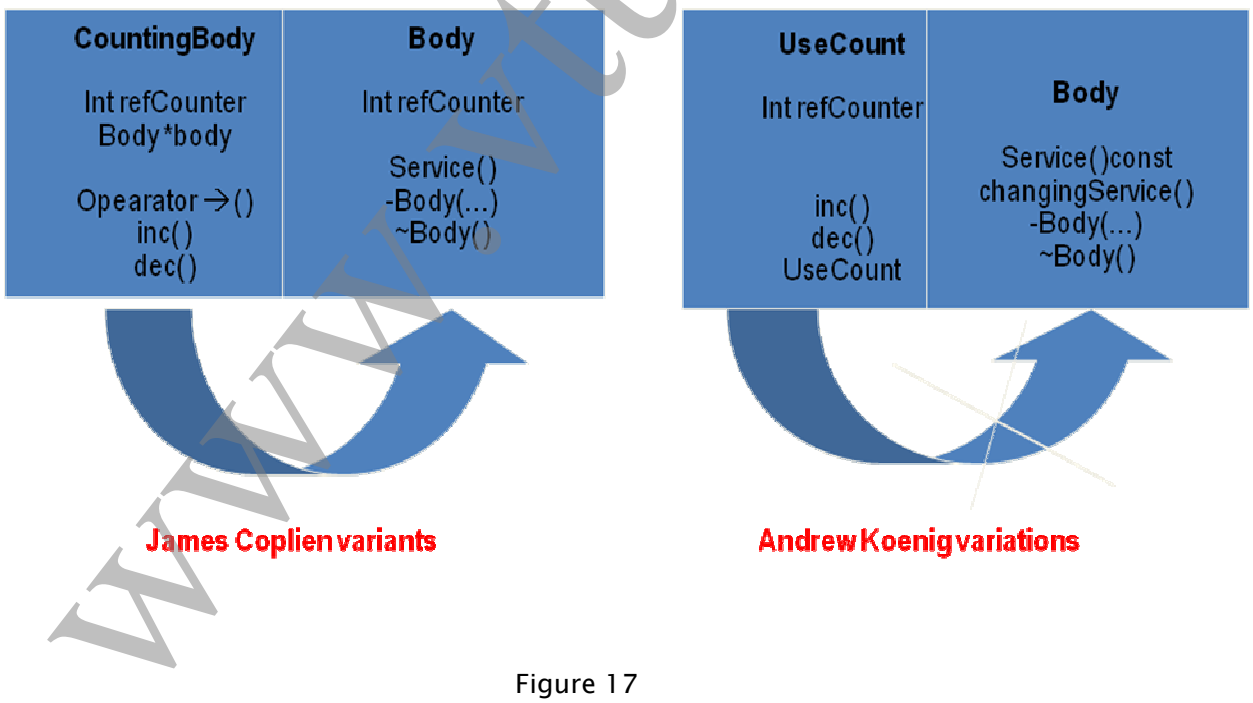


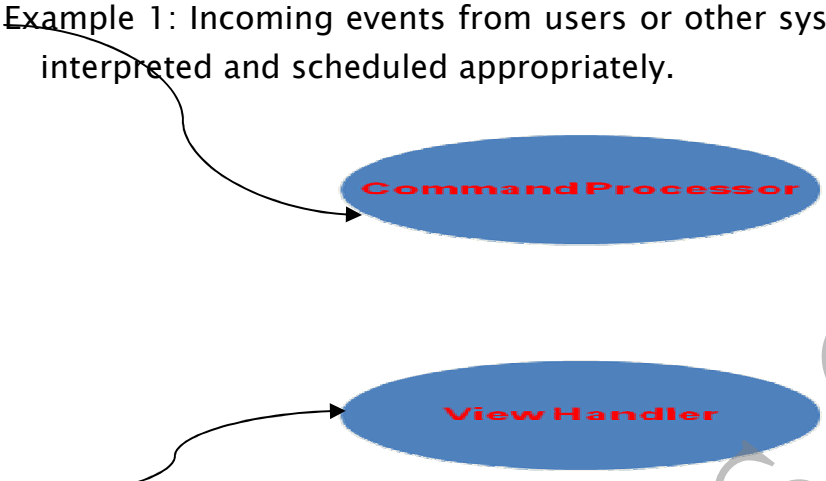
Figure 17

- We can observe from figure 17 that, the reference to the Body was created in counting body pattern but not in the use count pattern.

Management patterns

These patterns handle homogeneous collections of objects.

Example 1: Incoming events from users or other systems, which must be interpreted and scheduled appropriately.



Example 2: When interactive systems must present application-specific data in a variety of different ways. Such views must be handled appropriately, both individually and collectively.

Command Processor separates the request for a service from its execution. This pattern manages requests as separate objects and schedules the execution process. This pattern provides services for storing of request objects for later undo.

Example: A text editor to enable the undoing of multiple changes.

The user interface offer several means of interaction (keyboard inputs or pop-up menus).

Context: Applications that provide services related to the execution of user functions such as scheduling or undo.

Problem forces:

- Different users like to work an application in different ways.
- Enhancements of the application → not break existing code.
- Additional services as undo should be implemented consistently for all requests.

Solution: Whenever a user calls a specific function of the application, the request is turned into a command object.

To achieve the above solution, the following patterns can be used for implementation.

1. Command pattern
2. Command processor pattern
3. Supplier pattern
4. Controller pattern

The abstract command component defines the interface of all command objects. The command processor schedules the execution of commands, store them for later undo and log the sequence of commands for testing purpose. Each command object delegates the execution of its task to supplier components within the functional core of the application.

Structure :

The abstract command component defines the interface of all command objects. For each user function a command component is derived from the abstract command.

- The controller represents the interface of the application. It accepts requests such as paste text and creates the corresponding command objects.
- The command processor schedules the commands and starts execution. It saves already-performed commands for later undo.
- The supplier components provide the functionality required to execute commands as shown in the Figure 18. Supplier provides a means to save and restore command's internal state for an undo mechanism.

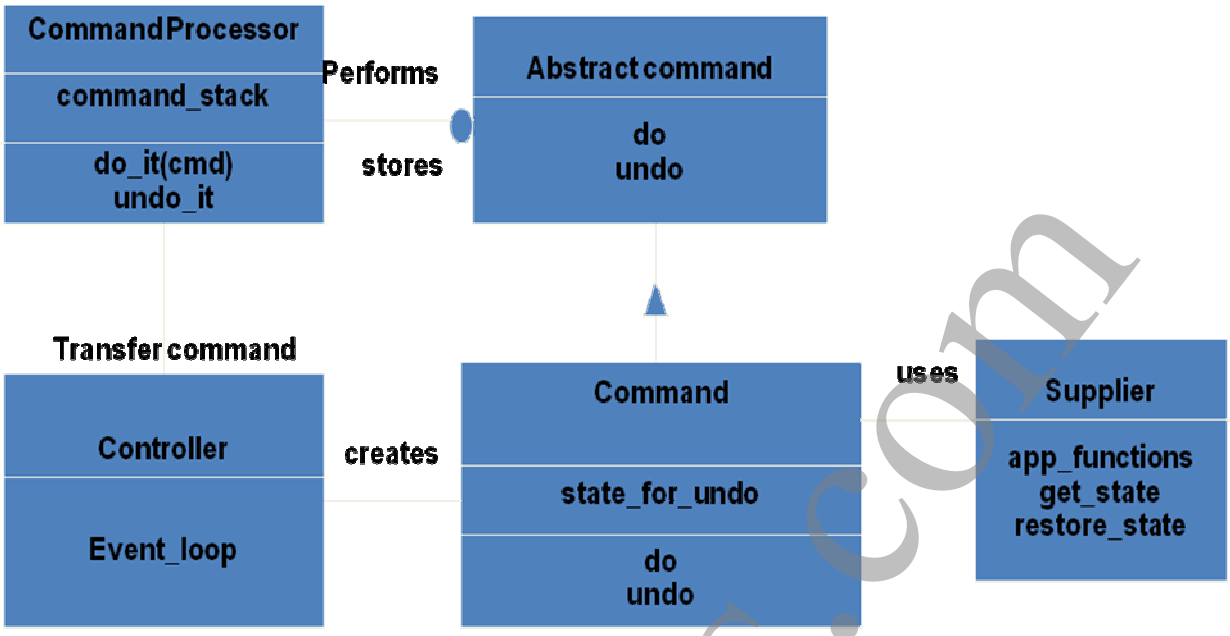


Figure 18

Dynamics

A request to capitalize a selected word arrives, is performed and then undone. For this the following transactions should happen.

1. The controller accepts the request from the user within its event loop and creates a 'capitalize' command object.
2. The controller transfers the new command object to the command processor for execution and further handling.
3. The command processor activates the execution of the command and stores it for later undo.
4. The capitalize command retrieves the selected text from its supplier, stores the text and its position in the document. The supplier actually capitalize the selection.
5. After accepting an undo request, the controller transfers request to the command processor. The command processor invokes the undo procedure of the most recent command.
6. The capitalize command resets the supplier to the previous state, by replacing the saved text in its original position.
7. If no further activity is required or possible of the command, the command processor deletes the command object.

Implementation

Step 1: The interface of the abstract command is defined.

For undo mechanism three types of commands are specified. They are no change command, normal command and no undo command. No change command is a command that requires no undo for instance cursor movement. The normal command is a command that can be undone. No undo command is a command that cannot be undone and prevents the undo of previously performed normal commands.

Step 2: The command components for each type of request that the application supports are designed.

The supplier component can be hard-coded within the command or the controller can provide the supplier to the command constructor as a parameter. Undoable commands store the state of their supplier for later undo without violating encapsulation.

Step 3: Flexibility is increased by providing macro commands that combine several successive commands.

Step 4: Controller component is implemented.

Example: menu controller contains a command prototype object for each menu entry and passes a copy of this object to the command processor whenever the user selects the menu entry. User-defined menu extensions are possible with macro-commands.

Step 5: Access to the additional services of the command processor are implemented.

Additional service is implemented by specific command class. The command processor supplies the functionality of 'do' method.

Step 6: Implement the command processor component.

The command processor receives command objects from the controller and takes the responsibility for them. For each command object, the command processor starts the execution by calling the do method.

Variants can be achieved by spreading controller functionality or with combination of interpreter pattern. Spreading controller functionality means the role of controller is distributed over several components. i.e the role of the controller is not restricted to components of the graphical user interface. Combination with interpreter pattern is a scripting language that provides the programmable interface to an application. The parser component of the script interpreter takes the role of controller.

Benefits achieved are:

- Flexibility in the way requests are activated: Different user interface elements for requesting a function will generate the same kind of command object.
- Flexibility in the number and functionality of requests: The controller and command processor are implemented independently of the functionality of individual commands.
- The central command processor allows the addition of services related to command execution.
- The command processor is an ideal entry for application testing.
- The command processor design pattern allows commands to be executed in separate threads of control.

Liabilities are:

Due to decoupling of the components, the additional indirection costs storage and time. Number of command classes increase.

- On the basis of abstractions commands should be grouped.
- Combination of low-level commands should be achieved by pre-programmed macro-command objects.
- The event-handling mechanism should deliver events to different destinations such as controller and some activated command object.

View Handler

To manage all the views that a software system provides, view handler is used. A view handler component allows clients to open, manipulate and dispose views. This pattern co-ordinates dependencies between views and organizes their update.

Examples:

1. Several documents to be worked on simultaneously.
2. Several independent views of the same document.

Efficient update mechanism for propagating changes between windows.

Context: A software system that provides multiple views of application-specific data Or that supports working with multiple documents.

Problem:

Managing multiple views should be easy from the users perspective and also for client components within the system. Implementations of individual views should not depend on each other or be mixed with the code used to manage views. View implementation vary and additional types of views are added during the life time of the system.

Solution: The view handler pattern adopts the idea of separating presentation form functional core, as proposed by the model-view-controller system by itself. It only removes the responsibility of managing the entirety of views and their mutual dependencies from the model and view components.

Besides creation and deletion of windows, the view handler offers functions such as to bring a specific window to the foreground, to clone the foreground window, to tile all open windows so that they do not overlap, to split individual views into several part, to refresh all views, to clone views and to provide several views of the same document.

The view handler offers functions for closing views, both individual ones and currently opened views, as is needed when quitting the application

Structure:

An abstract view component define an interface that is common to all views. The view handler uses this interface for creating, coordinating and closing views. Specific view components are derived from the abstract view and implement its interface. In addition, each view implements its own display function. Supplier components provide the data that is displayed by view components. Suppliers offer an interface that allows clients to retrieve and change data as shown in Figure19.

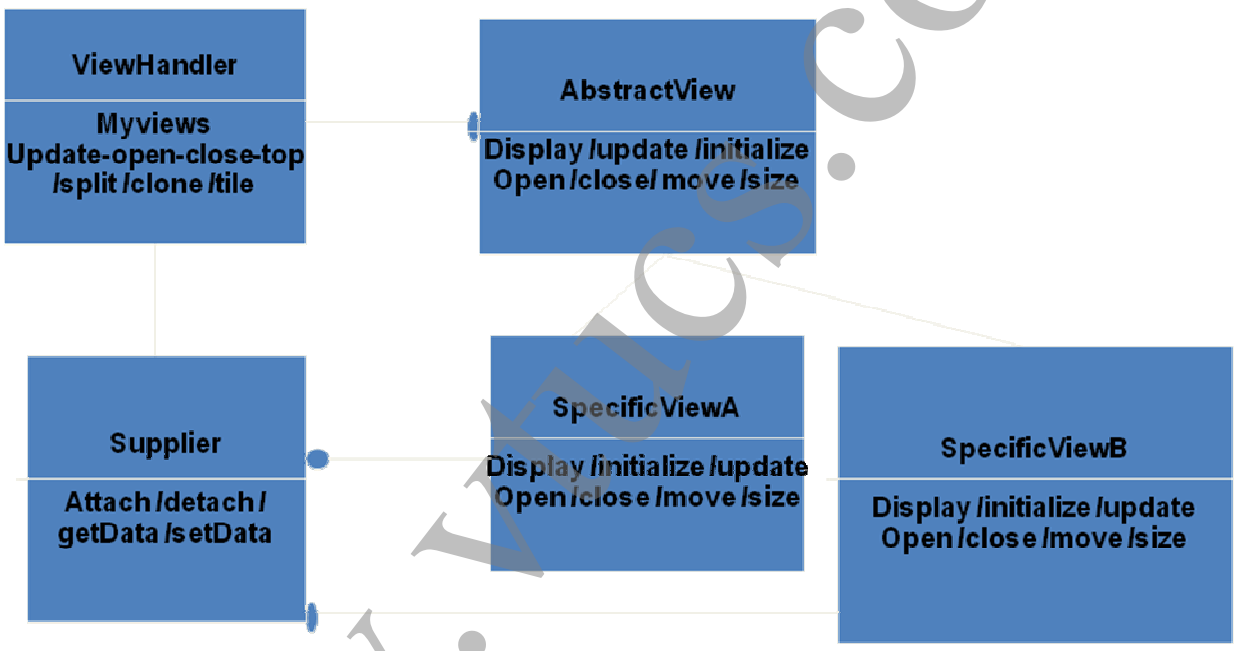


Figure 19

Dynamics

Scenario –I

- A client- which may be the user or another component of system, calls the view handler to open a particular view.
- The view handler instantiates and initializes the desired view. The view registers with the change-propagation mechanism of its supplier.
- The view handler adds the new view to its internal list of open views.
- The view handler calls the view to display itself.
- The view opens a new window, retrieves data from its supplier, prepares data for display.

Scenario II

- The user invokes the command to tile all open windows.
- For every open view, the view handler calculates a new size and position and calls its resize and move procedures.
- Each view changes its position and size, sets the corresponding clipping area and refreshes the image.
- Views cache the image they display. If this is not the case, views must retrieve data from their associated suppliers before redisplaying themselves.

Implementation

Views are identified. The common interface is specified for all views.
Views are implemented. View handler is defined.

Variants of view handler pattern

1. View handler with command objects.

   This variant uses command objects to keep the view handler independent of specific view interfaces. Instead of calling view functionality directly, the view handler creates an appropriate command and executes it. The command itself knows how to operate on the view. For example, we can specify a tile command, that when executed, first calls the size and then the move function of a view.

2. Another option is to create commands and pass them to a command processor which takes care of their correct execution.

Benefits of view-handler pattern

Uniform handling of views is achieved. All views share a common interface. Extensibility and changeability of views are achieved. The organization of view components in an inheritance hierarchy. Application-specific view co-ordination is provided. Since views are managed by a central instance, it is possible to implement specific view co-ordination strategies.

Liabilities: Due to restricted applicability, system must support many different views, views with logical dependencies between each other or views which can be configured with different suppliers or output devices. A level of indirection between clients that want to create views and also within chain of propagation of change notifications is introduced.

References

1. C.Alexander: The timeless way of building, oxford university press. 1979 (as stated in first chapter of ref 3 book).
2. A. Newell, H A Simon: human problem solving, prentice-hall, 1972 (as stated in first chapter of ref 3 book).
3. Pattern-oriented software Architecture- A system of patterns, Volume – I, by Frank Buschmann .et. al