- 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 Context 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 hand, the context can tie specific patterns 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:

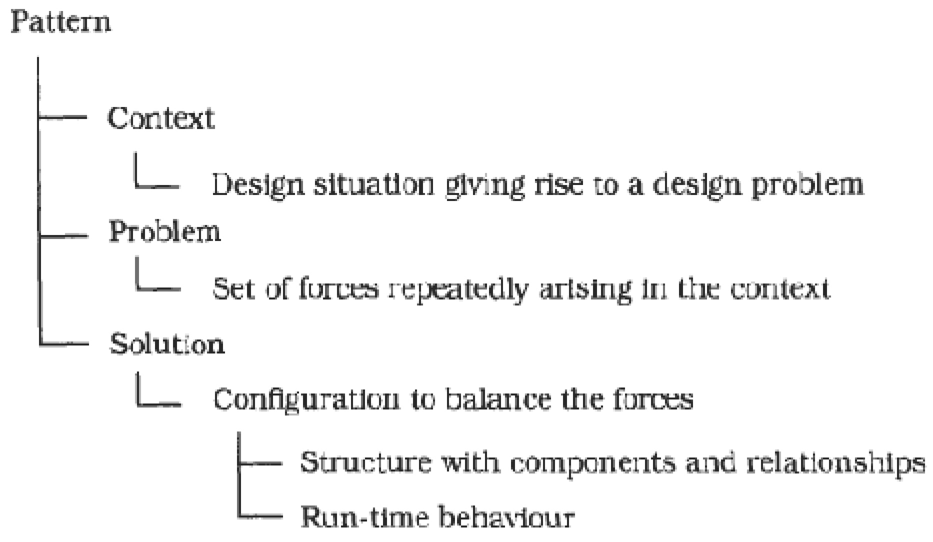
Every pattern specifies a certain structure, a spatial configuration of elements. This structure addresses the static aspects of the solution. It consists of both components and their relationships.

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:

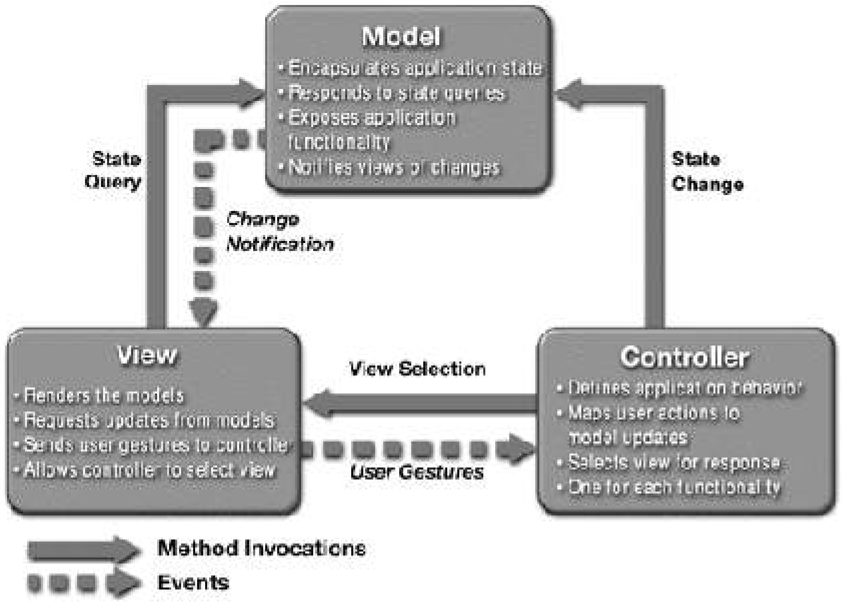
- Architectural patterns
- Design patterns
- Idioms

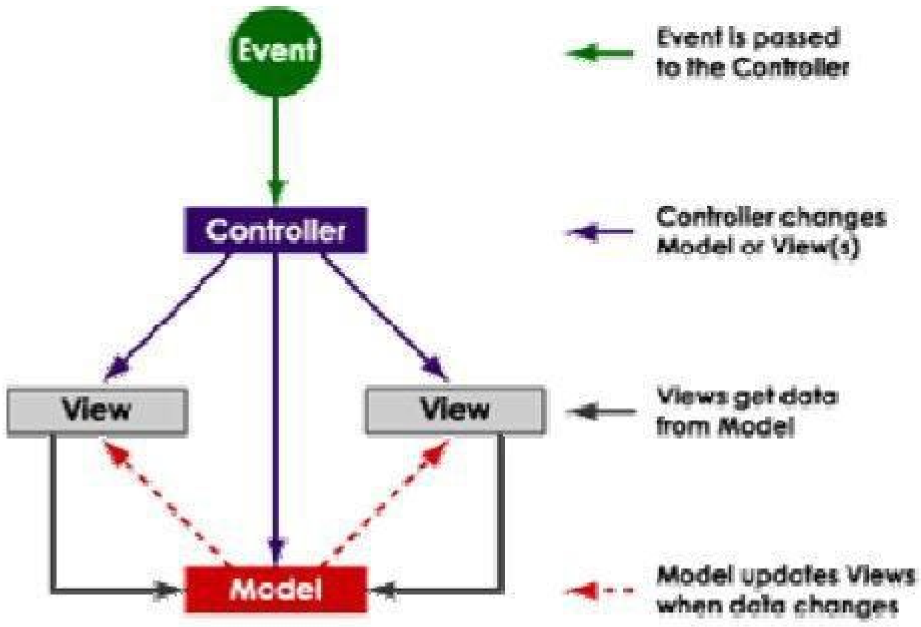
Each category consists of patterns having a similar range of scale or abstraction.

<u>Architectural patterns</u>

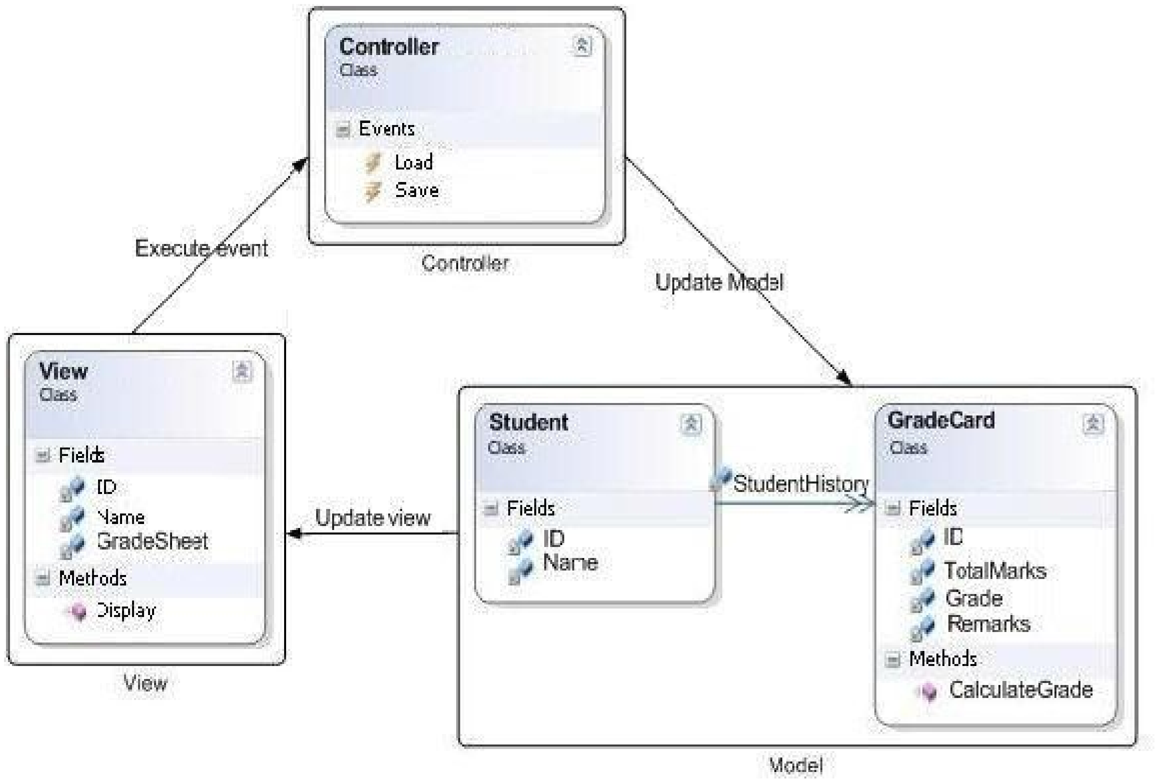
- Architectural patterns are used to describe viable software architectures that are built according to some overall structuring principle.
- Definition: An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, 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:



<u>Design patterns</u>

- 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.

<u>Idioms</u>

- 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

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.

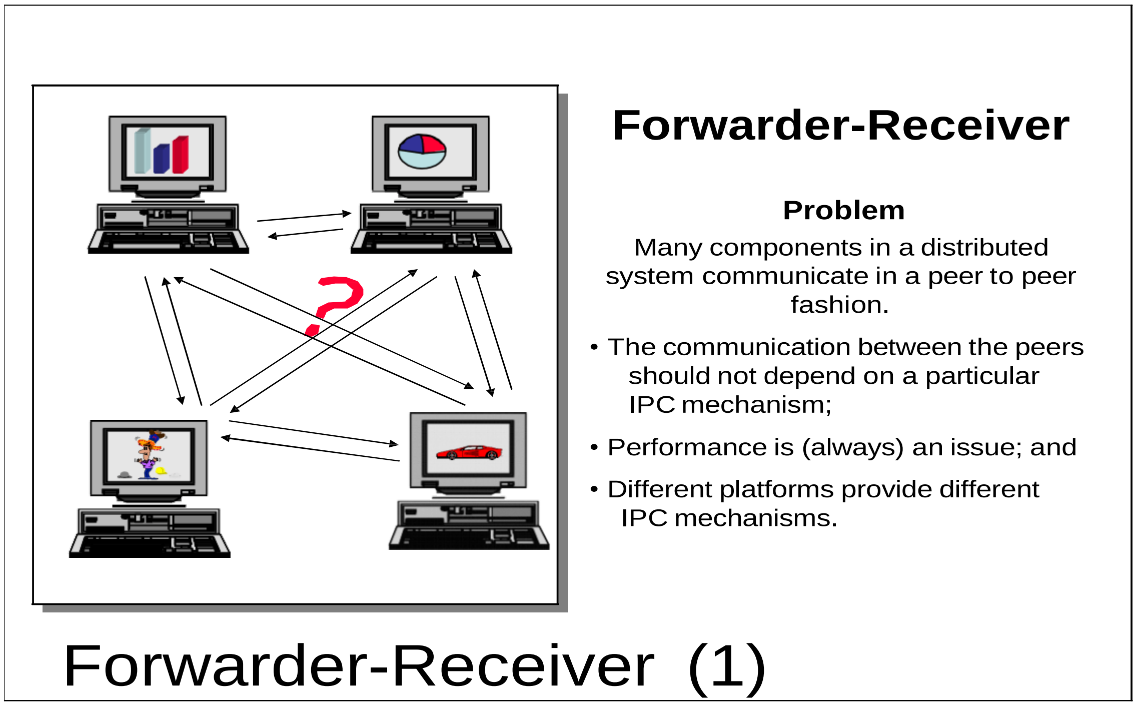
Communication pattern:

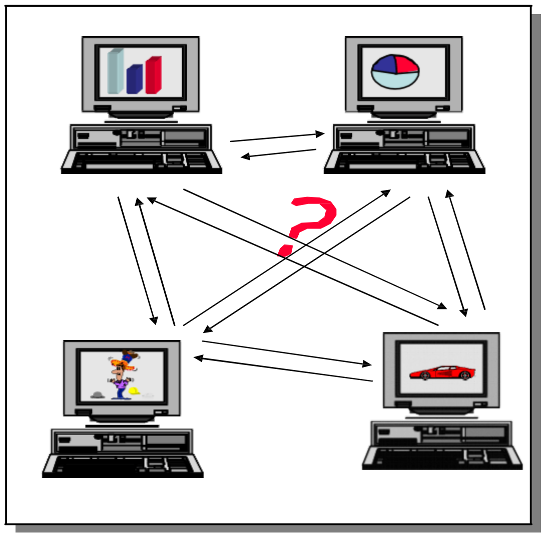
Most of the today's software systems run on distributed systems. These distributed systems need a means for communication.

- Problems:
    - Many communication mechanisms to choose from.
    - The use of communication facilities is often hard-wired into existing applications, leading to various problems.
        - Difficult to change the communication mechanism later.
        - Portability

- Migration of sub systems from one network node to another is only possible if the communication facility allows it.

- Solution:
  - Loosen the coupling between components of a distributed system and the mechanism it uses for communication, eg: by using
    - Encapsulation
    - Location transparency
- We discuss two patterns that addresses these topics:
  - The **Forwarder – Receiver** design pattern (provides encapsulation)
  - The **Client – Dispatcher – Server** design pattern (provides location transparency)
- Keeping cooperating component consistent is another problem in communication. We discuss one pattern that addresses this issue:
  - The **Publisher – Subscriber pattern**





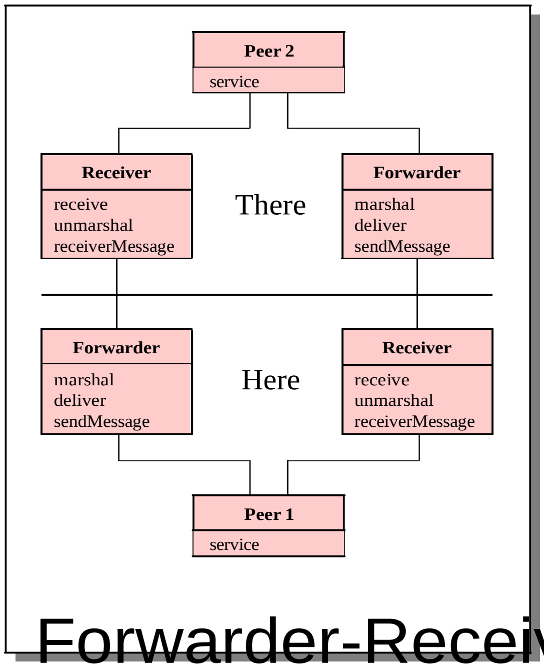
Forwarder-Receiver

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)



Solution

Encapsulate the inter-process communication mechanism:

• *Peers* implement application services.

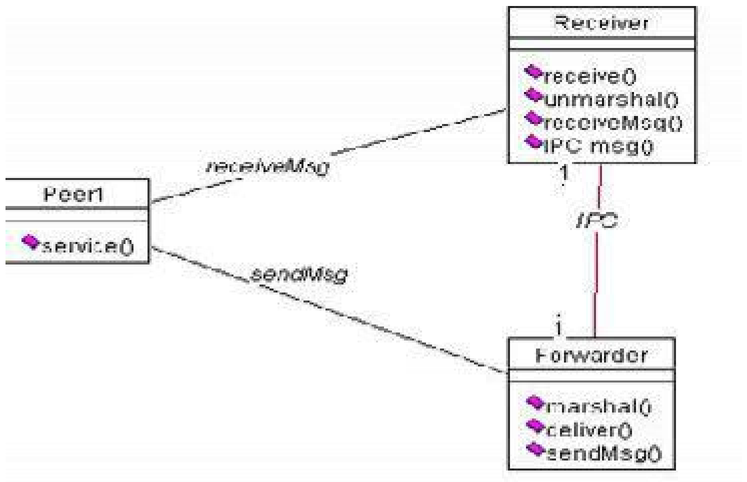
• *Forwarders* are responsible for sending requests or messages to remote peers using a specific IPC mechanism.

•*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

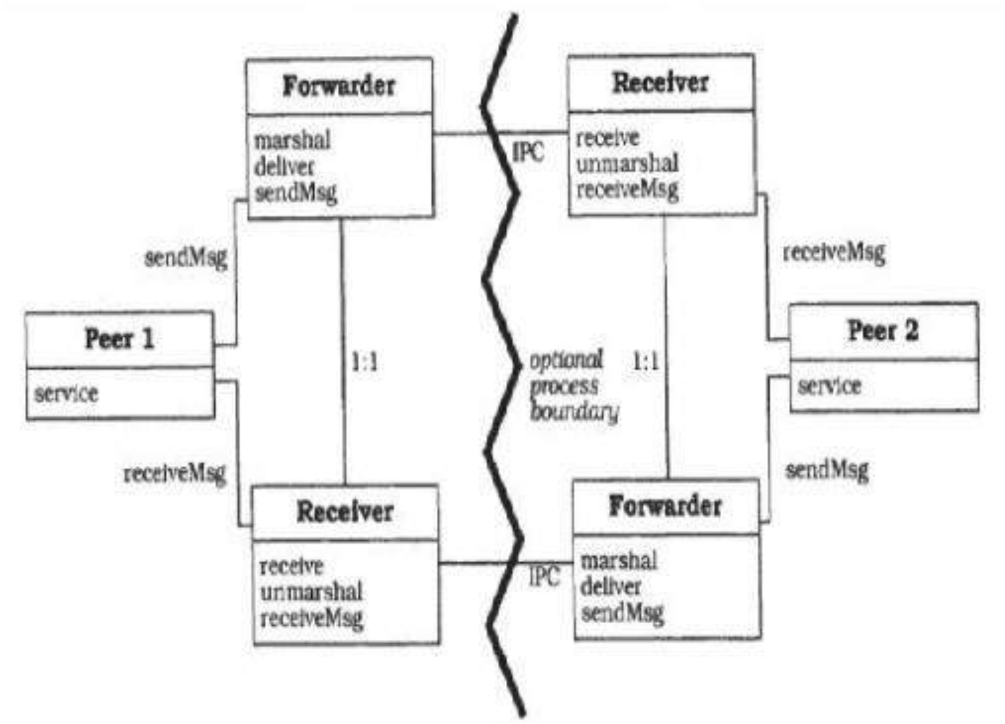
  recvmsg() unmarshal(the msg)
- Implement the peers of the application – partitioning into client and servers.
- Implement a start up configuration- initialize F-R with valid name to address mapping

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.

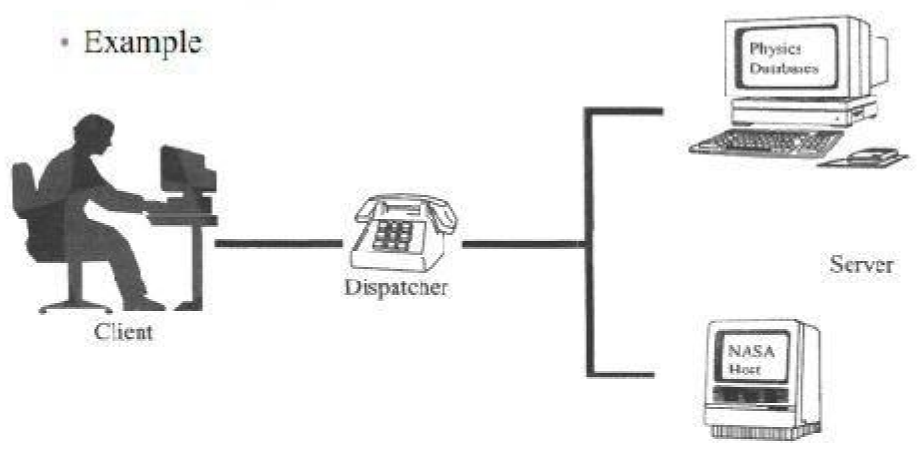


Client-Dispatcher-Server

- **Goals**
  - **Introduce an intermediate layer between clients and servers : the dispatcher**
  - **Provide location transparency**
  - **Hides details of establishment of communication**
- **Applicability**
  - **A software system integrating a set of distributed servers, with the servers running locally or distributed over a network.**

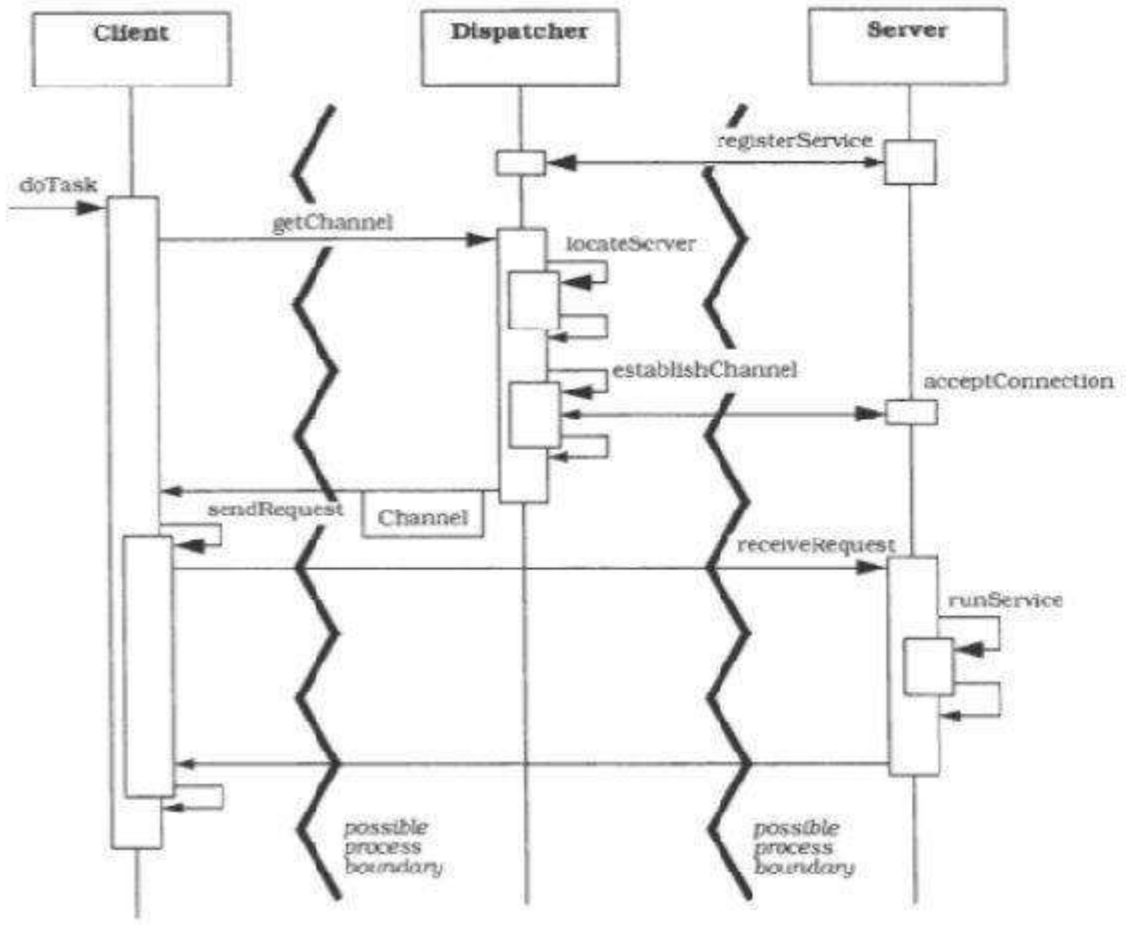
Client-Dispatcher-Server

- Example

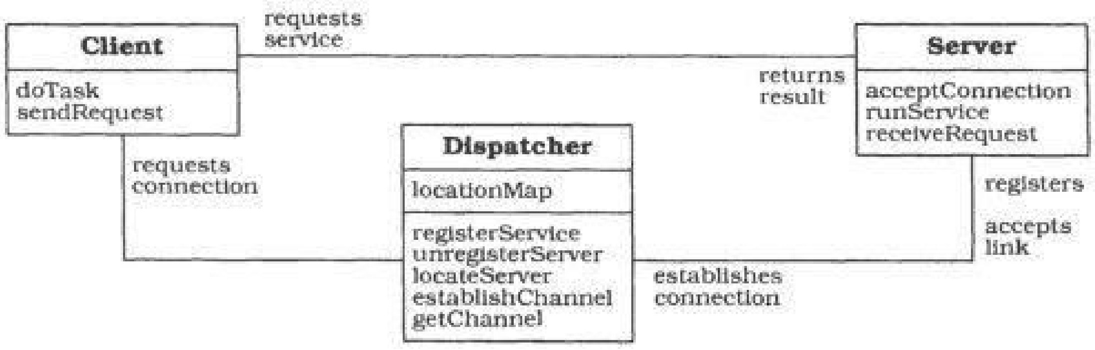


- **Components**
  - **Client**
    - **Performs some domain-specific tasks**
    - **Accesses operations offered by servers**
      - **Ask the dispatcher for a communication channel**
      - **Send its request to the server by this channel**
  - **Server**
    - **Provides services to clients**
    - **Registers itself with the dispatcher**
  - **Dispatcher**
    - **Establishes communications channels**
    - **Locates servers**
    - **(Un-)Registers servers**
    - **Maintains a map of server locations and name**

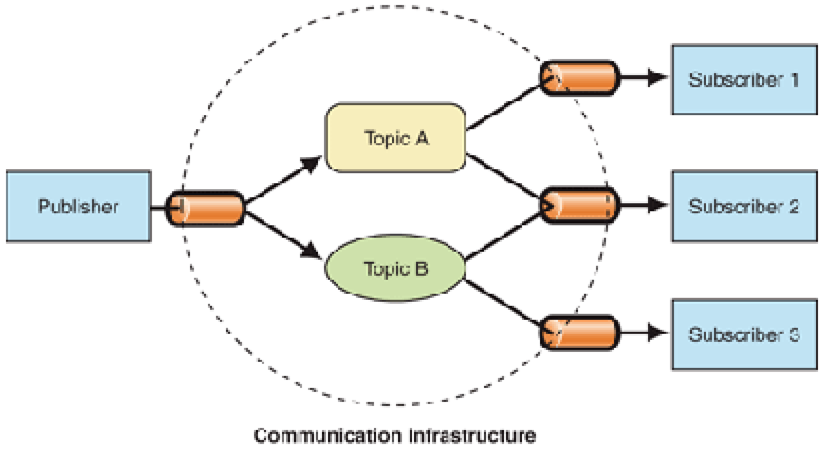
Interaction protocol

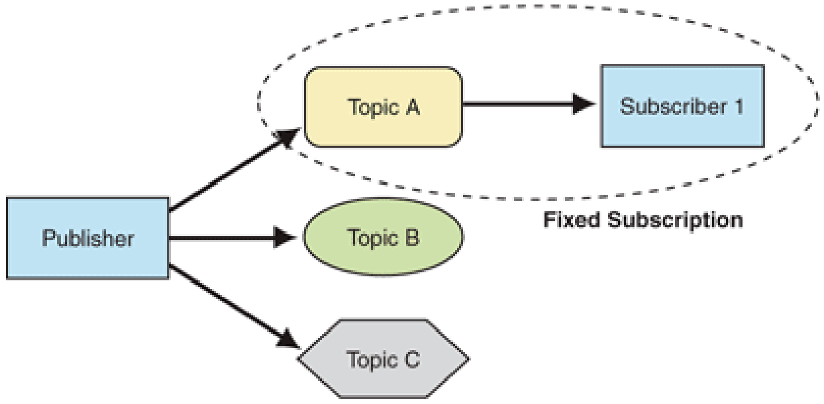


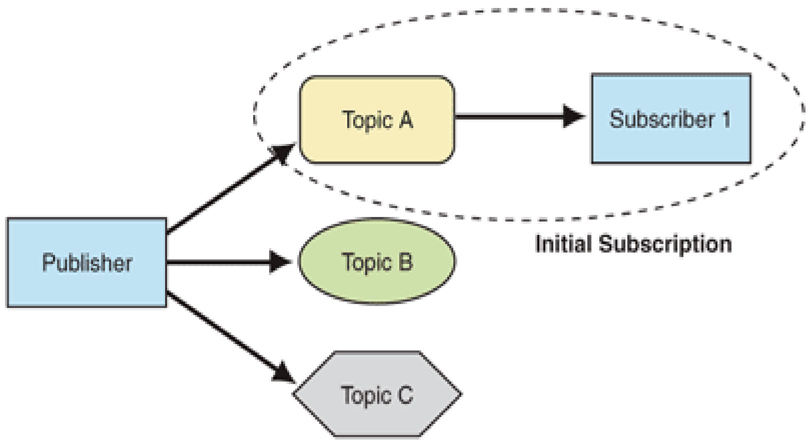
- Component structure and inter-relationships

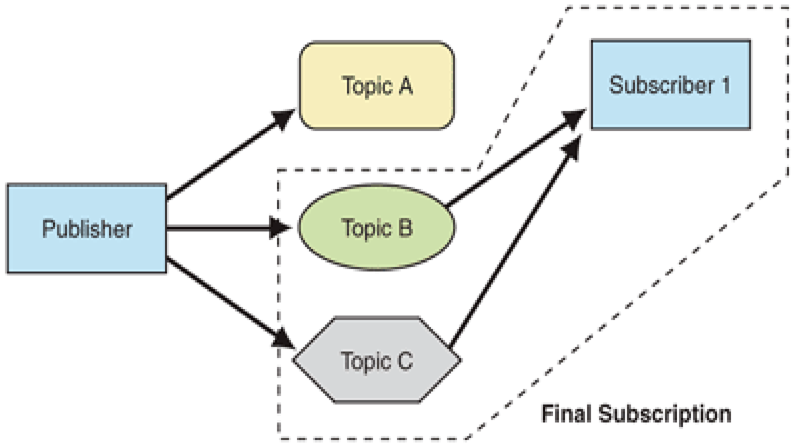


Publisher-Subscriber


Publisher
Topic A
Topic B
Subscriber 1
Subscriber 2
Subscriber 3
Communication Infrastructure


Topic A
Subscriber 1
Fixed Subscription
Publisher
Topic B
Topic C





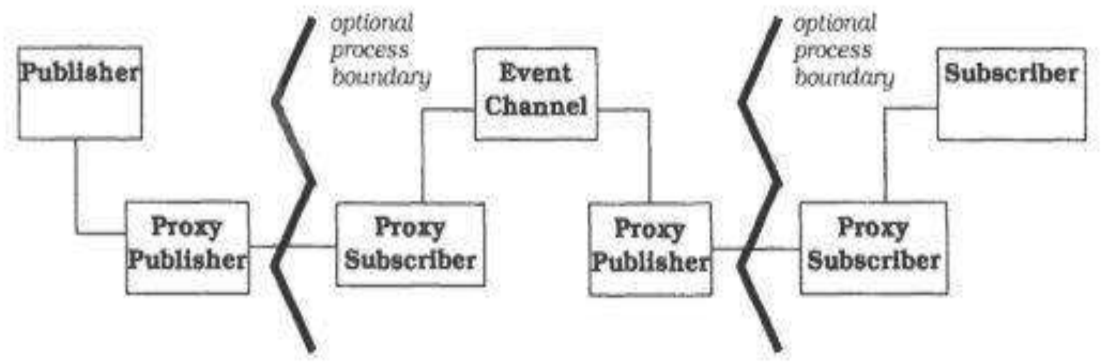
Publisher-Subscriber

- Goal
  - Help to keep the state of cooperation components synchronized
  - One publisher notifies any number of subscribers about changes to its state
- Applicability
  - Applications in which data changes in one place but many other components depend on this data
  - Number and identities of dependant components may changeover time
- Example : graphical user interfaces

Components

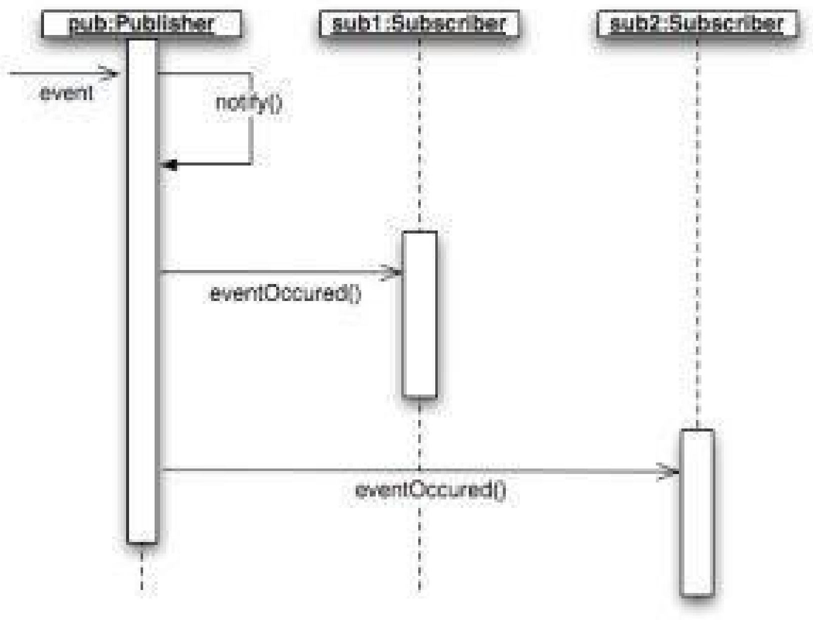
- Publisher
  - Maintains registry of currently-subscribed components
  - Sends notification to subscribers when its state has changed
- Subscriber
  - Can use the (un)subscribe interface of the publisher
  - Retrieve changed data from publisher

- **Push model**
  - Publisher sends all changed data when it notifies the subscriber
  - Rigid dynamic behavior
  - Poor choice for complex data changes
  - Useful when subscribers need published information most of the time
- **Pull model**
  - Publisher only sends minimal information when sending a change notification
  - Subscribers are responsible for retrieving the data they need
  - Offers more flexibility but higher number of messages between publisher and subscriber
  - Useful when only individual subscribers can decide if and when they need a specific piece of information
- **Strengths**
  - Loosely-coupled
  - Publishers are loosely coupled to subscribers
  - Scalable in small installations
- **Weaknesses**
  - Not so scalable in large installations
  - Publisher assumes that subscriber is listening
- **Variants**
  - **Gatekeeper**
    - Publisher notifies remote subscribers
  - **Event Channel**
    - Strongly decouples publishers and subscribers
    - Possible to have more than one publisher
    - Subscribers only wish to be notified about changes, don’t care in which component changes occurred
    - Publishers are not interested in which components are subscribing
    - Event channel created and placed between publishers and subscribers
    - Appears as a subscriber to publishers
    - Appears as a publisher to subscribers
    - Event channel, subscriber and publisher can be in different processes
    - Can use buffers, can be chained (Unix pipes)



- **Variants**
  - Use of Producer-Consumer style of cooperation
    - Producer supplies information, consumer accepts it
    - Strongly decoupled thanks to a buffer
    - Only synchronization is for buffer under/overflow
    - Event-Channel pattern can simulate a P-C with more than one producer or consumer
- **Known uses**
  - **Java Swing, GUIs**

- Interaction protocol



Unit 8: DESIGN PATTERNS-2

SYLLABUS: --------- 6 hr

- **Management Patterns**
  - **Command processor**
  - **View handler**

Idioms

- **Introduction**
- **What can idioms provide?**
- **Idioms and style**
- **Where to find idioms**

Counted pointer example

Design Patterns Management

Systems must often handle collections of objects of similar kinds, of service, or even complex components.

E.g.1 Incoming events from users or other systems, which must be interpreted and scheduled approximately.

e.g.2 When interactive systems must present application-specific data in a variety of different way, such views must be handled approximately, both individually and collectively.

- In well-structured s/w systems, separate manager components are used to handle such homogeneous collections of objects.

For this two design patterns are described

- The Command processor pattern
- The View Handler pattern

Command Processor

- The command processor design pattern separates the request for a service from its execution. A command processor component manages requests as separate objects, schedules their execution, and provides additional services such as the storing of request objects for later undo.

Context:

Applications that need flexible and extensible user interfaces or Applications that provides services related to the execution of user functions, such as scheduling or undo.

Problem:

- Application needs a large set of features.
- Need a solution that is well-structured for mapping its interface to its internal functionality
- Need to implement pop-up menus, keyboard shortcuts, or external control of application via a scripting language
- We need to balance the following forces:
  - Different users like to work with an application in different ways.
  - Enhancement of the application should not break existing code.

- Additional services such as undo should be implemented consistently for all requests.

Solution:

- Use the command processor pattern
- Encapsulate requests into objects
- Whenever user calls a specific function of the application, the request is turned into a command object.
- The central component of our pattern description, the command processor component takes care of all command objects.
- It schedules the execution of commands, may store them for later undo and may provide other additional services such as logging the sequences of commands for testing purposes.

Example : Multiple undo operations in Photosho

Structure:

- Command processor pattern consists of following components:
  - The abstract command component
  - A command component
  - The controller component
  - The command processor component
  - The supplier component

Components

- **Abstract command Component:**
- Defines a uniform interface of all commands objects
- At least has a procedure to execute a command
- May have other procedures for additional services as undo, logging,…

Class Abstract Command	***Collaborators***
Responsibility • Defines a uniform interface Interface to execute commands • Extends the interface for services of the command processor such as undo and logging	

- **A Command component:**
- For each user function we derive a command component from the abstract command.
- Implements interface of abstract command by using zero or more supplier components.
- Encapsulates a function request
- Uses suppliers to perform requests
- E.g. undo in text editor : save text + cursor position

Class	*Collaborators*
Command	▪ Supplier
Responsibility • Encapsulates a function request • Implements interface of abstract command • Uses suppliers to perform requests	

- **The Controller Component:**
- Represents the interface to the application
- Accepts service requests (e.g. bold text, paste text) and creates the corresponding command objects
- The command objects are then delivered to the command processor for execution

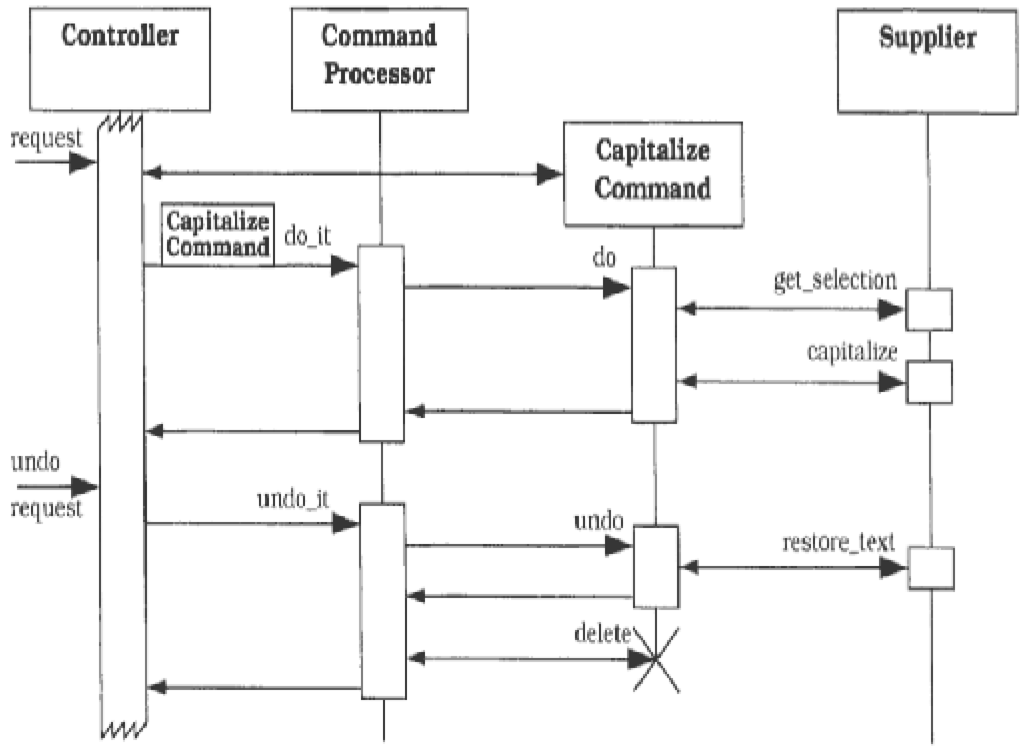
Class	*Collaborators*
Controller	▪ Command Processor ▪ Command
Responsibility • Accepts service requests • Translates requests into Commands • Transfer commands to command processor	

- **Command processor Component:**
- Manages command objects, schedule them and start their execution
- Key component that implements additional services (e.g. stores commands for later undo)
- Remains independent of specific commands (uses abstract command interface)

Class	*Collaborators*
Command Processor	▪ Abstract Command
Responsibility • Activates command execution • Maintains command objects • Provides additional services related to command execution	

- **The Supplier Component:**
- Provides functionality required to execute concrete commands
- Related commands often share suppliers
- E.g. undo : supplier has to provide a means to save and restore its internal state

Class Supplier	***Collaborators***
Responsibility • Provides application specific functionality	



The following steps occur:

The controller accepts the request from the user within its event loop and creates a capitalize' command object.

The controller transfers the new command object to the command processor for execution and further handling.

The command processor activates the execution of the command and stores it for later undo.

The capitalize command retrieves the currently-selected text from its supplier, stores the text and its position in the document, and asks the supplier to actually capitalize the selection.

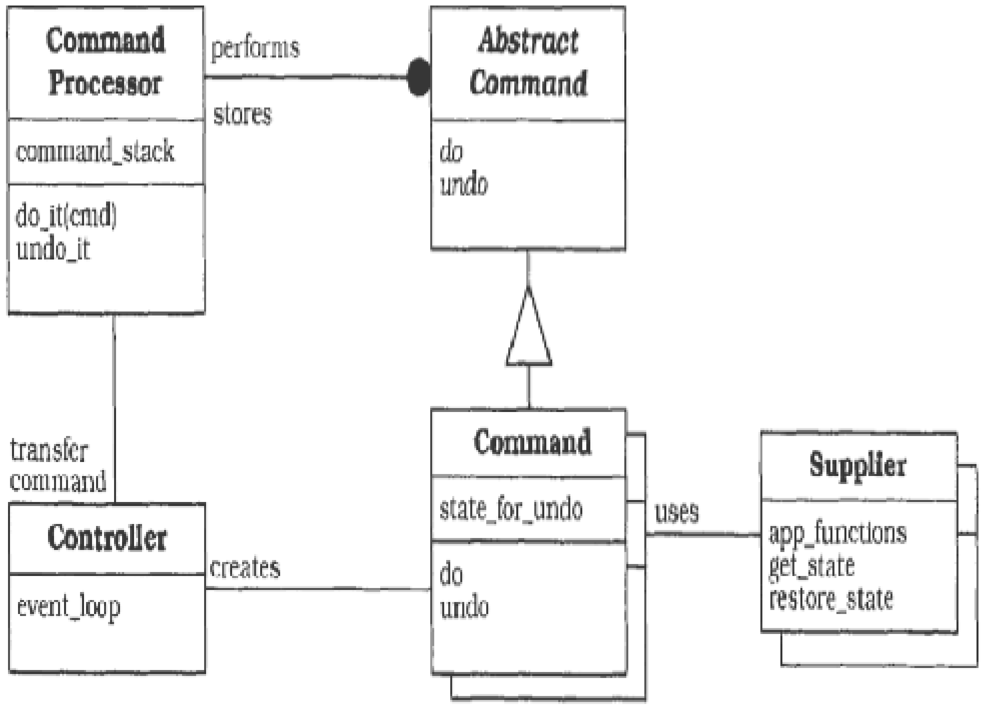
After accepting an undo request, the controller transfers this request to the command processor.

The command processor invokes the undo procedure of the most recent command.

The capitalize command resets the supplier to the previous state, by replacing the saved text in its original position

If no further activity is required or possible of the command, the command processor deletes the command object.

Component structure and inter-relationships



Strengths

▫ Flexibility in the way requests are activated

- Different requests can generate the same kind of command object (e.g. use GUI or keyboard shortcuts)

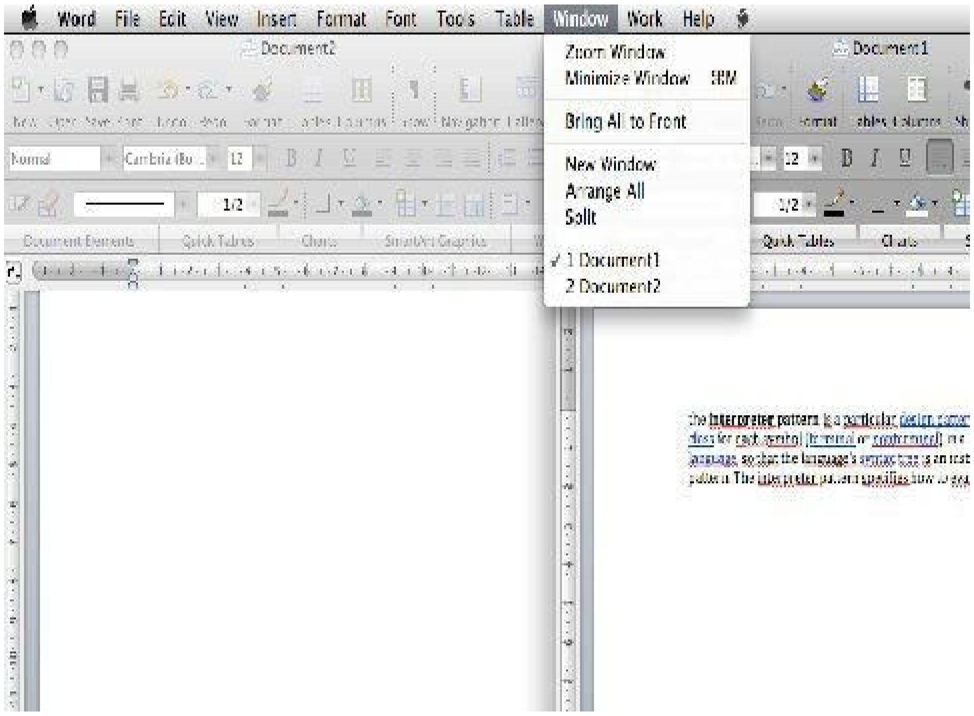
▫ Flexibility in the number and functionality of requests

- Controller and command processor implemented independently of functionality of individual commands

▪ Easy to change implementation of commands or to introduce new ones

▫ Programming execution-related services
• Command processor can easily add services like logging, scheduling,…
▫ Testability at application level
• Regression tests written in scripting language
▫ Concurrency
• Commands can be executed in separate threads
• Responsiveness improved but need for synchronization

Weaknesses
▫ Efficiency loss
▫ Potential for an excessive number of command classes
• Application with rich functionality may lead to many command classes
• Can be handled by grouping, unifying simple commands
▫ Complexity in acquiring command parameters
Variants
▫ Spread controller functionality
• Role of controller distributed over several components (e.g. each menu button creates a command object)
▫ Combination with Interpreter pattern
• Scripting language provides programmable interface
• Parser component of script interpreter takes role of controller
View Handler
• **Goals**
• Help to manage all views that a software system provides
• Allow clients to open, manipulate and dispose of views
• Coordinate dependencies between views and organizes their update
• **Applicability**
• Software system that provides multiple views of application specific data, or that supports working with multiple documents
• Example : Windows handler in Microsoft Word



View Handler and other patterns

• MVC

• View Handler pattern is a refinement of the relationship between the model and its associated views.

- PAC

• Implements the coordination of multiple views according to the principles of the View Handler pattern.

❖ **Components**

▫ View Handler

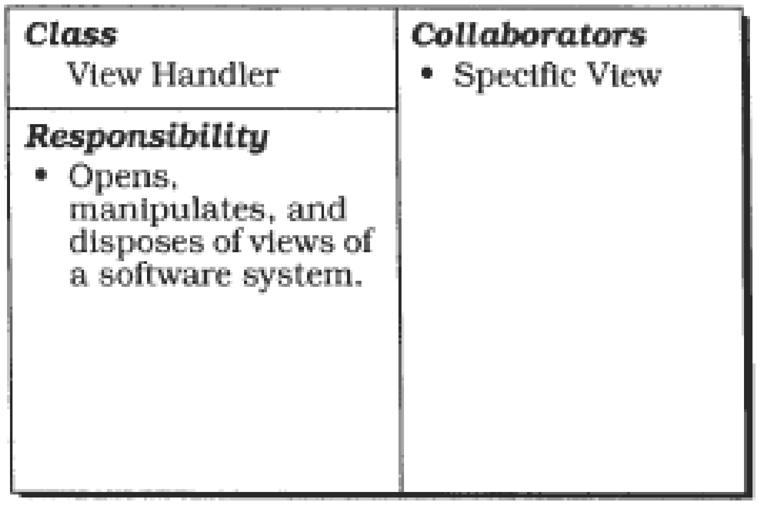
- Is responsible for opening new views, view initialization

- Offers functions for closing views, both individual ones and all currently-open views

- View Handlers patterns adapt the idea of separating presentation from functional core.

- The main responsibility is to Offers view management services (e.g. bring to foreground, tile all view, clone views)

- Coordinates views according to dependencies



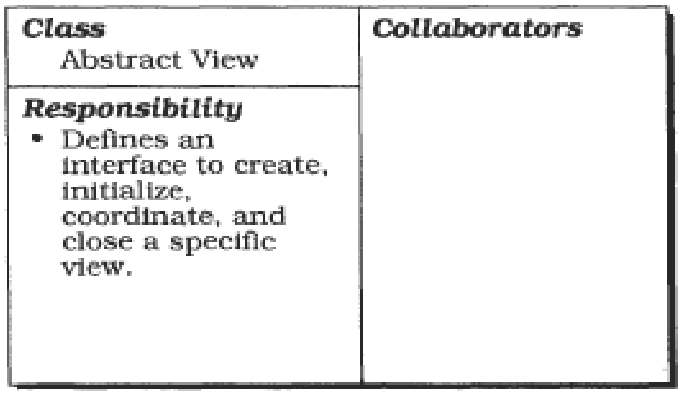
Class View Handler	***Collaborators*** • Specific View
Responsibility • Opens, manipulates, and disposes of views of a software system.	

❖ **Components**

▫ Abstract view

- Defines common interface for all views

- Used by the view handler : create, initialize, coordinate, close, etc.



Class Abstract View	***Collaborators***
Responsibility • Defines an interface to create, initialize, coordinate, and close a specific view.	

❖ **Components**

- Specific view

• Implements Abstract view interface

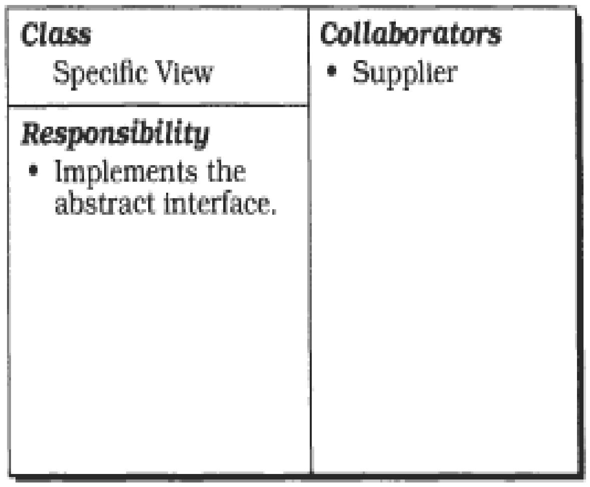
• Knows how to display itself

- Retrieves data from supplier(s) and change data

• Prepares data for display

• Presents them to the user

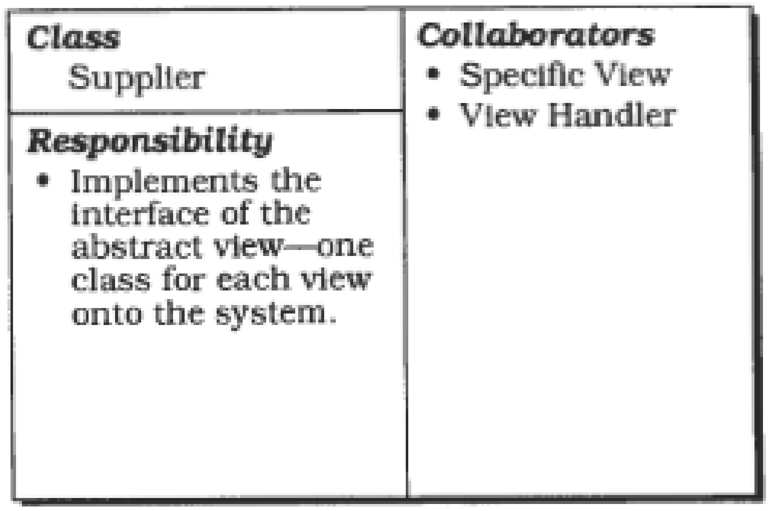
• Display function called when opening or updating a view



Class Specific View	Collaborators
Responsibility • Implements the abstract interface.	• Supplier

❖ **Components**

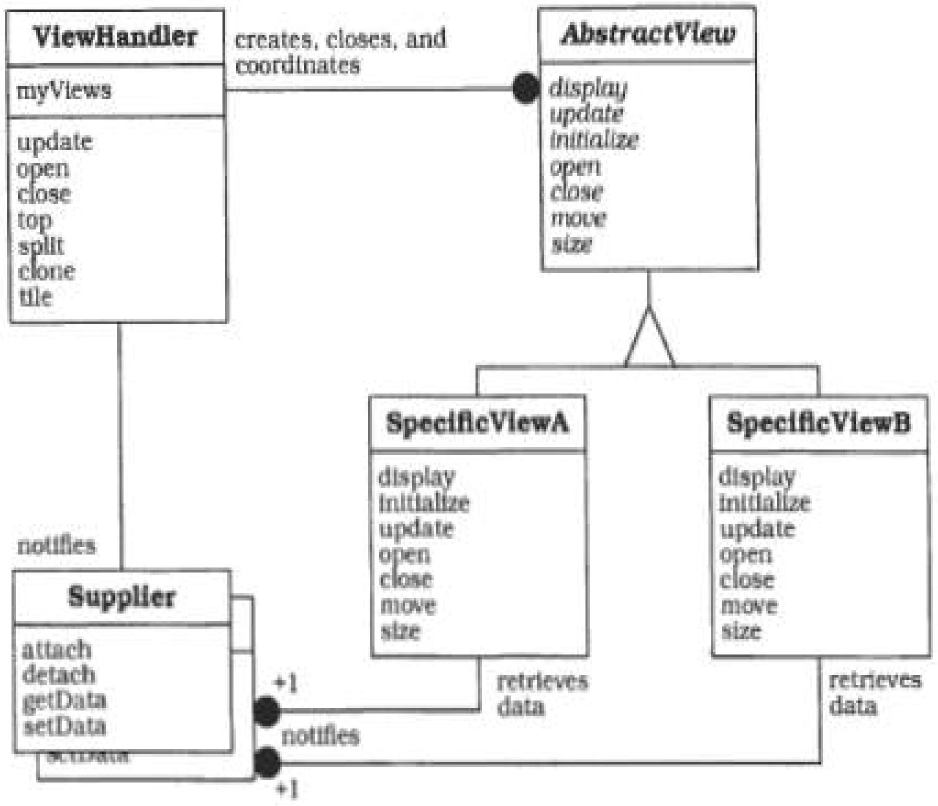
- **Supplier**
  - Provides the data that is displayed by the view components
  - Offers interface to retrieve or change data
  - Notifies dependent component about changes in data



Class Supplier	Collaborators
Responsibility • Implements the interface of the abstract view—one class for each view onto the system.	• Specific View • View Handler

The OMT diagram that shows the structure of view handler pattern

Component structure and inter-relationships



Two scenarios to illustrate the behavior of the View Handler

- View creation
- View tiling

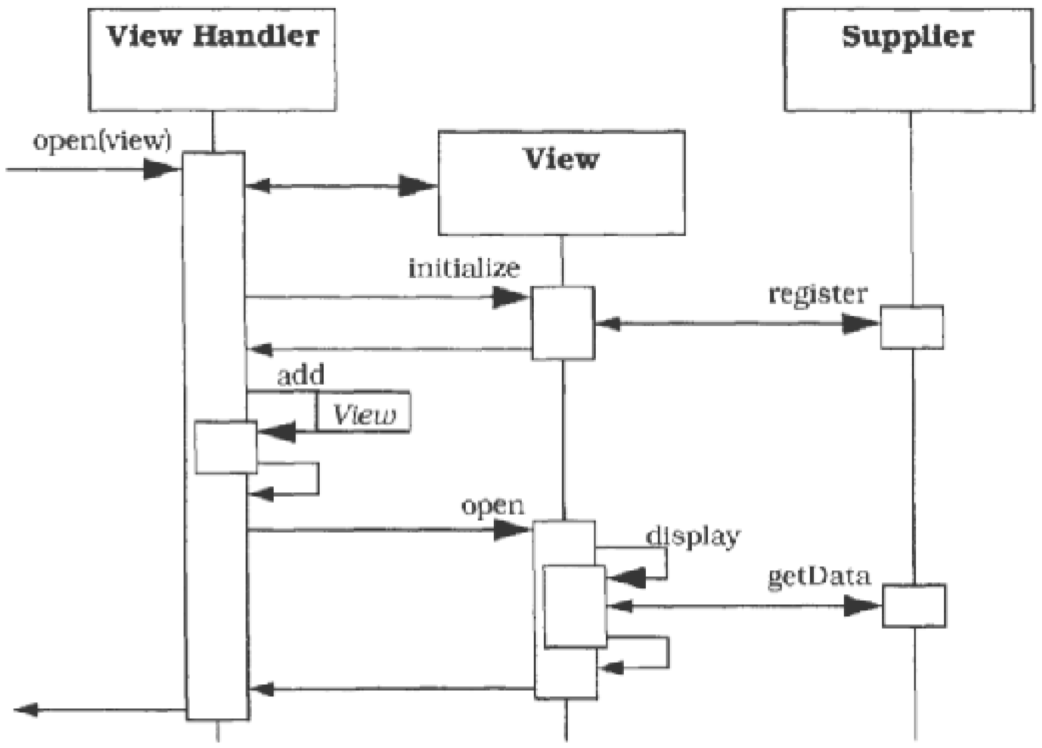
Both scenarios assume that each view is displayed in its own window.

Scenario I : View creation

Shows how the view handler creates a new view. The scenario comprises four phases:

- A client-which may be the user or another component of the 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, as specified by the Publisher-Subscriber pattern.
- 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 this data for display, and presents it to the user.

Interaction protocol

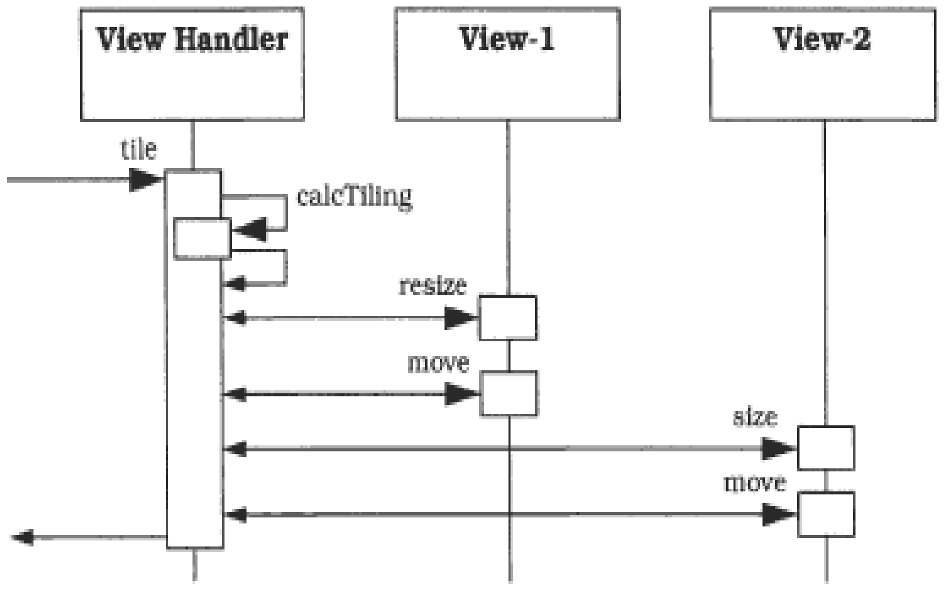


Scenario II : View Tiling

Illustrates how the view handler organizes the tiling of views. For simplicity, we assume that only two views are open. The scenario is divided into three phases:

- The user invokes the command to tile all open windows. The request is sent to the view handler.
- 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 it displays to the user. We assume that views cache the image they display. If this is not the case, views must retrieve data from their associated suppliers

Interaction protocol



Implementation

The implementation of a View Handler structure can be divided into four steps. We assume that the suppliers already exist, and include a suitable change-propagation mechanism.

1. Identify the ***views.***
2. Specify *a common interface for all* **views.**
3. ***Implement the views.***
4. *Define the* ***view handler***

Identify the ***views.*** Specify the types of views to be provided and how the user controls each individual view.

Specify *a common interface for all* **views.** This should include functions to open, close, display, update, and manipulate a view. The interface may also offer a function to initialize a view.

The public interface includes methods to open, close, move, size, drag, and update a view, as well as an initialization method.

Implementation

```
class AbstractView {
protected:
     // Draw the view
     virtual void displayData() = 0;
     virtual void displayWindow(Rectangle boundary) = 0;
     virtual void eraseWindow() = 0;

public:
     // Constructor and Destructor
     AbstractView() {};
     ~AbstractView() {};
     // Initialize the view
     void initialize() = 0;
     // View handling with default implementation
     virtual void open(Rectangle boundary) { /* ... */ };
     virtual void close() { /* ... */ };
     virtual void move(Point point) { /* ... */ };
     virtual void size(Rectangle boundary) { /* ... */ };
     virtual void drag(Rectangle boundary) { /* ... */ };
     virtual void update() { /* ... */ };
};                                                  [
```

Implement the views. *Derive a separate class from the* ***AbtrsactView*** class for each specific type of view identified in step 1. Implement the view-specific parts of the interface, such as the **displayData *()*** method in our example. Override those methods whose default implementation does not meet the requirements of the specific view.

In our example we implement three view classes: *Editview*, *Layoutview*, and *Thumbnailview***,** as specified in the solution section.

Define the ***view handler:*** *Implement functions for creating views as* Factory Methods.

The view handler in our example document editor provides functions to open and close views, as well as to tile them, bring them to the foreground, and clone them. Internally the view handler maintains references to all open views, including information about their position and size, and whether they are iconize.

```
class ViewHandler {
     // Data structures
     struct ViewInfo {
          AbstractView* view;
          Rectangle     boundary;
          bool          iconized;
     };
```

```
    Container<ViewInfo*> myViews;
    // The singleton instance
    static ViewHandler* theViewHandler;
    // Constructor and Destructor
    ViewHandler();
    ~ViewHandler();
public:
    // Singleton constructor
    static ViewHandler* makeViewHandler();

    // Open and close views
    void open(AbstractView* view);
    void close(AbstractView* view);

    // Top, clone, and tile views
    void top(AbstractView* view);
    void clone(); // Clones the top-most view
    void tile();
};
void ViewHandler::openView(AbstractView* view){
   ViewInfo*     viewInfo = new ViewInfo();

   // Add the view to the list of open views
   viewInfo->view        = view;
   viewInfo->boundary    = defaultBoundary;
   viewInfo->iconized    = false;
   myViews.add(viewInfo);

   // Initialize the view and open it
   view->initialize();
   view->open(defaultBoundary);
};
```

Strengths

- ❑ Uniform handling of views
  - ▪ All views share a common interface
- ▫ Extensibility and changeability of views
  - ▪ New views or changes in the implementation of one view don't affect other component
- ▫ Application-specific view coordination

▪ Views are managed by a central instance, so it is easy to implement specific view coordination strategies (e.g. order in updating views)

Weaknesses

▫Efficiency loss (indirection)

▪ Negligible

▫ Restricted applicability : useful only with

▪ Many different views

▪ Views with logical dependencies

▪ Need of specific view coordination strategies

Variant

▫ View Handler with Command objects

▪ Uses command objects to keep the view handler independent of specific view interface

▪ Instead of calling view functionality directly, the view handler creates an appropriate command and executes it

Known uses

▫ Macintosh Window Manager

▪ Window allocation, display, movement and sizing

▪ Low-level view handler : handles individual window

▫ Microsoft Word

▪ Window cloning, splitting, tiling…

Idioms

Introduction

➢ idioms are low-level patterns specific to a programming language

➢ An idiom describes how to implement particular aspects of components or the relationships between them with the features of the given language.

➢ Here idioms show how they can define a programming style, and show where you can find idioms.

➢ A programming style is characterized by the way language constructs are used to implement a solution, such as the kind of loop statements used, the naming of program elements, and even the formatting of the source code

What Can Idioms Provide?

- A single idiom might help you to solve a recurring problem with the programming language you normally use.

- They provide a vehicle for communication among software developers.(because each idiom has a unique name)

- idioms are less 'portable‘ between programming languages

Idioms and Style

If programmers who use different styles form a team, they should agree on a single coding style for their programs. For example, consider the following sections of C/C++ code, which both implement a string copy function for 'C-style' string

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

void strcopyPascal (char d [ I , const char s [I )
 { int i ;
   for (i = 0: s[il != ' \ O 1 : i = i + 1)
  { d[i]  = s [i];  }
  d[i] = ‘\0’ ; /* always asign 0 character */
}/* END of strcopyPasca1 */
```

Idioms and Style

A program that uses a mixture of both styles might be much harder to understand and maintain than a program that uses one style consistently.

Corporate style guides are one approach to achieving a consistent style throughout programs developed by teams.

Style guides that contain collected idioms work better. They not only give the rules, but also provide insight into the problems solved by a rule. They name the idioms and thus allow them to be communicated.

Idioms from conflicting styles do not mix well if applied carelessly to a program. Different sets of idioms may be appropriate for different domains. example, you can write C++ programs in an object-oriented style with inheritance and dynamic binding.

In real time system dynamic binding is not used which is required.

A single style guide can therefore be unsuitable for large companies that employ many teams to develop applications in different domains.

A coherent set of idioms leads to a consistent style in your programs.

Here is an example of a style guide idiom from Kent Beck's *Smalltalk Best Practice Patterns :*

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: [ . . . I
```

- Different sets of idioms may be appropriate for different domains.

- For example, you can write C++ programs in an object-oriented style with inheritance and dynamic binding.

- In some domains. such as real-time systems, a more 'efficient' style that does not use dynamic binding is required.

- A single style guide can therefore be unsuitable for large companies that employ many teams to develop applications in different domains.

- A style guide cannot and should not cover a variety of styles.

Where Can You Find Idioms?

- Idioms that form several different coding styles in C++ can be found for example in Coplien's *Advanced C++ Barton and Neck man's Scientific and Engineering C++ and Meyers' Effective C++* .

- You can find a good collection of Smalltalk programming wisdom in the idioms presented in Kent Beck's columns in the *Smalltalk Report.*

- His collection of *Smalltalk Best Practice Patterns is about to be* published as a book .

- Beck defines a programming style with his coding patterns that is consistent with the Smalltalk class library, so you can treat this pattern collection as a Smalltalk style guide.