UNIT - 7 DESIGN PATTERNS – 1:

Syllabus : - 6hrs

- **What is a pattern**
- **what makes a pattern?**
- **Pattern categories;**
- **Relationships between patterns;**
- **Pattern description.**
- **Communication Patterns:**
- **Forwarder-Receiver;**
- **Client-Dispatcher-Server;**
- **Publisher-Subscriber.**

Patterns

❖ Patterns help you build on the collective experience of skilled software engineers.

❖ They capture existing, well-proven experience in software development and help to promote good design practice.

❖ Every pattern deals with a specific, recurring problem in the design or implementation of a software system.

❖ Patterns can be used to construct software architectures with specific properties

What is a Pattern?

▪ Abstracting from specific problem-solution pairs and distilling out common factors leads to patterns.

▪ These problem-solution pairs tend to fall into families of similar problems and solutions with each family exhibiting a pattern in both the problems and the solutions.

Definition :

The architect Christopher Alexander defines the term pattern as

❖ Each pattern is a three-part rule, which expresses a relation between
a certain context,
a problem, and
a solution.

▪ As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.

- As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.

- The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing. And when we must create it. It is both a process and a thing: both a description of a thing which is alive, and a description of the process which will generate that thing.

Properties of patterns for Software Architecture

- ❖ A pattern addresses a recurring design problem that arises in specific design situations, and presents a solution to it.
- ❖ Patterns document existing, well-proven design experience.
- ❖ Patterns identify & and specify abstractions that are above the level of single classes and instances, or of components.
- ❖ Patterns provide a common vocabulary and understanding for design principles
- ❖ Patterns are a means of documenting software architectures.
- ❖ Patterns support ***the*** construction of software with defined properties.
- ❖ Patterns help you build complex and heterogeneous software architectures
- ❖ Patterns help you to manage software complexity

Putting all together we can define the pattern as:

Conclusion or final definition of a Pattern:
A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.

What Makes a Pattern?

Three-part schema that underlies every pattern:

Context: a situation giving rise to a problem.

Problem: the recurring problem arising in that context.

Solution: a proven resolution of the problem.

Context:

- The Contest extends the plain problem-solution dichotomy by describing the situations in which the problems occur
- Context of the problem may be fairly general. For eg: ―developing software with a human-computer interface". On the other had, the contest can tie specific patters together.
- Specifying the correct context for the problem is difficult. It is practically impossible to determine all situations in which a pattern may be applied.

Problem:

- This part of the pattern description schema describes the problem that arises repeatedly in the given context.
- It begins with a general problem specification (capturing its very essence what is the concrete design issue we must solve?)
- This general problem statement is completed by a set of forces
- Note: The term ‗force denotes any aspect of the problem that should be considered while solving it, such as
  - Requirements the solution must fulfill
  - Constraints you must consider
  - Desirable properties the solution should have.
- Forces are the key to solving the problem. Better they are balanced, better the solution to the problem

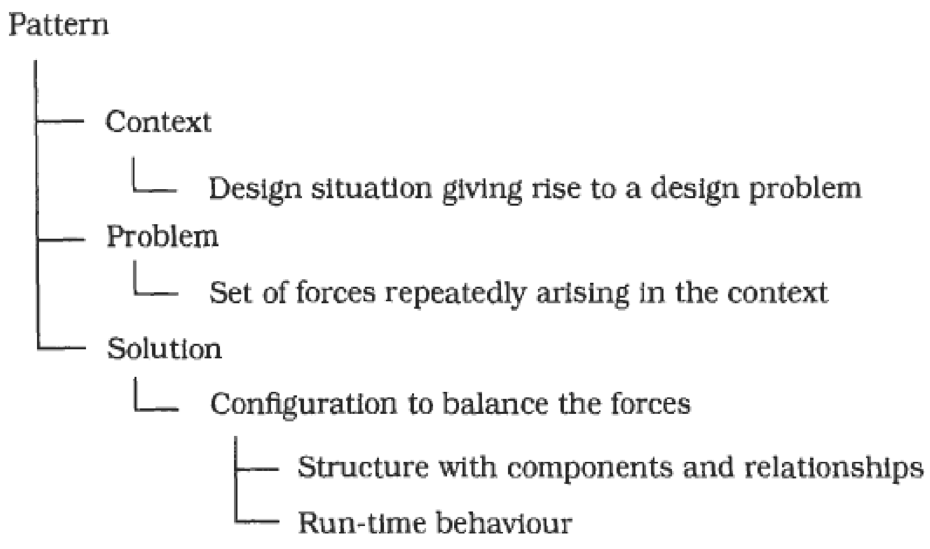
Solution:

- The solution part of the pattern shows how to solve the recurring problem(or how to balance the forces associated with it)
- In software architectures, such a solution includes two aspects:

Every pattern specifies a certain structure, a spatial configuration of elements. This structure addresses the static aspects of the solution. It consists of both components and their relationships.

Every pattern specifies runtime behavior. This runtime behavior addresses the dynamic aspects of the solution like, how do the participants of the patter collaborate? How work is organized between then? Etc.

- The solution does not necessarily resolve all forces associated with the Problem.
- A pattern provides a solution schema rather than a full specified artifact or blue print.
- No two implementations of a given pattern are likely to be the same.
- The following diagram summarizes the whole schema.


Pattern
Context
Design situation giving rise to a design problem
Problem
Set of forces repeatedly arising in the context
Solution
Configuration to balance the forces
Structure with components and relationships
Run-time behaviour

Pattern Categories

we group patterns into three categories:

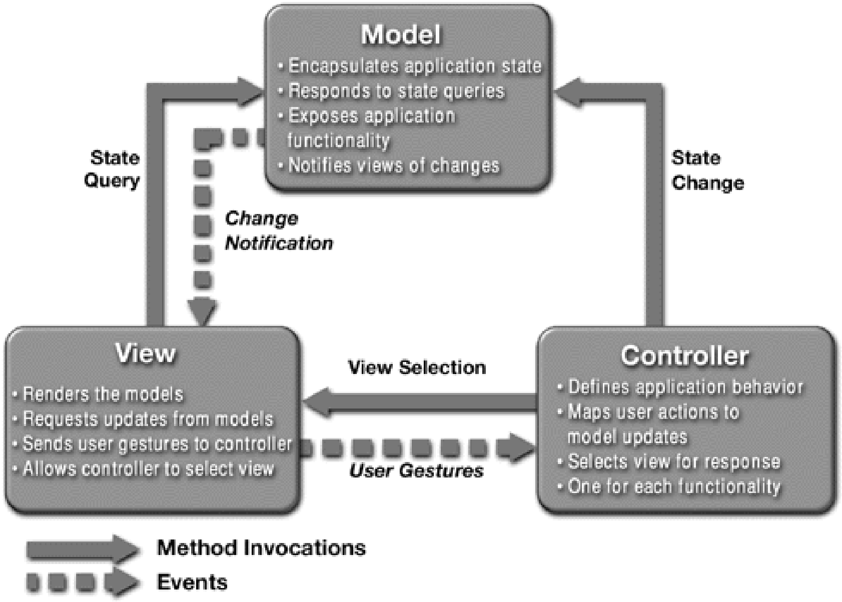
- Architectural patterns
- Design patterns
- Idioms

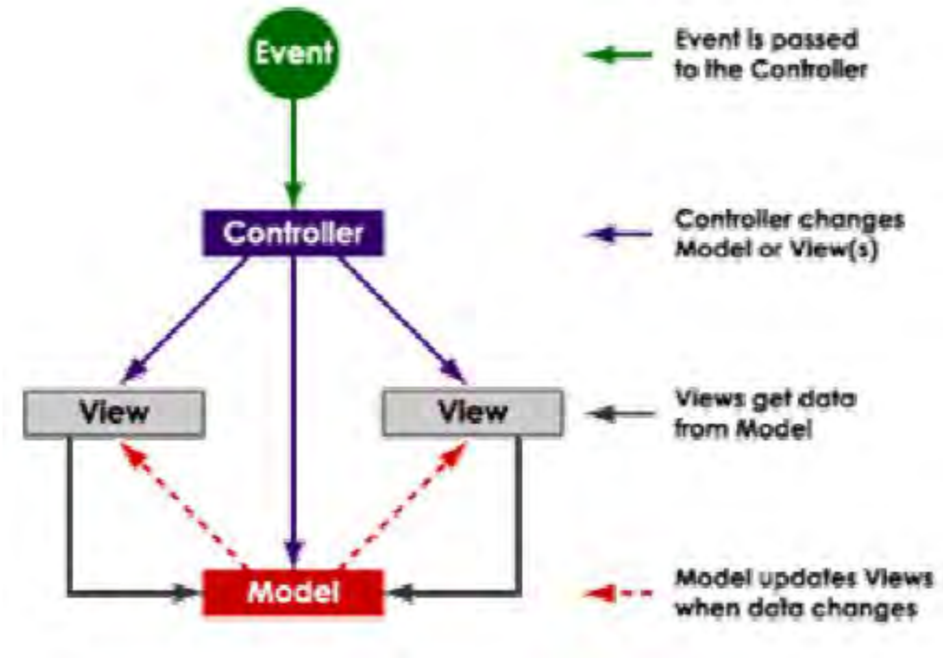
Each category consists of patterns having a similar range of scale or abstraction.

Architectural patterns

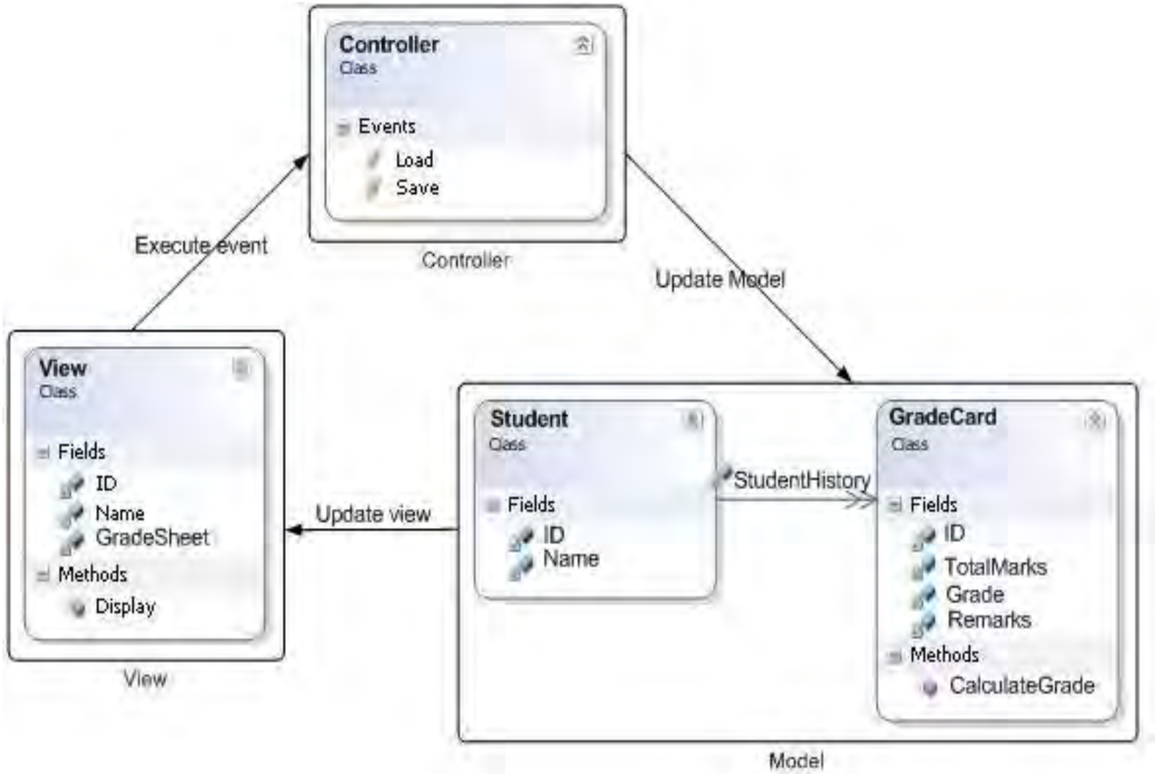
- Architectural patterns are used to describe viable software architectures that are built according to some overall structuring principle.
- Definition: An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- Eg: Model-view-controller pattern.

Structure→


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events


Event
Controller
View
View
Model
Event is passed to the Controller
Controller changes Model or View(s)
Views get data from Model
Model updates Views when data changes

Eg:



Design patterns

- Design patterns are used to describe subsystems of a software architecture as well as the relationships between them (which usually consists of several smaller architectural units)
- Definition: **A** design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them.It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular Context.
- They are medium-scale patterns. They are smaller in scale than architectural patterns, but tend to be independent of a particular programming language or programming paradigm.
- Eg: Publisher-Subscriber pattern.

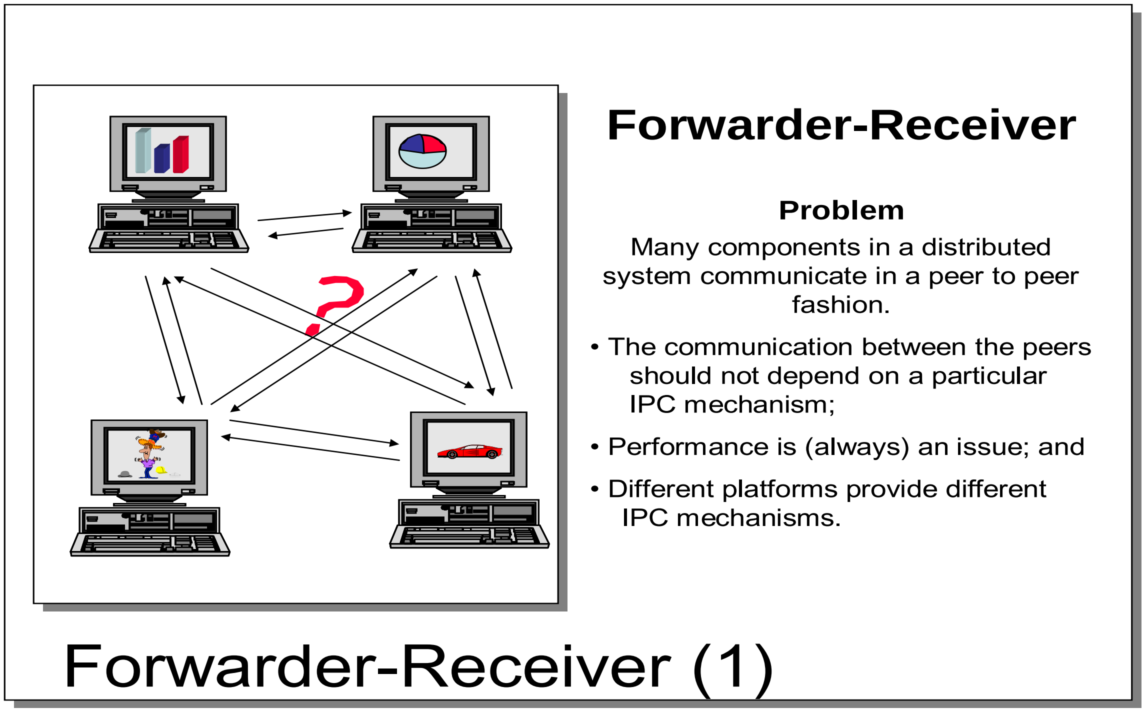
Idioms

- Idioms deals with the implementation of particular design issues.
- Definition: ***An idiom*** is a low-level pattern specific to a programming language. ***An*** idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.
- Idioms represent the lowest- level patterns. They address aspects of both design and implementation.
- Eg: counted body pattern.

Pattern description (see text book for description)

- **Name :** The name and a short summary of the pattern
- **Also known as:** Other names for the pattern, if any are known
- **Example :** A real world example demonstrating the existence of the problem and the need for the pattern
- **Context :** The situations in which the patterns may apply
- **Problem :** The problem the pattern addresses, including a discussion of its associated forces.
- **Solution :** The fundamental solution principle underlying the pattern
- **Structure :** A detailed specification of the structural aspects of the pattern, including CRC – cards for each participating component and an OMT class diagram.
- **Dynamics :** Typical scenarios describing the run time behavior of the pattern
- **Implementation:** Guidelines for implementing the pattern. These are only a suggestion and not a immutable rule.
- **Examples resolved**: Discussion for any important aspects for resolving the example that are not yet covered in the solution , structure, dynamics and implementation sections.
- **Variants:** A brief description of variants or specialization of a pattern
- **Known uses:** Examples of the use of the pattern, taken from existing systems
- **Consequences:** The benefits the pattern provides, and any potential liabilities.
- **See Also:** References to patterns that solve similar problems, and the patterns that help us refine the pattern we are describing.

Communication pattern:



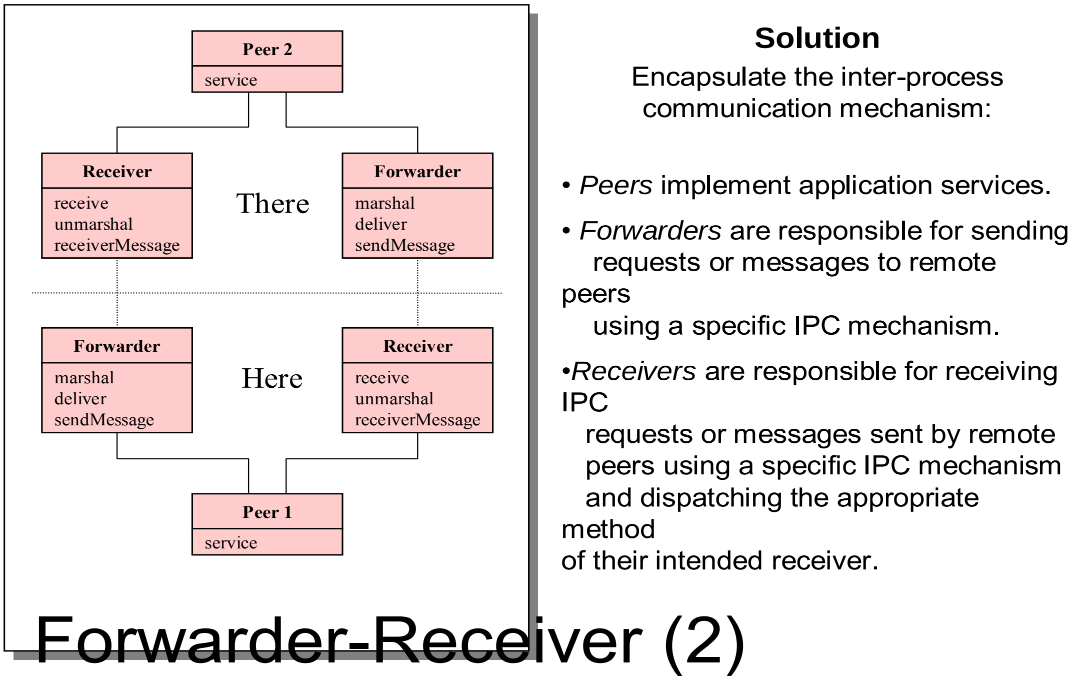
Forwarder-Receiver

Problem

Many components in a distributed system communicate in a peer to peer fashion.

- The communication between the peers should not depend on a particular IPC mechanism;
- Performance is (always) an issue; and
- Different platforms provide different IPC mechanisms.

Forwarder-Receiver (1)



Peer 2
service

Receiver
receive
unmarshal
receiverMessage

There

Forwarder
marshal
deliver
sendMessage

Forwarder
marshal
deliver
sendMessage

Here

Receiver
receive
unmarshal
receiverMessage

Peer 1
service

Solution

Encapsulate the inter-process communication mechanism:

- *Peers* implement application services.
- *Forwarders* are responsible for sending requests or messages to remote peers using a specific IPC mechanism.

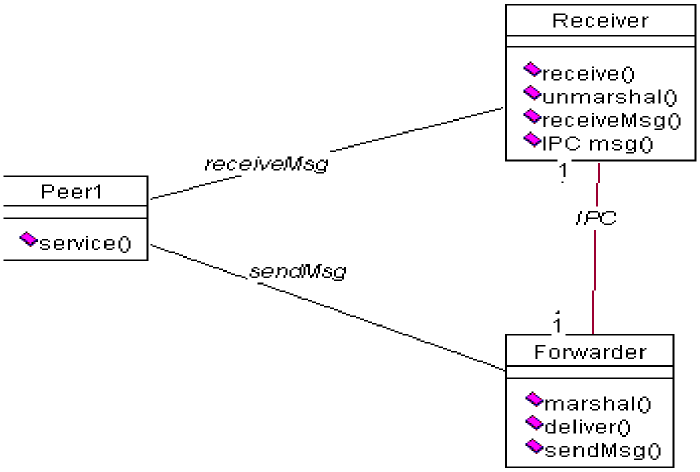
•*Receivers* are responsible for receiving IPC requests or messages sent by remote peers using a specific IPC mechanism and dispatching the appropriate method of their intended receiver.

Forwarder-Receiver (2)

- **Intent**
  - "The Forwarder-Receiver design pattern provides transparent interprocess communication for software systems with a peer-to-peer interaction model.

- It introduces forwarders and receivers to decouple peers from the underlying communication mechanisms."
- **Motivation**
- Distributed peers collaborate to solve a particular problem.
- A peer may act as a client - requesting services- as a server, providing services, or both.
- The details of the underlying IPC mechanism for sending or receiving messages are hidden from the peers by encapsulating all system-specific functionality into separate components. Examples of such functionality are the mapping of names to physical locations, the establishment of communication channels, or the marshaling and unmarshaling of messages.

Structure



- F-R consists of three kinds of components, Forwarders, receivers and peers.
- Peer components are responsible for application tasks.
- Peers may be located in different process, or even on a different machine.
- It uses a forwarder to send messages to other peers and a receiver to receive messages form other peers.
- They continuously monitor network events and resources, and listen for incoming messages form remote agents.
- Each agent may connect to any other agent to exchange information and requests.

- To send a message to remote peer, it invokes the method sendmsg of its forwarder.
- It uses marshal.sendmsg to convert messages that IPC understands.
- To receive it invokes receivemsg method of its receiver to unmarshal it uses unmarshal.receivemsg.
- Forwarder components send messages across peers.
- When a forwarder sends a message to a remote peer, it determines the physical location of the recipient by using its name-to-address mapping.
- Kinds of messages are
- Command message- instruct the recipient to perform some activities.
- Information message- contain data.
- Response message- allow agents to acknowledge the arrival of a message.
- It includes functionality for sending and marshaling
- Receiver components are responsible for receiving messages.
- It includes functionality for receiving and unmarshaling messages.

Dynamics

- P1 requests a service from a remote peer P2.
- It sends the request to its forwarder forw1 and specifies the name of the recipient.
- Forw1 determines the physical location of the remote peer and marshals the message.
- Forw1 delivers the message to the remote receiver recv2.
- At some earlier time p2 has requested its receiver recv2 to wait for an incoming request.
- Now recv2 receives the message arriving from forw1.
- Recv2 unmarshals the message and forwards it to its peer p2.
- Meanwhile p1 calls its receiver recv1 to wait for a response.
- 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 delivers it recv1.
- Recv1 receives the response from p2, unmarshals it and delivers it to p1.

Implmentation

- Specify a name to address mapping.-/server/cvramanserver/…..
- Specify the message protocols to be used between peers and forwarders.-class message consists of sender and data.
- Choose a communication mechanism-TCP/IP sockets
- Implement the forwarder.- repository for mapping names to physical addresses-desitination Id, port no.

  sendmsg(dest, marshal(the mesg))
- Implement the receiver – blocking and non blocking

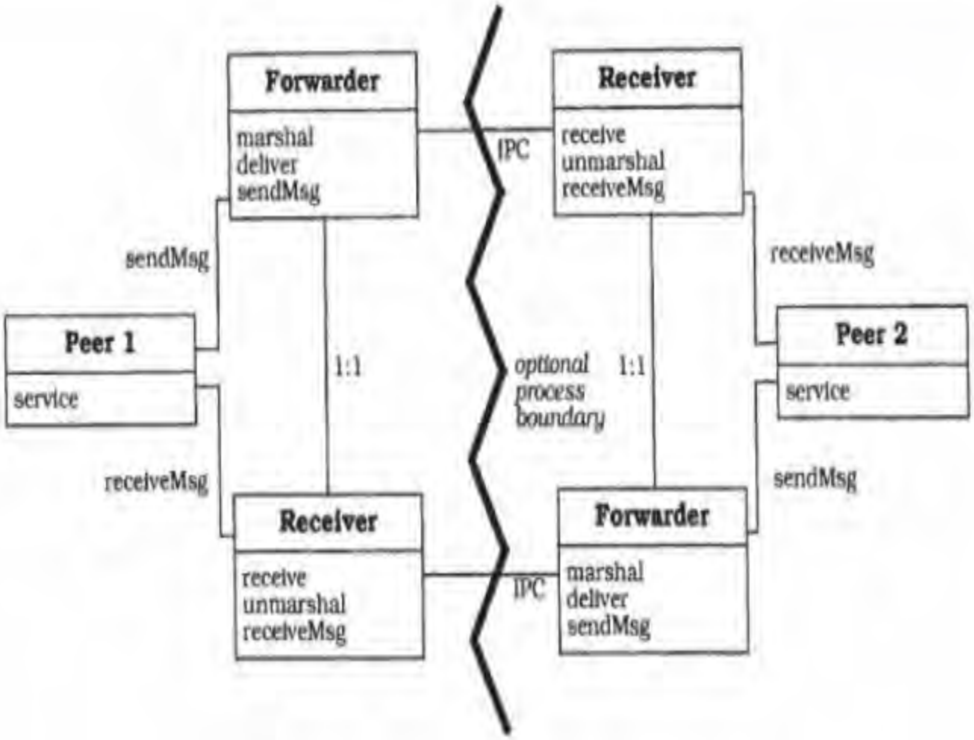
  recvmsg() unmarshal(the msg)
- Implement the peers of the application – partitioning into client and servers.

- Implement a start up configuration- initialize F-R with valid name to address mapping

Benefits and liability

- Efficient inter-process communication
- Encapsulation of IPC facilities

- No support for flexible re-configuration of components.
- **Known Uses**
- This pattern has been used on the following systems: TASC, a software development toolkit for factory automation systems, supports the implementation of Forwarder-Receiver structures within distributed applications.
- Part of the REBOOT project uses Forwarder-Receiver structures to facilitate an efficient IPC in the material flow control software for flexible manufacturing.
- ATM-P implements the IPC between statically-distributed components using the Forwarder-Receiver pattern..)
- In the Smalltalk environment BrouHaHa, the Forwarder-Receiver pattern is used to implement interprocess communication.


Forwarder
marshal
deliver
sendMsg
Receiver
receive
unmarshal
receiveMsg
IPC
sendMsg
receiveMsg
Peer 1
service
1:1
optional
process
boundary
1:1
Peer 2
service
receiveMsg
sendMsg
Receiver
receive
unmarshal
receiveMsg
IPC
Forwarder
marshal
deliver
sendMsg