SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-VI

A-7E Avionics System: A Case Study in Utilizing Architectural Structures

An object-oriented program's runtime structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program's runtime structure consists of rapidly changing networks of communicating objects. In fact, the two structures are largely independent. Trying to [understand] one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa.

software architecture describes elements of a system and the relations among them. We architecture of a system. Each structure concentrates on one aspect of the architecture. case study of an architecture designed by engineering and specifying three specific architectural structures: module decomposition, uses, and process. We will see how these structures complement each other to provide a complete picture of how the system works, and we will see how certain qualities of the system are affected by each one.

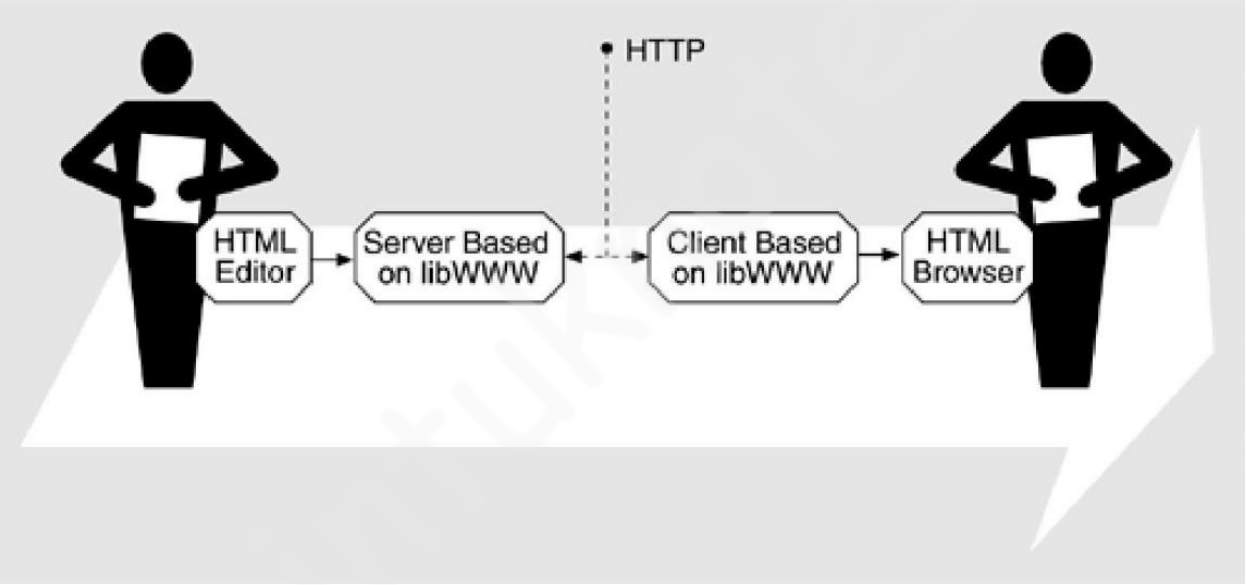
Table summarizes the three structures we will discuss.

Table The A-7E's Architecural Structures

Structure	Elements	Relation among Elements	Has Influence Over
Module Decomposition	Modules (implementation units)	Is a submodule of; shares a secret with	Ease of change
Uses	Procedures	Requires the correct presence of	Ability to field subsets and develop incrementally
Process	Processes; thread of procedures	Synchronizes with; shares CPU with; excludes	Schedulability; achieving performance goals through parallelism

The World Wide Web'A Case Study in Interoperability Architectural Solution

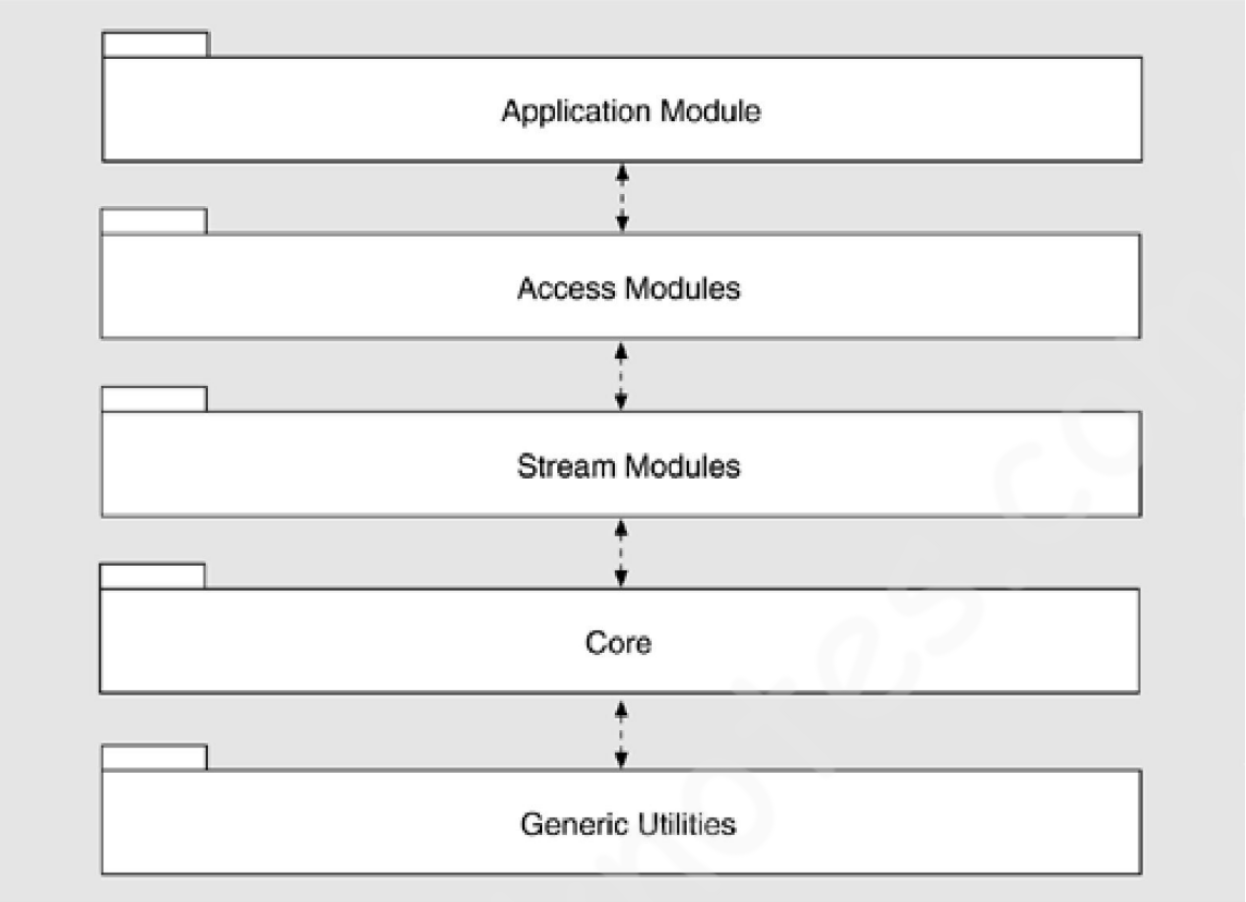
The basic architectural approach used for the Web, first at CERN and later at the World Wide Web Consortium (W3C), relied on clients and servers and a library (libWWW) that masks all hardware, operating system, and protocol dependencies. Figure 13.3 shows how the content producers and consumers interact through their respective servers and clients. The producer places content that is described in HTML on a server machine. The server communicates with a client using the HyperText Transfer Protocol (HTTP). The software on both the server and the client is based on libWWW, so the details of the protocol and the dependencies on the platforms are masked from it. One of the elements on the client side is a browser that knows how to display HTML so that the content consumer is presented with an understandable image. Content producers and consumers interact through clients and servers consumers interact through clients and servers



Consumers interact through clients and servers

MEETING THE ORIGINAL REQUIREMENTS: libWWW

As stated earlier, libWWW is a library of software for creating applications that run on either the client or the server. It provides the generic functionality that is shared by most applications: the ability to connect with remote hosts, the ability to understand streams of HTML data, and so forth. libWWW is a compact, portable library that can be built on to create Web-based applications such as clients, servers, databases, and Web spiders. It is organized into five layers, as Figure A layered view of libWWW



ACHIEVING INITIAL QUALITY GOALS

describes how the Web achieved its initial quality goals of remote access, interoper
extensibility, and scalability.

How the WWW Achieved Its Initial Quality Goals

Goal	HowAchieved	TacticsUsed

How the WWW Achieved Its Initial Quality Goals

Goal	HowAchieved	TacticsUsed
RemoteAccess	Build Web on topofInternet	Adherenceto definedprotocols
Interoperability	Use libWWW to mask platform details	Abstractcommon services Hide information
Extensibility of Software	Isolate protocol and data type extensions in libWWW; allow for plug-in components (applets and servlets)	Abstract common services Hide information Replace components Configuration files
Extensibility of Data	Make each data item independent except for references it controls	Limit possible options
Scalability	Use client-server architectureand keep references to other data local to referring data location	Introduce concurrency Reduce computational overhead

Air Traffic Control: A Case Study in Designing for High Availability

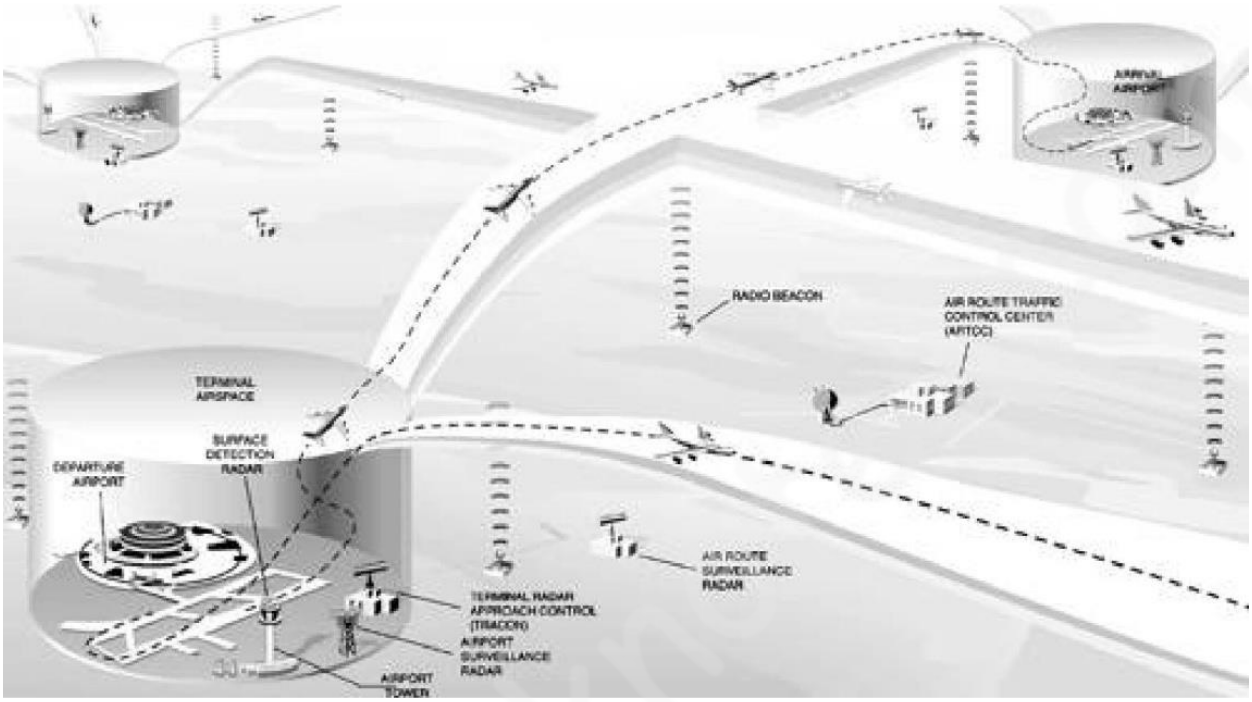
Air traffic control (ATC) is among the most demanding of all software applications. It is *hard real time*, meaning that timing deadlines must be met absolutely; it is *safety critical*, meaning that human lives may be lost if the system does not perform correctly; and it is*highly distributed*, requiring dozens of controllers to work cooperatively to guide aircraft safe, reliable airways system requires enormous expenditures of public money. ATC is a multibillion-dollar undertaking.

This chapter is a case study of one part of a once-planned, next-generation ATC system for the United States. We will see how its architecture? in particular, a set of carefully chosen views coupled with the right tactics held the key to achieving its demanding and wide-ranging requirements. Although this system was never that the system could meet its quality goals.

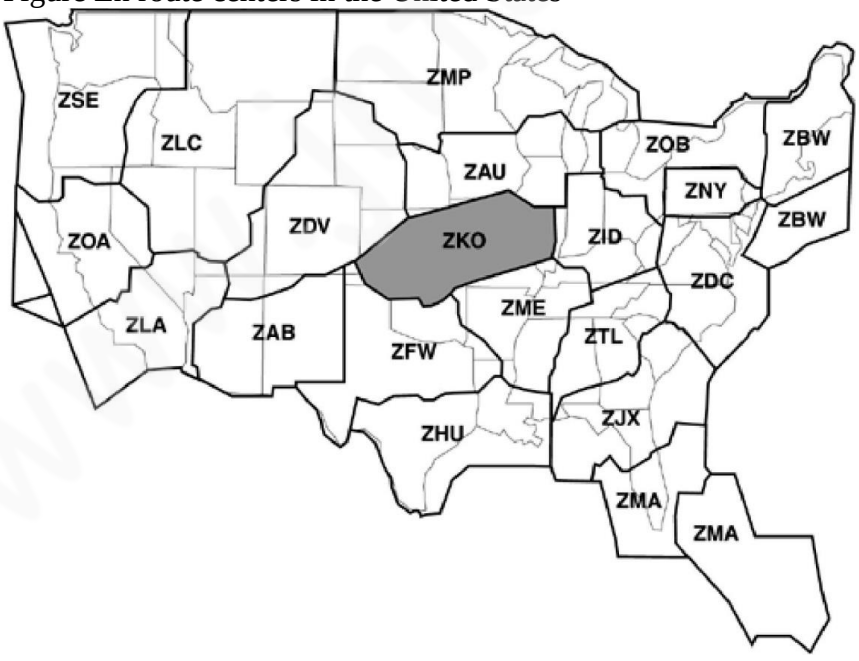
In the United States, air traffic is controlled by the Federal Aviation Administration (FAA), a government agency responsible for aviation safety in general. The FAA is the customer for the system we will describe. As a flight progresses from its departure airport to its arrival airport, it deals with several ATC entities that guide it safely through each portion of the airways (and ground facilities) it is using. *Ground control* coordinates the movement of responsibility.

Consider an airline flight from Key West, Florida, to Washington, D.C.'s Dulles Airport. The crew through the airways system. In the United States, whose skies are filled with more commercial, private, and military aircraft than any other part of the world, ATC is an area of intense public scrutiny. Aside from the obvious safety issues, building and maintaining a of the flight will communicate with Key West ground control to taxi from the gate to the end of the runway, Key West tower during takeoff and climb-out, and then Miami Center (the en route center whose airspace covers Key West) once it leaves the Key West terminal control area. From there the flight will be handed off to Jacksonville Center, Atlanta Center, and so forth, until it enters the airspace controlled by Washington Center. From Washington Center, it will be handed off to the Dulles tower, which will guide its approach and landing.

Flying from point A to point B in the U.S. air traffic control system.



When it leaves the runway, the flight will communicate with Dulles ground control for its taxi to
Figure En route centers in the United States



the gate. This is an oversimplified view of ATC in the United States, but it suffices for our case study. Figure 6.1 shows the hand-off process, and Figure 6.2 shows the 22 enroutecenters. designs and elements where possible because the ISSS developer also intended to bid on the other systems. After all, these different systems (en route center, tower, ground control) share many elements: interfaces to radio systems, interfaces to flight plan databases, interfaces to each other, interpreting radar data, requirements for reliability and intended to be an upgraded hardware and software system for the 22 en route centers in the United States. It was part of a much larger government procurement that would have, in stages, installed similar upgraded systems in the towers and ground control facilities, as well as the transoceanic ATC facilities.

The fact that ISSS was to be **procured as only one of a set of strongly related systems had a profound effect on its architecture. In particular**, there was great incentive to adopt common performance, and so on. Thus, the ISSS design was influenced broadly by the requirements for all of the upgraded systems, not just the ISSS-specific ones. The complete set of upgraded systems was to be called the Advanced Automation System (AAS).

Ultimately, the AAS program was canceled in favor of a less ambitious, less costly, more staged upgrade plan. Nevertheless, ISSS is still an illuminating case study because, when the program was canceled, the design and most of the code were actually already completed. Furthermore, the architecture of the system (as well as most other aspects) was studied by an independent audit team and found to be well suited to its requirements. Finally, the system that was deployed instead of ISSS borrowed heavily from the ISSS architecture. For these reasons, we will present the ISSS architecture as an actual solution to an extremely difficult problem.

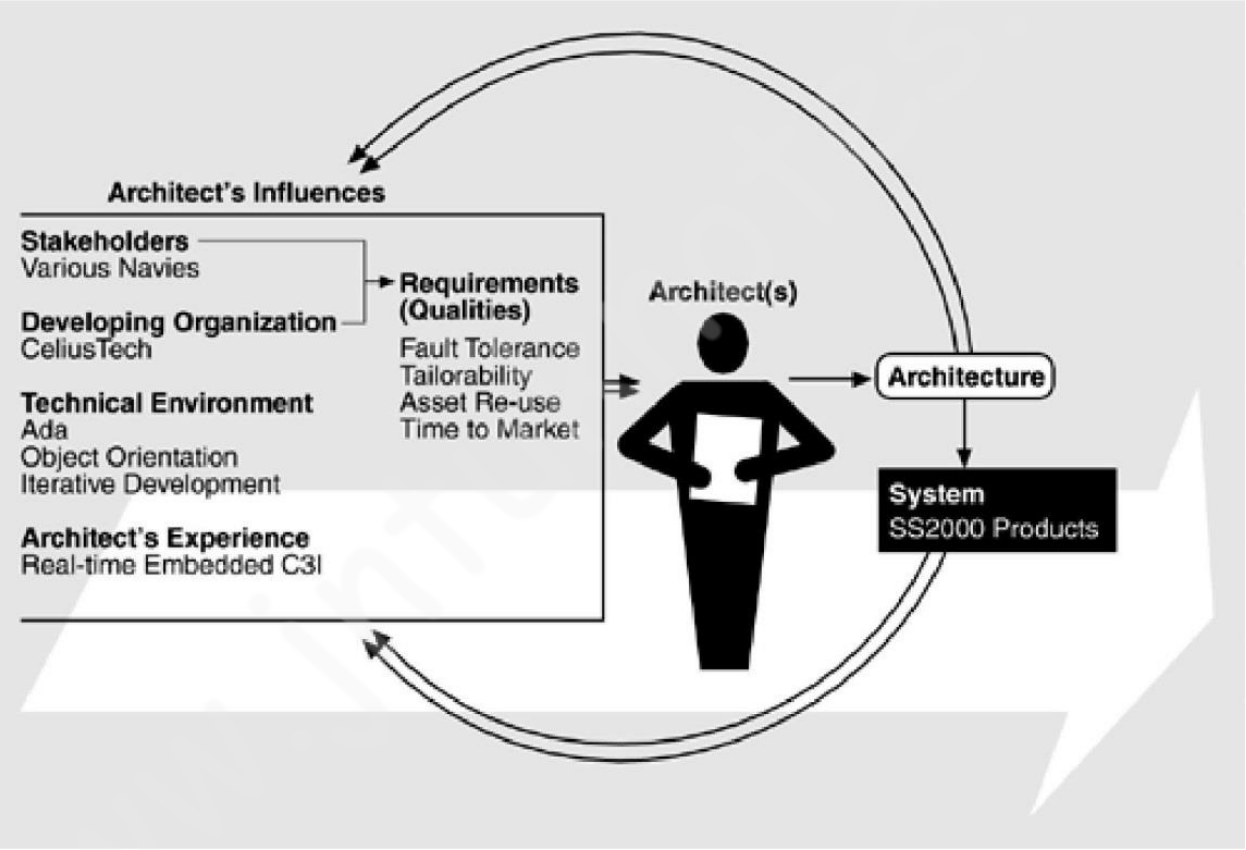
How the ATC System Achieves Its Quality Goals

Goal	How Achieved	Tactic(s) Used
High Availability	Hardware redundancy (both processor and network); software redundancy (layered fault detection and recovery)	State resynchronization; shadowing; active redundancy; removal from service; limit exposure; ping/echo; heartbeat; exception; spare
High Performance	Distributed multiprocessors; front-end schedulability analysis, and network modeling	Introduce concurrency
Openness	Interface wrappingand layering	Abstract common services; maintain interface stability
Modifiability	Templates and table-driven adaptation data; careful assignment of module responsibilities; strict use of specified interfaces	Abstract common services; semantic coherence; maintain interface stability; anticipate expected changes; generalize the module; component replacement; adherence to defined procotols; configuration files
Ability to Field Subsets	Appropriate separation of concerns	Abstract common services

CelsiusTech: A Case Study in Product Line Development This chapter relates the experience of CelsiusTech AB, a Swedish naval defense contractor that successfully adopted a product line approach to building complex software-intensive systems.

Called Ship System 2000 (SS2000), their product line consists of shipboard command-and control systems for Scandinavian, Middle Eastern, and South Pacific navies. This case study illustrates the entire Architecture Business Cycle (ABC), but especially SS2000 Requirements and How the Architecture Achieved Them Requirement HowAchieved RelatedTactic(s) shows how a product line architecture led CelsiusTech to new business opportunities. Figure shows the roles of the ABC stakeholders in the CelsiusTech experience.

Figure The ABC as applied to CelsiusTech



SS2000 Requirements and How the Architecture Achieved Them

Requirement	HowAchieved	RelatedTactic(s)
Performance	Strict network traffic protocols; software is written as a set of processes to maximize concurrency and written to be location independent, allowing for relocation to tune performance; COOB is by-passed for high-data-volume transactions; otherwise, data sent only when altered and distributed so response times are short	Introduce concurrency Reduce demand Multiple copies Increase resources
Reliability, Availability, and Safety	Redundant LAN; fault-tolerant software; standard Ada exception protocols; software written to be location independent and hence can be migrated in case of failure; strict ownership of data prevents multi-writer race conditions	Exceptions Active redundancy State resynchronization Transactions
Modifiability (including ability to produce new members of the SS2000 family)	Strict use of message-based communication provides interface isolated from implementation details; software written to be location independent; layering provides portability across platforms, network topologies, IPC protocols, etc.; data producers and consumers unaware of each other because of COOB; heavy use of Ada;	

Requirement	HowAchieved	RelatedTactic(s)
		Configuration files Component replacement Adherence to defined protocols
Testability	Interfaces using strongly typed messages push a whole class of errors to compile time; strict data ownership, semantic coherence of elements, and strong interface definitions simplify discovery of responsibility	

A Case Study (Designing a Document Editor): Design Problems, Document Structure, Formatting, Embellishing the User Interface, Supporting Multiple Look-and-Feel Standards, Supporting Multiple Window Systems, User Operations, Spelling Checking and Hyphenation.

Document Editor: A Case Study

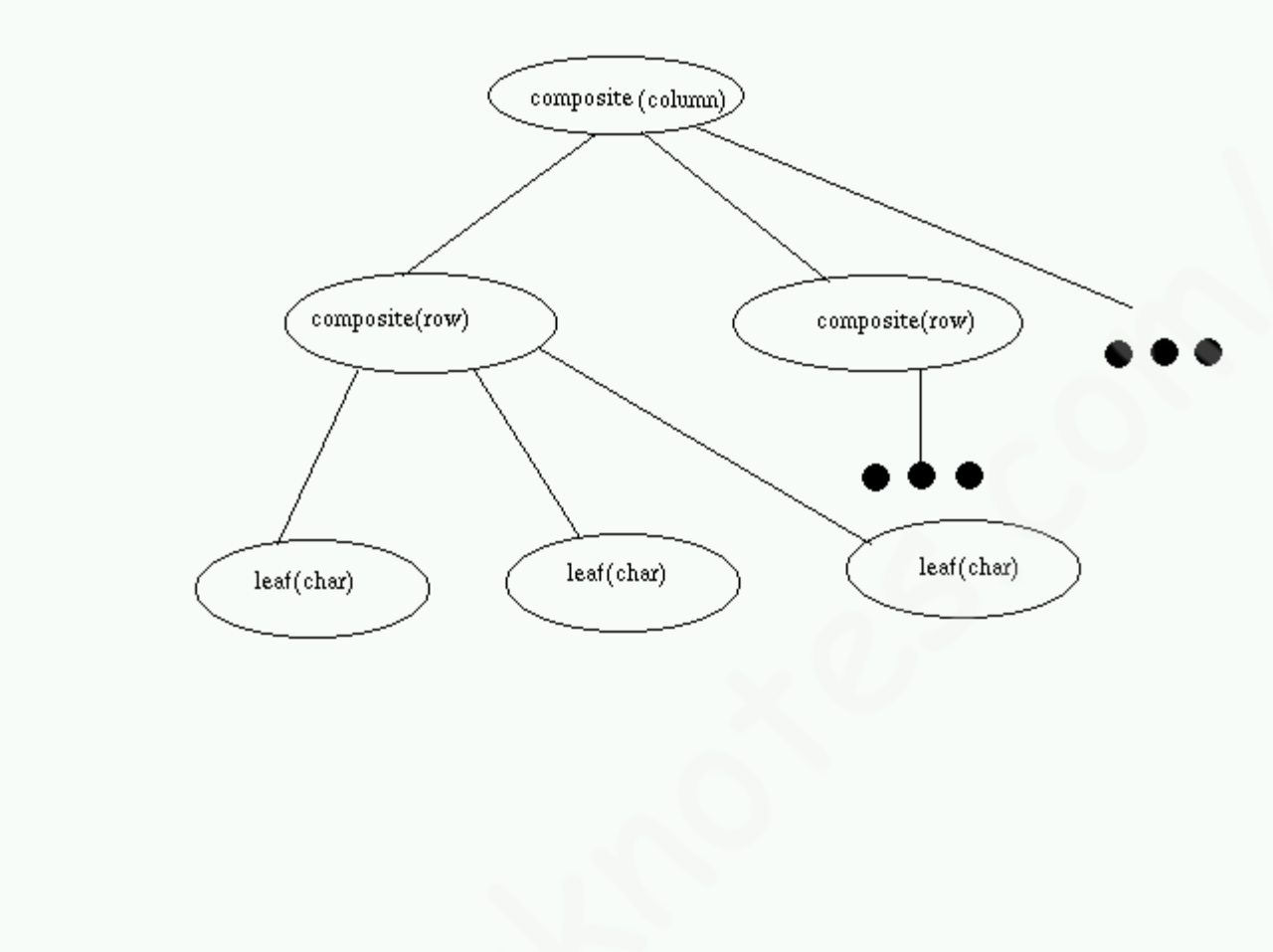
Requirements Specification:

1. Document Structure: Treat characters, lines, rows and columns uniformly.
2. Formatting: It should be possible to dynamically switch between different formatting algorithms.
3. Embellish the user interface in a very flexible and dynamic way.
4. Support multiple look and feel standards: Lexi should easily adapt to multiiple look and feel standards like Motif, PM etc.
5. Support multiple windowing systems: Lexi should be independent of the windowing systems used.
6. Support flexible user operations.
7. Spell check and hyphenation.

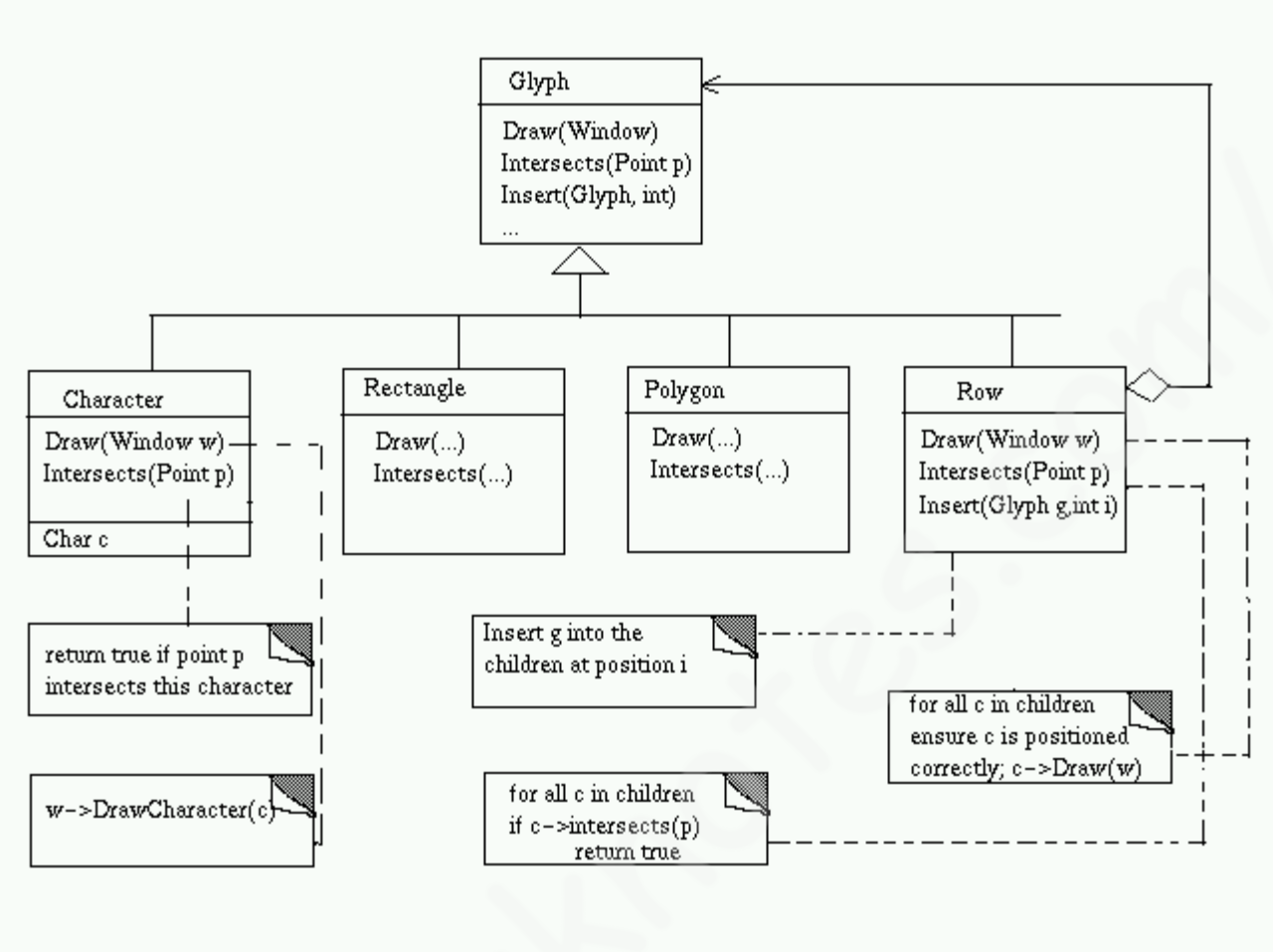
Document Structure:

- A document is an arrangement of basic graphical elements such as characters, lines, polygons and other shapes.
- The basic idea used to compose a document is that of recursive composition.
- Recursive composition is the technique by which we build complex elements out of simpler ones.
- It uses a mix of composition and inheritance.
- For example, a set of characters and graphics are arranged to form a row, multiple rows are arranged to form a column, multiple columns can form a page and so on.
- The design pattern, Composite captures the essence of recursive composition in object-oriented terms.

The resulting object structure is as shown below.

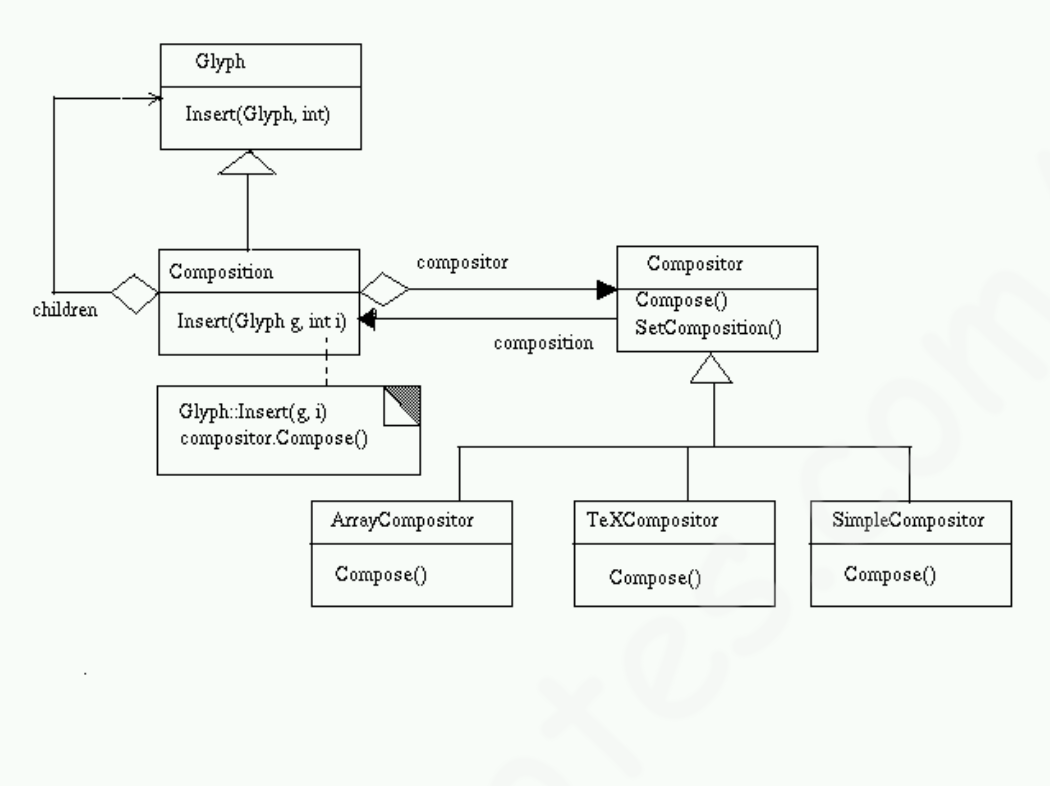


We define an abstract class, Glyph, to represent all the objects that can appear in a document. It provides three operations- Draw(Window) draws itself in the window, Intersects(Point) returns checks whether the glyph is intersecting with the point p and Insert(Glyph, int) inserts the Glyph at position 'int'. The partial Glyph class hierarchy is shown below. Here Character, Polygon and Rectangle are the leaves and Row represents the composite.



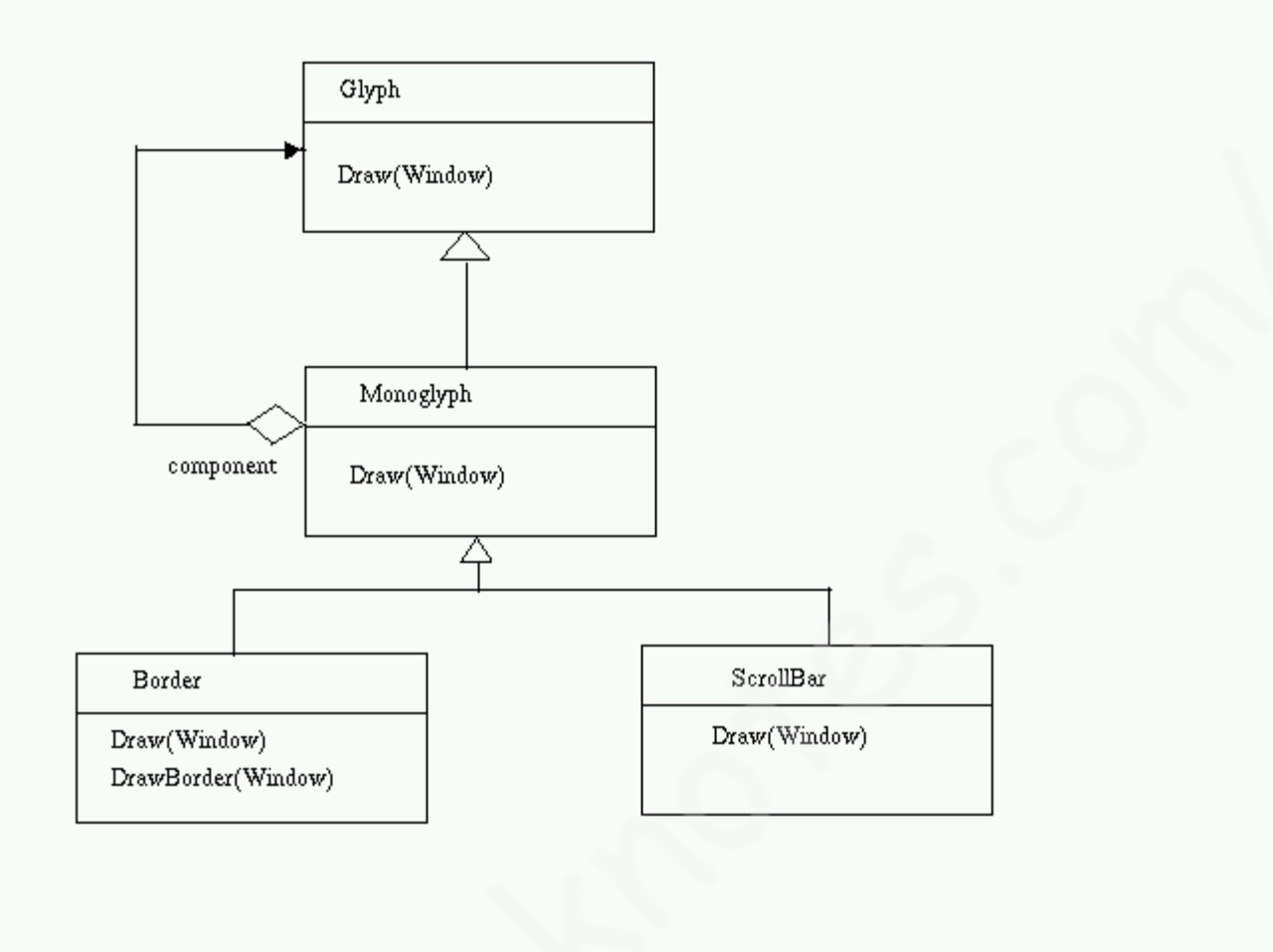
Formatting:

- Usually different formatting algorithms are used within the same document structure.
- It should be possible to switch between different algorithms.
- An important trade-off to consider in this case is the balance between formatting speed and formatting quality.
- The formatting algorithms should be completely independent of the document structure ie, they should not change with change in the document structure.
- Conversely, addition of a new algorithm shouldn't require the modification of the existing glyphs.
- Main idea used here is the encapsulation of algorithms using objects.
- Strategy is best suited for encapsulating algorithms and dynamically switching between them.
- In the class diagram show below, whenever an object of type Composition is created, a Compositor is also created, which is responsible for choosing tha appropriate algorithm.
- Here Composition is the context, Compositor is the strategy and the concrete strategies are ArrayCompositor, TeXCompositor and SimpleCompositor.

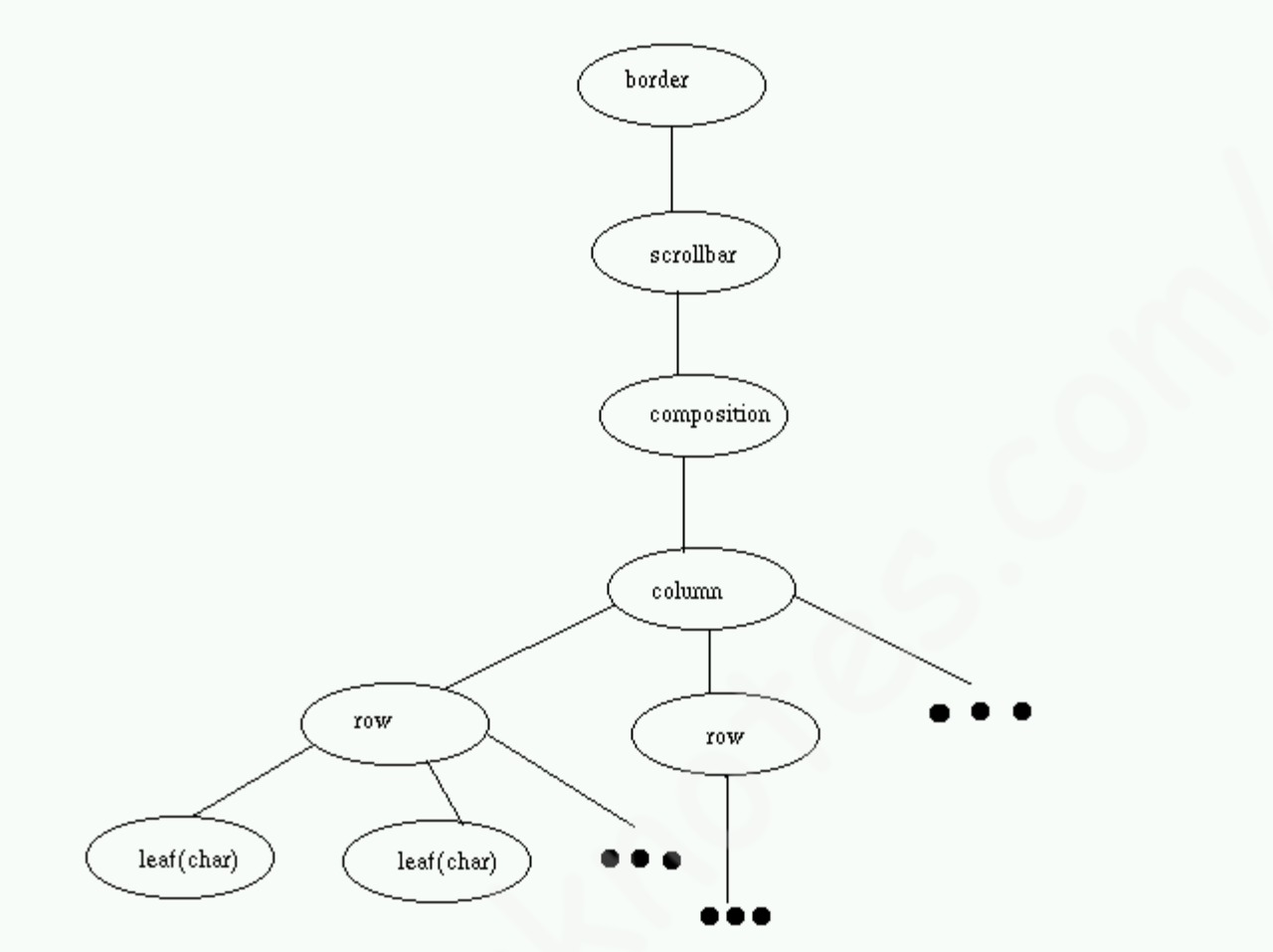


Embellishing User Interface:

- It should be possible to add and remove embellishments dynamically.
- The main idea used is that of transparent enclosure.
- This concept combines the notions of single child composition and compatible interfaces.
- The design pattern, Decorator, gives emphasis to transparent enclosure and single child composition.
- Embellishments such as borders and scrollbars can be added to the above document editor as shown in the class diagram below.
- Here the decorator is Monoglyph and the concrete decorators are Border and ScrollBar.
- Responsibilities can be easily added and detached dynamically using the structure given below.



Monoglyph serves as an abstract class for embellishment glyphs such as borders and scrollbars. The resulting object structure is as shown below.



Adding multiple look and feel standards:

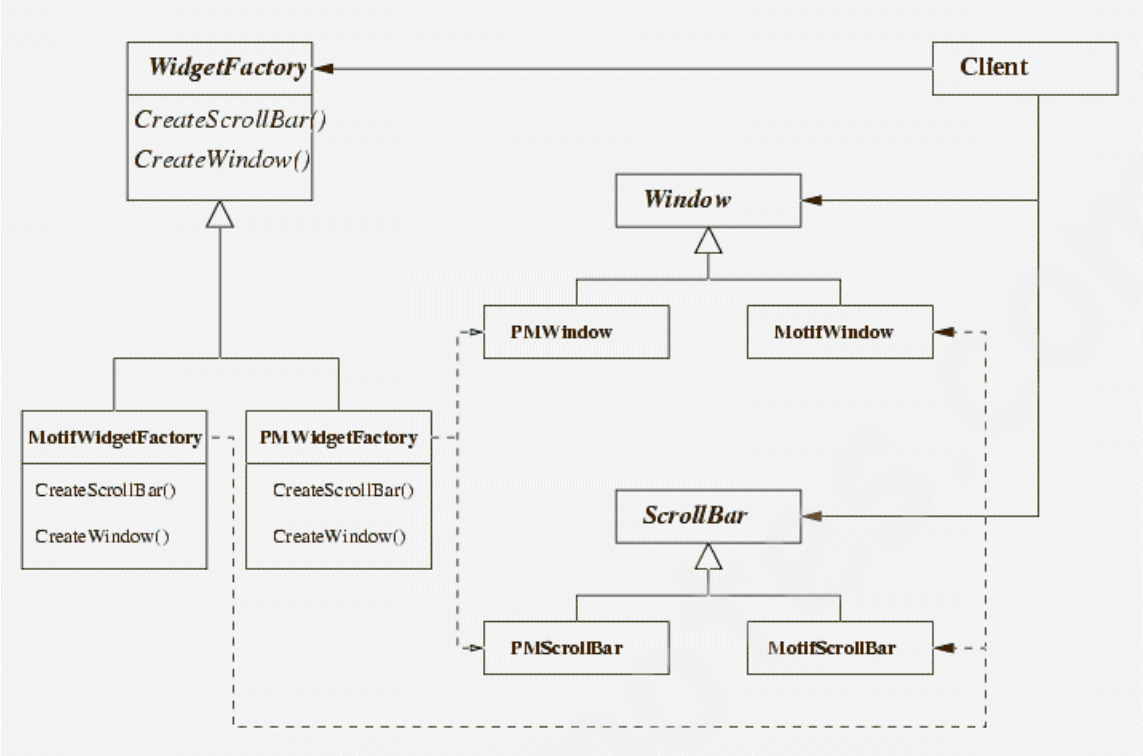
- It should be possible to support multiple look and feel standards like Presentation Manager, Motif, Mac.
- Each of these provide standard scrollbars, buttons and menus.
- Main idea used here is to postpone the creation of objects till they are required and to create them indirectly.
- Abstract Factory can be used for abstracting the process of object creation.
- Normally, we create an instance of the MotifFactory class using the following code:

```
ScrollBar *sb = new MotifScrollBar;
```

Here, the client is tied to a particular look and feel standard namely, Motif. This can be avoided using the following code.

```
ScrollBar *sb = guiFactory->CreateScrollBar();
GUIFactory *guiFactory = new MotifFactory;
```

 The resulting GUIFactory hierarchy is as shown below. The Abstract Factory is WidgetFactory, the concrete factories are MotifWidgetFactory and PMWidgetFactory and the abstract products are Window and Scrollbar.



Supporting multiple window systems:

 There can be multiple windowing systems such as Windows, X, Presentation Manager etc.
 Abstractions in this case are not as clear as in the case of look and feel standards. Hence Abstract Factory is not a good choice.
 Bridge pattern helps to create two separate class hierarchies, one that supports the logical notion of windows and the other for capturing the different implementations of the windows.
 It isolates abstraction from implementation.
 The different operations, the window class should support are:

- Window management:

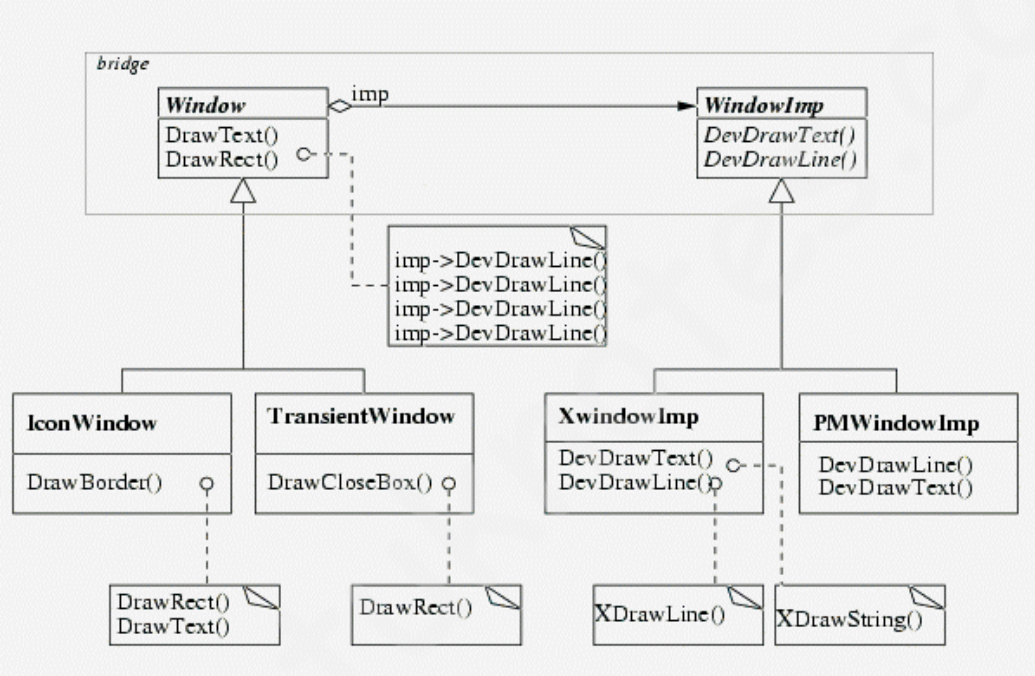
  virtual void redraw()
  virtual void raise()
  virtual void lower()
  virtual void iconify()
  virtual void deiconify()

- graphics:

  virtual void DrawLine()
  virtual void DrawRect()

virtual void DrawPolygon()
virtual void DrawText()

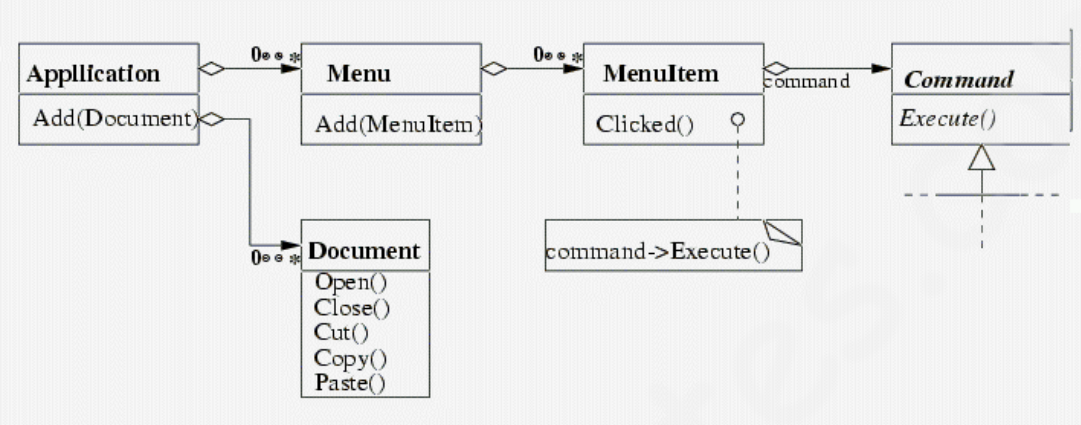
 Two extreme philosophies can be considered:make the interface extremely rich with all the functionalities.take the minimum common functionality.
 Neither extreme is a viable solution, so the window class will provide a convenient interface that supports the most popular features.
 The resulting hierarchy of window abstractions separating the abstraction from implementation is as follows. The Abstraction is Window, Implementor is WindowImp, the concrete implementors are XwindowImp and PMWindowImp and the RefinedAbstractions are IconWindow and TransientWindow.



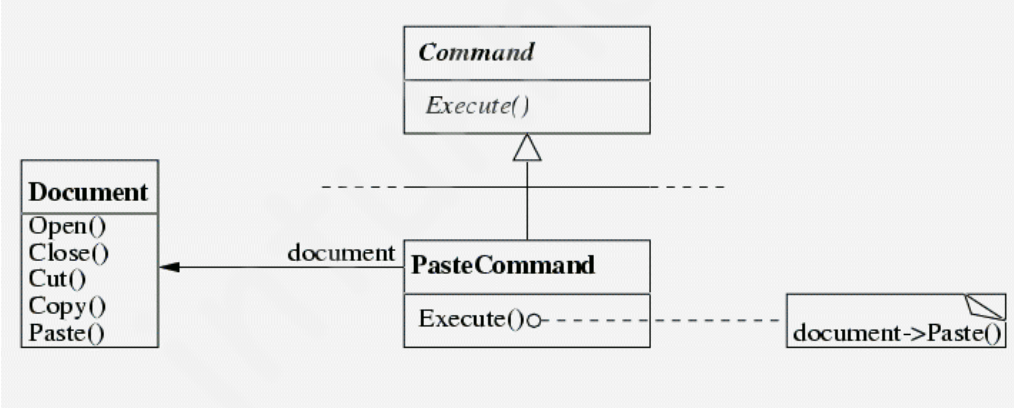
User operations:

- The document editor should provide functionalities like
  - creating a new document.
  - open a document.
  - save a document.
  - print a document.
  - cut and paste.
  - fonts.
  - alignment and justification.
  - undo and redo.
  - quit ...
- A lot of other factors are also to be considered like no undo for save or open, which are operation specific.

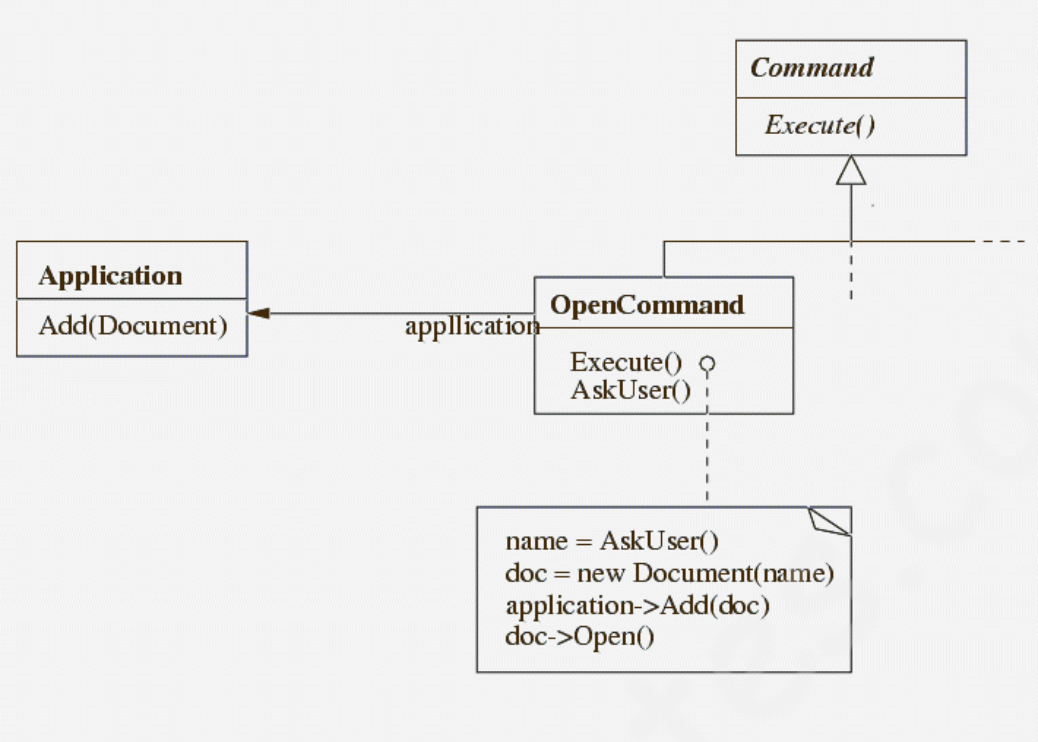
- For doing undo and redo, we need to keep track of the states. To store the state, we can use objects since each object has an identity, state and behavior.
- Define each operation as a command.
- The pattern Command can be used in which the emphasis is on encapsulating variation.
- The partial Command class hierarchy is as shown below.



The figure below shows the illustration of Command.

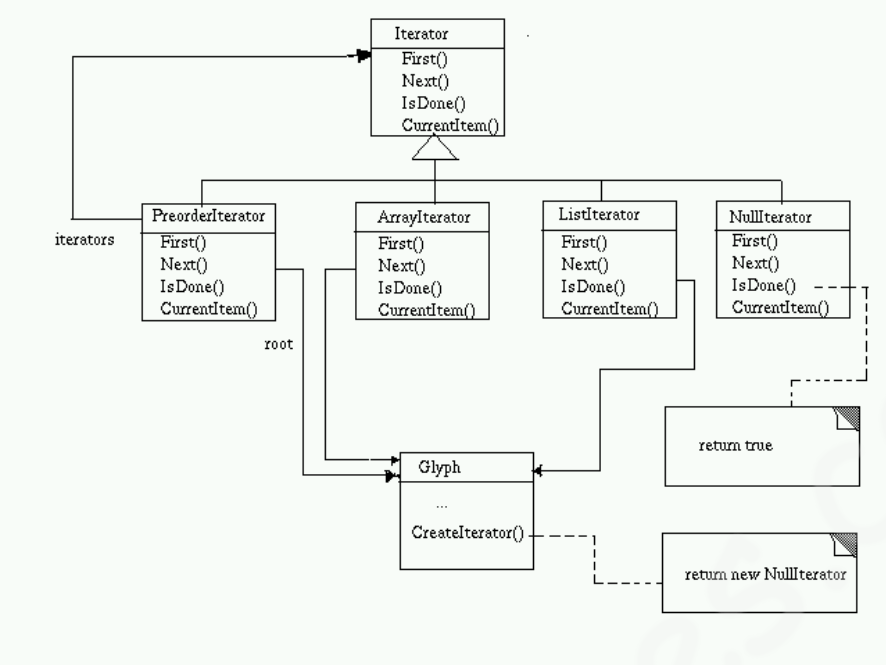


The usage of Command is depicted by the following figure.



Spell Checking and Hyphenation:

- Textual Analysis is to be done to check for misspellings and introducing hyphenation points.
- A diverse set of algorithms exist for spell checking and hyphenation.
- Two factors are to be considered:
    - Accessing the information to be analyzed which is scattered over the glyphs in the document structure.
        - Should handle different data structures used for storing the objects.
        - Support different kinds of traversals like preorder, postorder and inorder.
    - Doing the analysis.
- The design pattern, Iterator can be used to provide a general interface for access and traversal.
- Subclasses can be created for handling different data structures like arrays and lists and for handling different traversals like preorder, postorder etc.
- The operations like First(), Next() and IsDone () are used for the traversal.
- The resulting Iterator classes and subclasses are as follows.



- It is better to keep the traversal and analysis phases separate because most often different analyses require the same kind of traversal.
- Analysis can be done using the design pattern, Visitor.
- Visitor class represents an abstract interface for visiting glyphs in a structure.
- Concrete subclasses of Visitor like SpellCheckerVisitor and HyphenationVisitor perform different analyses like spell checking and hyphenation.
- The class, Glyph includes an operation Accept(Visitor v).
- When it accepts a visitor, it sends a request to the visitor with the Glyph element as argument.
- The visitor will then execute the operation for that element- spell checking and hyphenation.
- Adding a new analysis requires just defining a new subclass of Visitor.