DESIGN PATTERNS
[R15A0528]
LECTURE NOTES

B.TECH IV YEAR – I SEM

(R15)

(2019-20)

DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

Recognized under 2(f) and 12 (B) of UGC ACT 1956

(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)

Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, India

IV Year B. Tech. CSE –I Sem

L	T/P/D	C
4	1/- / -	3

(R15A0528) DESIGN PATTERNS

Objectives:

- Design patterns are a systematic approach that focus and describe abstract systems of interaction between classes, objects, and communication flow
- Given OO design heuristics, patterns or published guidance, evaluate a design for applicability, reasonableness, and relation to other design criteria.
- Comprehend the nature of design patterns by understanding a small number of examples from different pattern categories, and to be able to apply these patterns in creating an OO design.
- Good knowledge on the documentation effort required for designing the patterns.

Outcomes:

Upon completion of this course, students should be able to:

- Have a deeper knowledge of the principles of object - oriented design
- Understand how these patterns related to object - oriented design.
- Understand the design patterns that are common in software applications.
- Will able to use patterns and have deeper knowledge of patterns.
- Will be able to document good design pattern structures.

UNIT I:

Introduction: What Is a Design Pattern? Design Patterns in Smalltalk MVC, Describing Design Patterns, The Catalog of Design Patterns, Organizing the Catalog, How Design Patterns Solve Design Problems, How to Select a Design Pattern, How to Use a Design Pattern.

UNIT II:

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

Creational Patterns: Abstract Factory, Builder, Factory Method, Prototype, Singleton, Discussion of Creational Patterns.

UNIT III:

Structural Pattern Part - I: Adapter, Bridge, and Composite

Structural Pattern Part - II: Decorator, Façade, Flyweight, Proxy.

UNIT IV:

Behavioral Patterns Part - I: Chain of Responsibility, Command

Behavioral Patterns Part - II: Mediator, Memento, Observer

UNIT V:

Behavioral Patterns Part – II(cont'd): State, Strategy, Template Method, Visitor, Discussion of Behavioral Patterns. What to Expect from Design Patterns, A Brief History, The Pattern Community An Invitation, A Parting Thought.

TEXT BOOK:

1. Design Patterns by Erich Gamma, Pearson Education

References:

1. Pattern's in Java Vol-I by Mark Grand, Wiley DreamTech.
2. Pattern's in Java Vol-II by Mark Grand, Wiley DreamTech.
3. Java Enterprise Design Patterns Vol-III by Mark Grand, Wiley DreamTech.
4. Head First Design Patterns by Eric Freeman – Oreilly-spd.
5. Design Patterns Explained by Alan Shalloway, Pearson Education.

INDEX

UNIT -I

Introduction

What is a Design pattern?

- Each pattern Describes a problem which occurs over and over again in our environment ,and then describes the core of the problem
- Novelists, playwrights and other writers rarely invent new stories.
- Often ideas are reused, such as the “Tragic Hero” from Hamlet or Macbeth.
- Designers reuse solutions also, preferably the “good” ones
  - Experience is what makes one an ‘expert’
- Problems are addressed without rediscovering solutions from

scratch. “My wheel is rounder.

Design Patterns are the best solutions for the re-occurring problems in the application programming environment.

- Nearly a universal standard.
- Responsible for design pattern analysis in other areas, including GUIs.
- Mainly used in Object Oriented programming.

Design Pattern Elements

1. **Pattern Name**

   Handle used to describe the design

   problem. Increases vocabulary.

   Eases design discussions.

   Evaluation without implementation details.

2. **Problem**

   Describes when to apply a pattern.

   May include conditions for the pattern to be

   applicable. Symptoms of an inflexible design or

   limitation.

3. **Solution**

   Describes elements for the design.

   Includes relationships, responsibilities, and

   collaborations. Does not describe concrete designs or

implementations.

A pattern is more of a template.

4. **Consequences**

Results and Trade Offs.

Critical for design pattern

evaluation. Often space and time

trade offs.

Language strengths and limitations.

(Broken into benefits and drawbacks for this discussion).

Design patterns can be subjective.

One person's pattern may be another person's primitive building block.

The focus of the selected design patterns

are: Object and class

communication.

Customized to solve a general design

problem. Solution is context specific.

Design patterns in Smalltalk MVC:

- The Model/View/Controller triad of classes is used to build user interfaces in Smalltalk-80
- MVC consists of three kinds of objects.
- M->>MODEL is the Application object.
- V->>View is the screen presentation.
- C->>Controller is the way the user interface reacts to user input MVC decouples to increase flexibility and reuse.

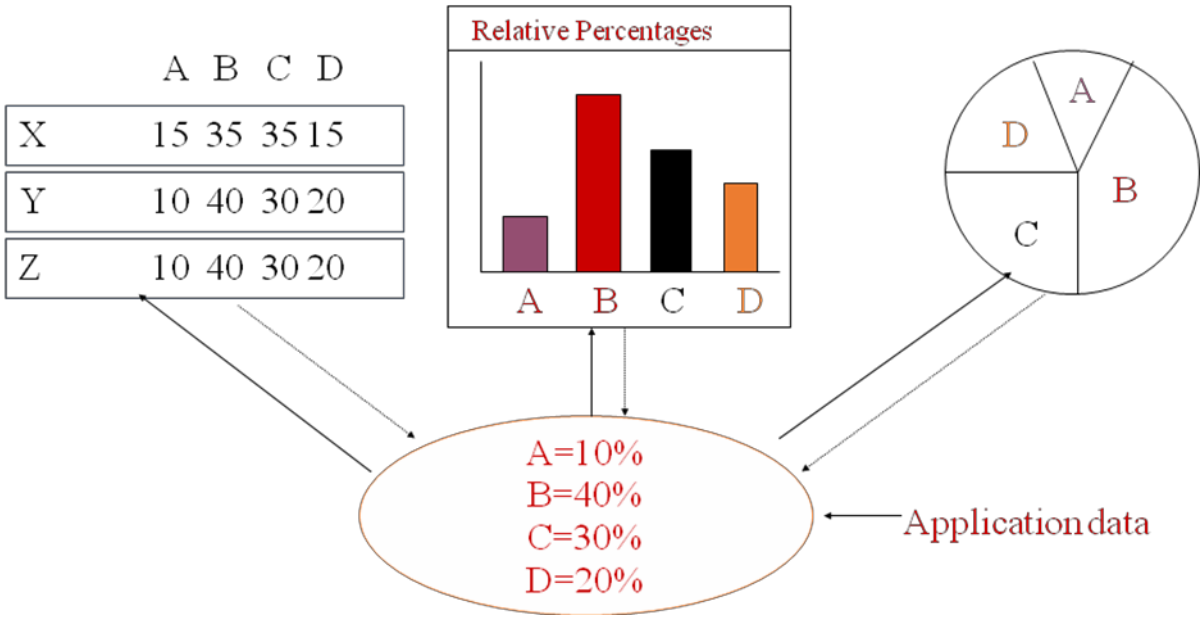
MVC decouples views and models by establishing a subscribe/notify protocol between them. A view must ensure that its appearance must reflects the state of the model. Whenever the model's data changes, the model notifies views that depends on it. You can also create new views for a model without Rewriting it.

- The below diagram shows a model and three views.
- The model contains some data values, and the views defining a

spreadsheet, histogram, and pie chart display these data in various ways.

- The model communicates with it's values change, and the views communicate with the model to access these values.
- Feature of MVC is that views can be nested.

Easy to maintain and enhancement.



Describing Design Patterns:

- Graphical notations ,while important and useful, aren't sufficient.
  They capture the end product of the design process as relationships between classes and objects.
  By using a consistent format we describe the design pattern .
  Each pattern is divided into sections according to the following template**.**
- **Pattern Name and Classification:**
- it conveys the essence of the pattern succinctly good name is vital, because it will become part of design vocabulary.
- **Intent:** What does the design pattern do?
- What is it's rational and intend?
- What particular design issue or problem does it address?
- **Also Known As:** Other well-known names for the pattern, if any.
- **Motivation:**
- A scenario that illustrates a design problem and how the class and object

structures in the pattern solve the problem.

- The scenario will help understand the more abstract description of the pattern that follows

4

Applicability:

- Applicability: What are the situations in which the design patterns can be applied?
- What are example of the poor designs that the pattern can address?
- How can recognize situations?
- Structure: Graphical representation of the classes in the pattern using a notation based on the object Modeling Technique(OMT).
- Participants: The classes and/or objects participating in the design pattern and their responsibilities.

Structure:

- Graphical representation of the classes in the pattern using a notation based on the object Modeling Technique(OMT).

Participants:

The classes and/or objects participating in the design pattern and their responsibilities.

Collaborations:

- How the participants collaborate to carry out their responsibilities.

Consequences:

- How does the pattern support its objectives?
- What are the trade-offs and result of using the pattern ?
- What aspect of the system structure does it let vary independently?

Implementation:

- What pitfalls,hints,or techniques should be aware of when implementing the pattern ?
- Are there language-specific issues?

Sample Code:

- Code fragments that illustrate how might implement the pattern in c++ or Smalltalk.

Known Uses:

Examples of the pattern found in real systems.

Related Patterns:

What design patterns are closely related to this one? What are the imp

differences? With Which other patterns should this one be used?

The Catalog of Design Pattern:

Abstract Factory: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Adaptor: Convert the interface of a class into another interface clients expect.

Bridge: Decouple an abstraction from its implementation so that two can vary independently.

- **Builder:**
- Separates the construction of the complex object from its representation so that the same constriction process can create different representations.
- **Chain of Responsibility:** Avoid coupling the sender of a request to it's receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an objects handles it.
- **Command:**
- Encapsulate a request as an object ,thereby letting parameterize clients with different request, queue or log requests, and support undoable operations.
- **Composite:**

Compose objects into three objects to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

- **Decorator:**
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.
- **Façade:** Provide a unified interface to a set of interfaces in a subsystem's Facade defines a higher-level interface that makes the subsystem easier to use.
- **Factory Method:**
- Defines an interface for creating an object ,but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **Flyweight:**
- Use sharing to support large numbers of fine-grained objects efficiently.
- **Interpreter:**
- Given a language, defining a representation of its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- **Memento:** Without violating encapsulation, capture and externalize an object's internal state so that object can be restored to this state later.

- **Observer:** Define a one-to-many dependency between objects so that when one object changes state, all it's dependents are notified and updated automatically.
- **Prototype:**
- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- **Proxy:** Provide a surrogate or placeholder for another object to control access to it.
- **Singleton:** Ensure a class has only one instance, and provide a point of access to it.
- **State:**
- Allow an object to alter its behavior when its internal state changes. the object will appear to change its class.
- **Strategy:**
- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- **Template Method:**
- Define the Skelton of an operation, deferring some steps to subclasses. Template method subclasses redefine certain steps of an algorithm without changing the algorithms structure.
- **Visitor:**

  Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Purpose: what a pattern does

Creational: concern the process of object creation

Structural: the composition of classes or objects

Behavioral: characterize the ways in which classes or objects interact and distribute responsibility

Scope: whether the pattern applies primarily to classes or to ob

How Design Patterns Solve Design Problems:

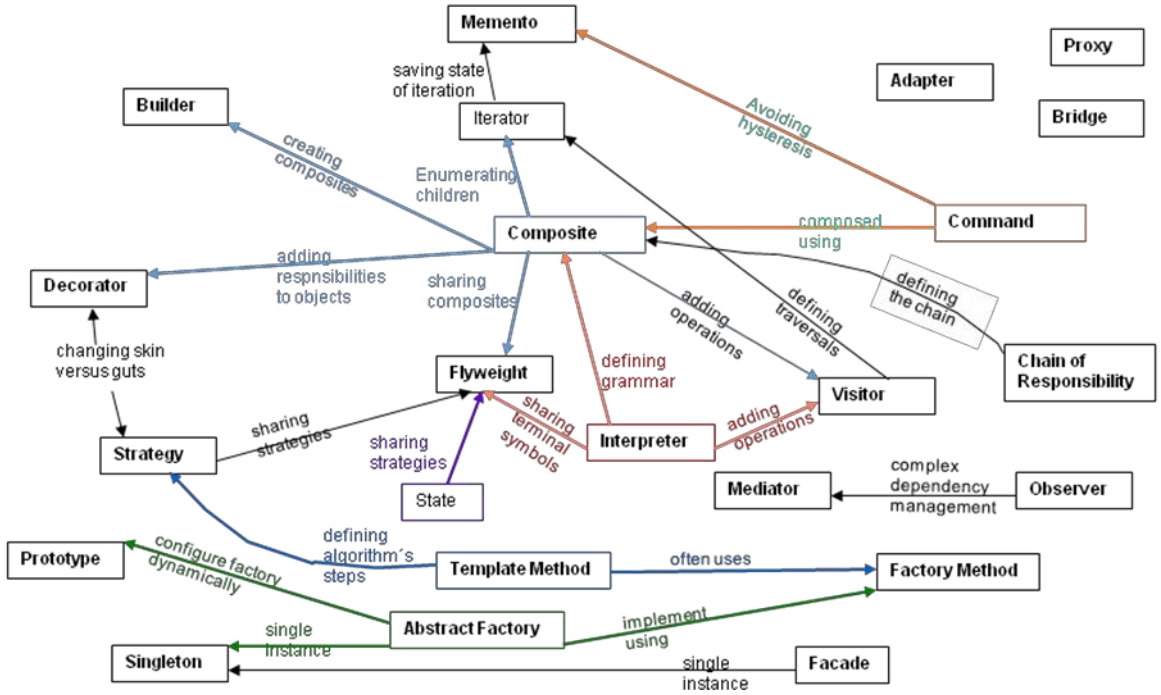
- Finding Appropriate Objects
  - Decomposing a system into objects is the hard par.t
  - OO-designs often end up with classes with no counterparts in real world (low- level classes like arrays).

Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrows.

  - Design patterns identify less-obvious abstractions.

- Determining Object Granularity
  - Objects can vary tremendously in size and number
  - **Facade pattern** describes how to represent subsystems as objects
  - **Flyweight pattern** describes how to support huge numbers of objects

- Mapping



8

Specifying Object Interfaces:

- Interface:
  - Set of all signatures defined by an object's operations.
  - Any request matching a signature in the objects interface may be sent to the object.
  - Interfaces may contain other interfaces as subsets.
- Type:
  - Denotes a particular interfaces.
  - An object may have many types.
  - Widely different object may share a type.
  - Objects of the same type need only share parts of their interfaces.
  - A **subtype** contains the interface of its **super type.**
- Dynamic binding, polymorphism.

- An object's implementation is defined by its ***class***
- The class specifies the object's internal data and defines the operations the object can perform
- Objects is created by ***instantiating*** a class
  - an object = an *instance* of a class
- ***Class inheritance***
  - parent class and subclass

- ***Abstract class*** versus ***concrete class***
  - abstract operations.
- ***Override*** an operation.
- Class versus type:
  - An object's *class* defines how the object is implemented.
  - An object's *type* only refers to its interface.

  - An object can have many types, and objects of different classes can have the same type.

- Class versus Interface Inheritance
  - *class inheritance* defines an object's implementation in terms of another object's implementation (code and representation sharing).
  - *interface inheritance* (or *subtyping*) describes when an object can be used in place of another.
- Many of the design patterns depend on this distinction.

Programming to an Interface, not an Implementation

- Benefits
  - clients remain unaware of the specific types of objects they use.
  - clients remain unaware of the classes that implement these objects.
- Manipulate objects solely in terms of interfaces defined by abstract classes!
- Benefits:
  - Clients remain unaware of the specific types of objects they use.
  - Clients remain unaware of the classes that implement the objects. Clients only know about abstract class(es) defining the interfaces
  - Do not declare variables to be instances of particular concrete classes
  - Use creational patterns to create actual objects.

Favor object composition over class inheritance

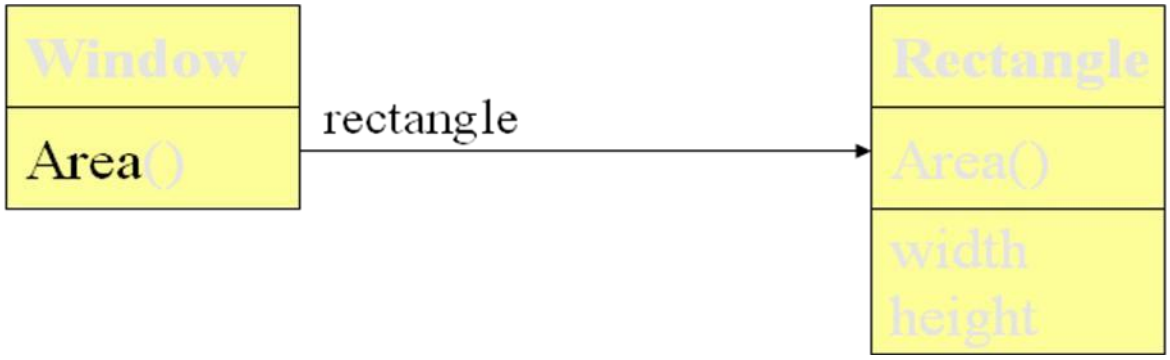
- **White-box** reuse:
  - Reuse by subclassing (class inheritance)
  - Internals of parent classes are often visible to subclasses
  - works statically, compile-time approach
  - Inheritance breaks encapsulation
- **Black-box** reuse:
  - Reuse by object composition
  - Requires objects to have well-defined interfaces
  - No internal details of objects are visible

Inheritance versus Composition

- Two most common techniques for reuse
  - class inheritance
    - white-box reuse
  - *object composition*
    - black-box reuse
- **Class inheritance**
  - advantages
    - static, straightforward to use.
    - make the implementations being reuse more easily.
- **Class inheritance (cont.)**
  - disadvantages
    - the implementations inherited can't be changed at run time.
    - parent classes often define at least part of their subclasses' physical representation.
    - breaks encapsulation.
    - implementation dependencies can cause problems when you're trying to reuse a subclass.
- **Object composition**
  - dynamic at run time.
  - composition requires objects to respect each others' interfaces.
    - but does not break encapsulation.
  - any object can be replaced at run time.
  - Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task.
- Object composition (cont.)
  - class and class hierarchies will remain small.
  - but will have more objects.

Delegation:

- Two objects are involved in handling a request: a receiving object delegates operations to its delegate.



- Makes it easy to compose behaviors at run-time and to change the way they're composed.
- Disadvantage:dynamic, highly parameterized software is harder to understand than more static software.
- Delegation is a good design choice only when it simplifies more than it complicates.
- Delegation is an extreme example of object composition.

Inheritance versus Parameterized Types

- Let you define a type without specifying all the other types it uses, the unspecified types are supplied as parameters at the point of use.
- Parameterized types, generics, or templates.
- Parameterized types give us a third way to compose behavior in object-oriented systems.
- Three ways to compose
  - object composition lets you change the behavior being composed at run-time, but it requires indirection and can be less efficient.
  - inheritance lets you provide default implementations for operations and lets subclasses override them.
  - parameterized types let you change the types that a class can use.

Relating Run-Time and Compile-Time Structures:

- An object-oriented program's run-time structure often bears little resemblance to its code structure.
- The code structure is frozen at compile-time.
- A program's run-time structure consists of rapidly changing networks of communicating objects.
- The distinction between acquaintance and aggregation is determined more by intent than by explicit language mechanisms
- The system's run-time structure must be imposed more by the designer than the language.
- The distinction between acquaintance and aggregation is determined more by intent than by explicit language mechanisms.
- The system's run-time structure must be imposed more by the designer than the language.

Designing for Change:

- A design that doesn't take change into account risks major redesign in the future.
- Design patterns help you avoid this by ensuring that a system can change in specific ways
  - each design pattern lets some aspect of system structure vary independently of other aspects.

Common Causes of Redesign:

- Creating an object by specifying a class explicitly.
- Dependence on specific operations.
- Dependence on hardware and software platform.
- Dependence on object representations or implementations.
- Algorithmic dependencies.

Common Causes of Redesign

(cont.)

- Tight coupling.
- Extending functionality by subclassing .
- Inability to alter classes conveniently.

Design for Change (cont.)

- Design patterns in application programs.
  - Design patterns that reduce dependencies can increase internal reuse.
  - Design patterns also make an application more maintainable when they're used to limit platform dependencies and to layer a system.
- Design patterns in toolkits
  - A ***toolkit*** is a set of related and reusable classes designed to provide useful, general-purpose functionality.
  - Toolkits emphasize code reuse. They are the object-oriented equivalent of subroutine libraries.
  - Toolkit design is arguably harder than application design.
- Design patterns in framework
  - A framework is a set of cooperating classes that make up a reusable design for a specific class of software.
  - You customize a framework to a particular application by creating application- specific subclasses of abstract classes from the framework.
  - The framework dictates the *architecture* of your application.

- Design patterns in framework (cont.)
  - Frameworks emphasize *design reuse* over code reuse.
  - When you use a *toolkit*, you write the main body of the application and call the code you want to reuse. When you use a *framework*, you reuse the main body and write the code ***it*** calls.
  - Advantages: build an application faster, easier to maintain, and more consistent to their users.
- Design patterns in framework (cont.)
  - Mature frameworks usually incorporate several design patterns.
  - People who know the patterns gain insight into the framework faster.
  - differences between framework and design pattern.
    - design patterns are more abstract than frameworks.
    - design patterns are smaller architectural elements than frameworks.
    - design patterns are less specialized than frameworks.

How To Select a Design Pattern:

- Consider how design patterns solve design problems.
- Scan Intent sections.
- Study how patterns interrelate.
- Study patterns of like purpose.
- Examine a Cause of redesign.
- Consider what should be variable in your design**.**

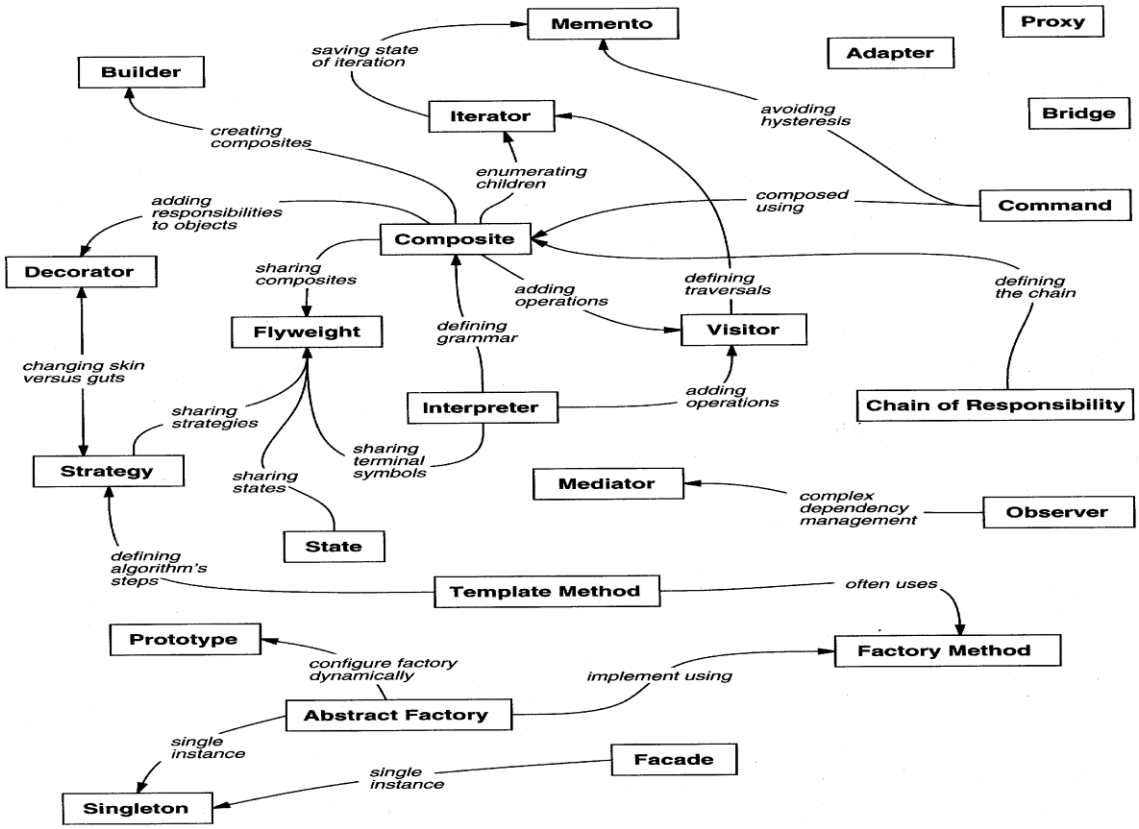


Figure 1.1: Design pattern relationships

- Read the pattern once through for an overview.
- Go Back and study the Structure, Participants ,and Collaborations sections.

- Look At the Sample Code section to see a concrete Example of the pattern in code.

Choose names for pattern participants that are meaningful in the application context. Define the classes.

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151)	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location
Behavioral	Chain of Responsibility (223)	object that can fulfill a request
	Command (233)	when and how a request is fulfilled
	Interpreter (243)	grammar and interpretation of a language
	Iterator (257)	how an aggregate's elements are accessed, traversed
	Mediator (273)	how and which objects interact with each other
	Memento (283)	what private information is stored outside an object, and when
	Observer (293)	number of objects that depend on another object; how the dependent objects stay up to date
	State (305)	states of an object
	Strategy (315)	an algorithm
	Template Method (325)	steps of an algorithm
	Visitor (331)	operations that can be applied to object(s) without changing their class(es)

Table 1.2: Design aspects that design patterns let you vary

Unit –II

A Case Study

Design Problems:

- seven problems in Lexis's design:

Document Structure:

- ✓ The choice of internal representation for the document affects nearly every aspect of Lexis's design. All editing , formatting, displaying, and textual analysis will require traversing the representation.

Formatting:

- ✓ How does Lexi actually arrange text and graphics into lines and columns?
- ✓ What objects are responsible for carrying out different formatting policies?
- ✓ How do these policies interact with the document's internal representation?

Embellishing the user interface:

Lexis user interface include scroll bar, borders and drop shadows that embellish the WYSIWYG document interface. Such embellishments are likely to change as Lexis user interface evolves.

Supporting multiple look-and-feel standards:

Lexi should adapt easily to different look-and-feel standards such as Motif and Presentation Manager (PM) without major modification.

Supporting multiple window systems:

Different look-and-fell standards are usually implemented on different window system. Lexi's design should be independent of the window system as possible.

User Operations:

User control Lexi through various interfaces, including buttons and pull-down menus. The functionality beyond these interfaces is scattered throughout the objects in the application.

Spelling checking and Hyphenation:

How does Lexi support analytical operations checking for misspelled words and determining hyphenation points? How can we minimize the number of classes we have to modify to add a new analytical operation?

Document

Structure: Goals:

- present document's visual aspects
- drawing, hit detection, alignment
- support physical structure (e.g., lines, columns)

Constraints/forces:

- treat text & graphics uniformly.
- no distinction between one & many.

The internal representation for a document:

- The internal representation should support.
- maintaining the document's physical structure.
- generating and presenting the document visually.
- mapping positions on the display to elements in the internal representations.

Some constraints:

- we should treat text and graphics uniformly.
- our implementation shouldn't have to distinguish between single elements and groups of elements in the internal representation.

Recursive Composition:

- a common way to represent hierarchically structured information.

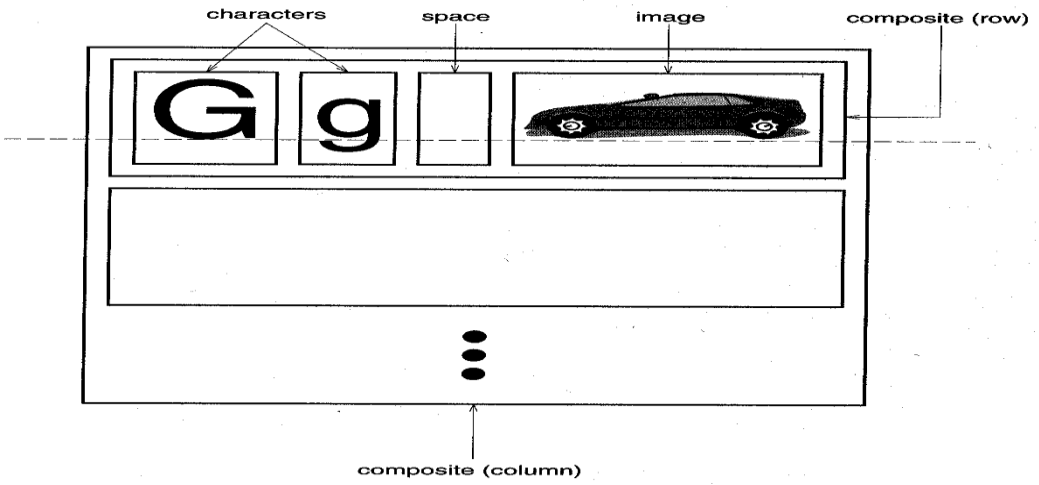


Figure 2.2: Recursive composition of text and graphics

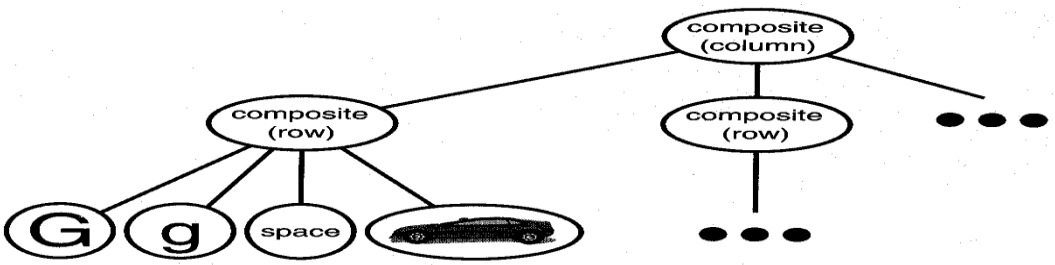


Figure 2.3: Object structure for recursive composition of text and graphics

Glyphs:

– an abstract class for all objects that can appear in a document structure. Three basic responsibilities, they know

- How to draw themselves
- What space they occupy
- Their children and parent.

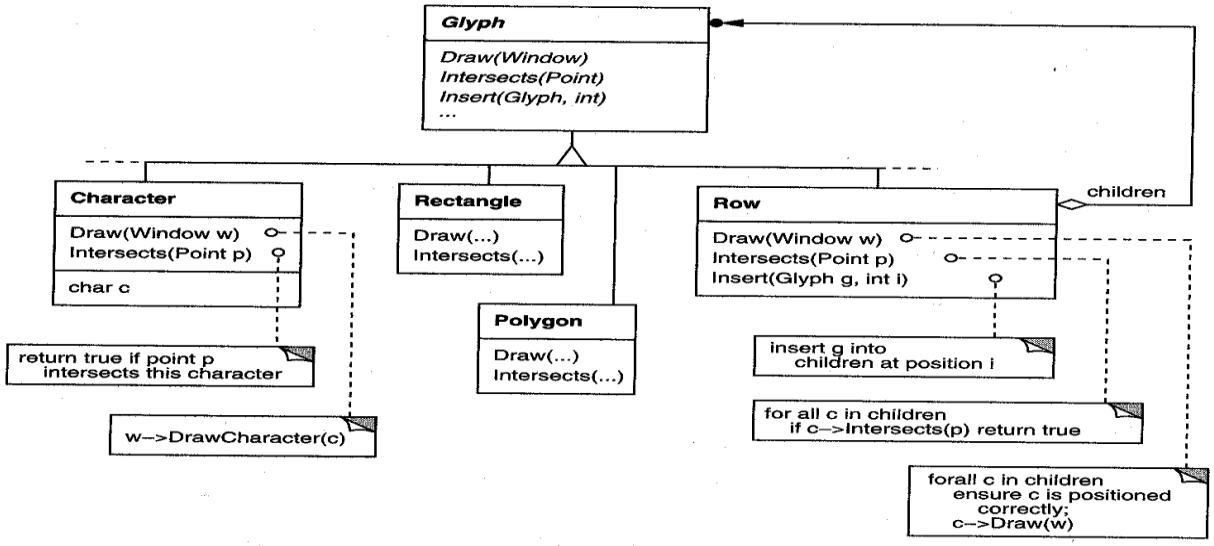


Figure 2.4: Partial Glyph class hierarchy

Responsibility	Operations
appearance	`virtual void Draw(Window*)` `virtual void Bounds(Rect&)`
hit detection	`virtual bool Intersects(const Point&)`

structure	`virtual void Insert(Glyph*, int)`
	`virtual void Remove(Glyph*)`
	`virtual Glyph* Child(int)`
	`virtual Glyph* Parent()`

Table 2.1: Basic glyph interface

Formatting :

- A structure that corresponds to a properly formatted document.
- Representation and formatting are distinct
  - the ability to capture the document's physical structure doesn't tell us how to arrive at a particular structure.
- here, we'll restrict "formatting" to mean breaking a collection of glyphs in to lines.

- **Encapsulating the formatting algorithm:**
  - keep formatting algorithms completely independent of the document structure.
  - make it is easy to change the formatting algorithm.
  - We'll define a separate class hierarchy for objects that encapsulate

formatting algorithms.

- **Compositor and Composition:**
  - We'll define a ***Compositor*** class for objects that can encapsulate a formatting algorithm.
  - The glyphs Compositor formats are the children of a special Glyph subclass called ***Composition***.
  - When the composition needs formatting, it calls its compositor's *Compose* operation.
  - Each Compositor subclass can implement a different line breaking algorithm.

Responsibility	Operations
what to format	`void SetComposition(Composition*)`
when to format	`virtual void Compose()`

Table 2.2: Basic compositor interface

Compositor and Composition (cont):

- The Compositor-Composition class split ensures a strong *separation* between code that supports the document's physical structure and the code for different formatting algorithms.

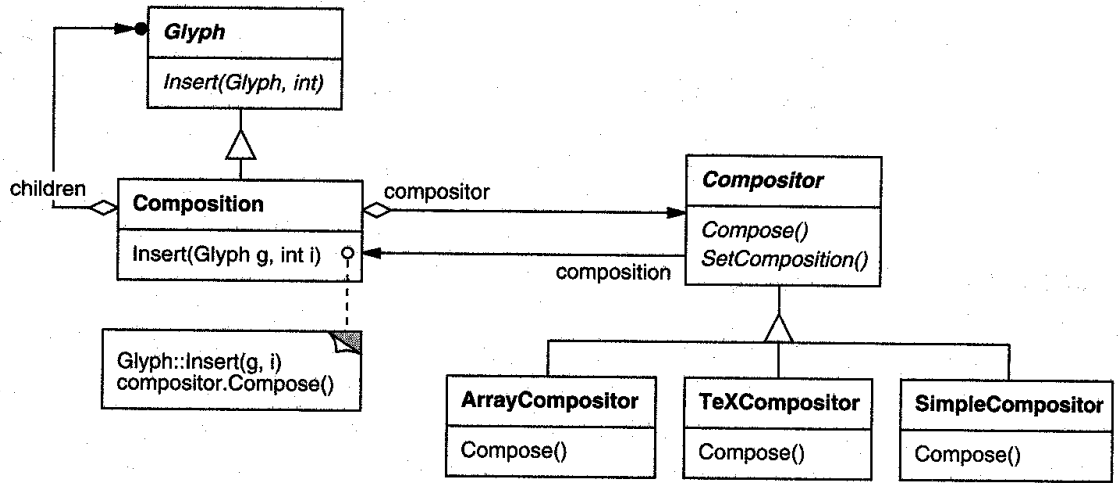


Figure 2.5: Composition and Compositor class relationships

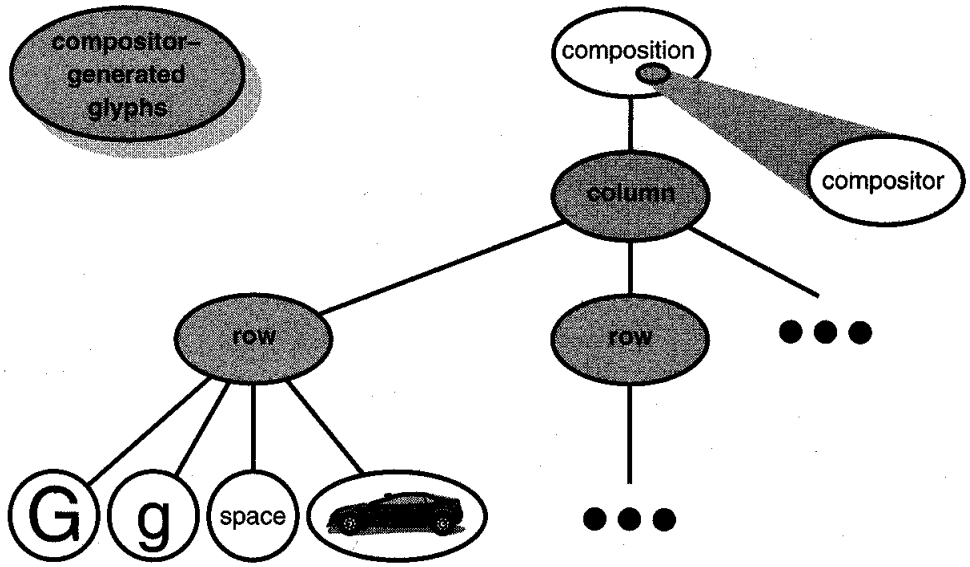


Figure 2.6: Object structure reflecting compositor-directed linebreaking

- **Strategy pattern:**
  - intent: encapsulating an algorithm in an object.
  - Compositors are strategies. A composition is the context for a compositor strategy.

Embellishing the User Interface:

- Considering adds a border around the text editing area and scrollbars that let the user view the different parts of the page here
- **Transparent Enclosure:**
  - inheritance-based approach will result in some problems.
  - Composition, ScollableComposition, BorderedScrollableComposition.
  - object composition offers a potentially more workable and flexible extension mechanism.
- **Transparent enclosure (cont):**
  - object composition (cont)
    - Border and Scroller should be a subclass of Glyph.
  - two notions
    - single-child (single-component) composition.
    - compatible interfaces.
- **Monoglyph**
  - We can apply the concept of transparent enclosure to all glyphs that embellish other glyphs.
  - the class, Monoglyph .

```
void MonoGlyph::Draw(Window* w) {
        _component-> Draw(w);
}
void Border:: Draw(Window * w) {
        MonoGlyph::Draw(w);
        DrawBorder(w);
}
```

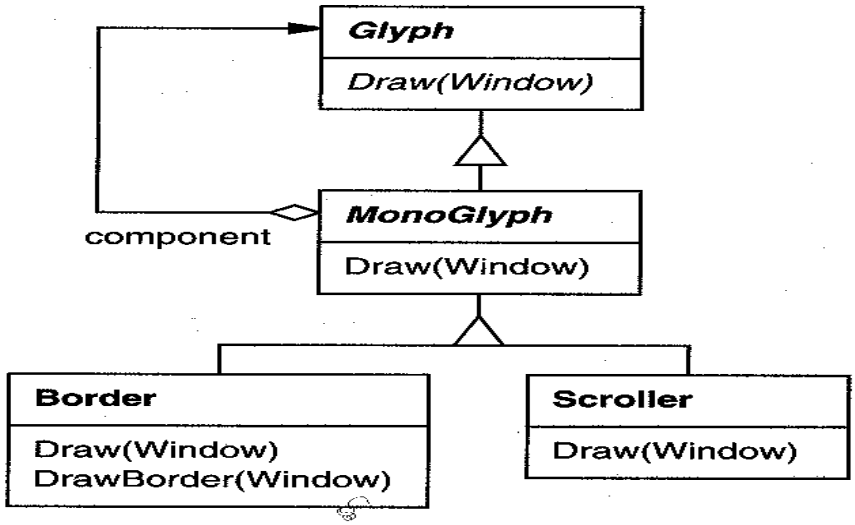


Figure 2.7: MonoGlyph class relationships

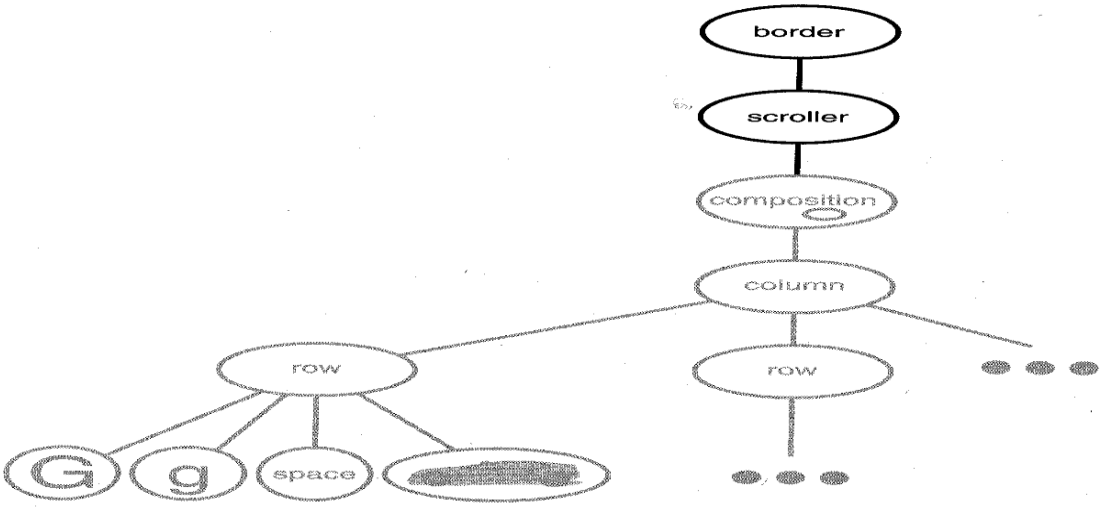
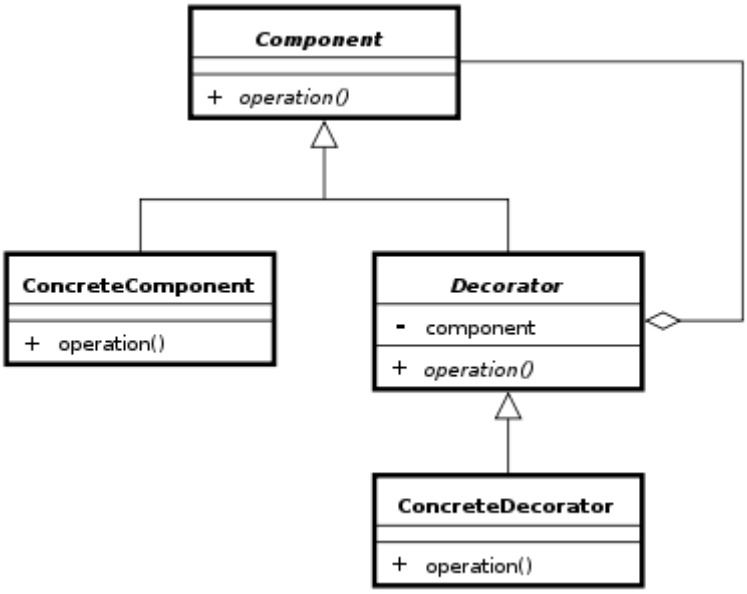


Figure 2.8: Embellished object structure

- **Decorator Pattern**
  - captures class and object relationships that support embellishment by transparent enclosure.



Supporting Multiple Look-and-Feel Standards:

- Design to support the look-and-feel changing at run-time
- Abstracting Object Creation
  - widgets
  - two sets of widget glyph classes for this purpose
    - a set of abstract glyph subclasses for each category of widget glyph (e.g., ScrollBar).
    - a set of concrete subclasses for each abstract subclass that implement different look-and-feel standards (e.g., MotifScrollBar and PMScrollBar).

Abstracting Object Creation (cont):

- Lexi needs a way to determine the look-and-feel standard being targeted
- We must avoid making explicit constructor calls
- We must also be able to replace an entire widget set easily
- We can achieve both by abstracting the process of object creation

- **Factories and Product Classes:**
  - **Factories** create **product** objects.
- **Abstract Factory Pattern:**
  - capture how to create families of related product objects without instantiating classes directly.

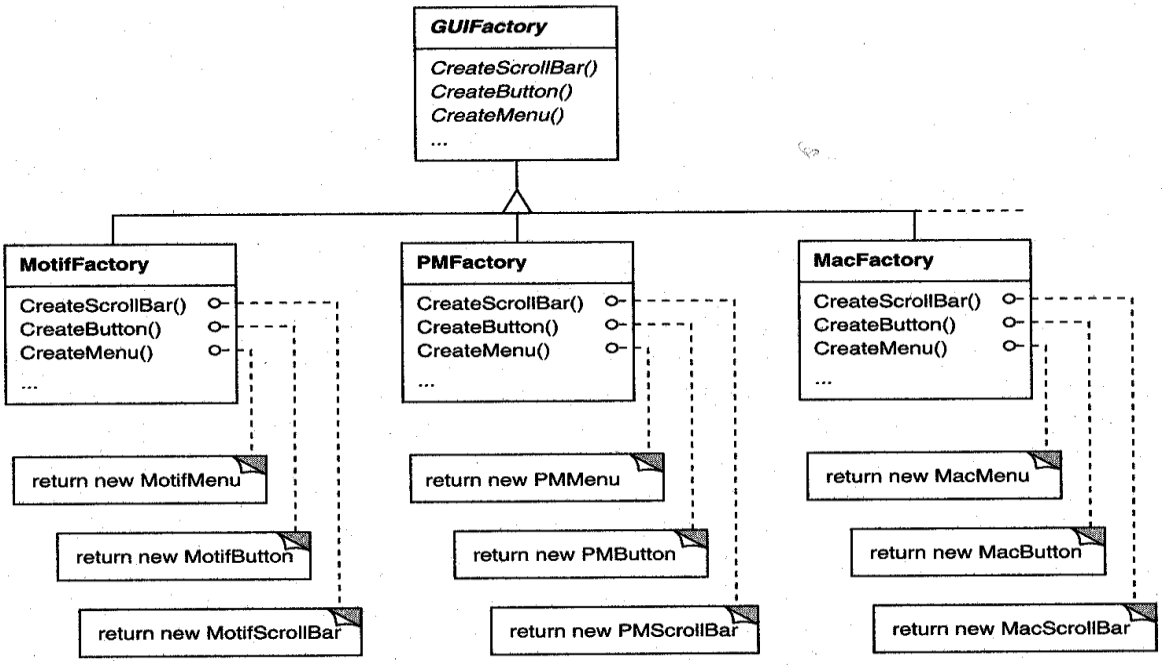


Figure 2.9: GUIFactory class hierarchy

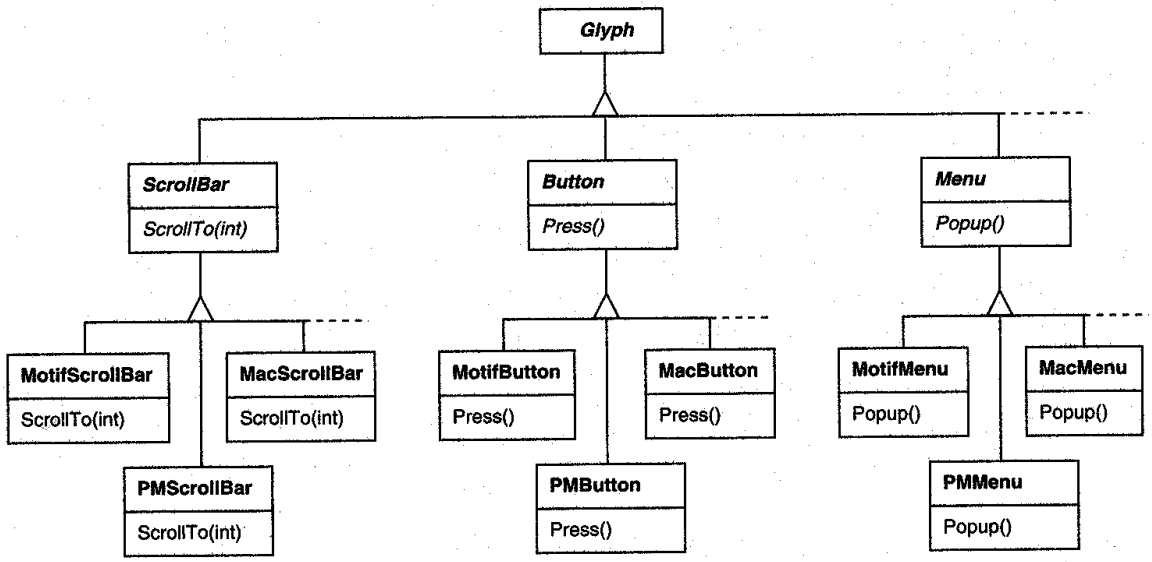


Figure 2.10: Abstract product classes and concrete subclasses

Supporting Multiple Window Systems:

- We'd like Lexi to run on many existing window systems having different programming interfaces.
- Can we use an Abstract Factory?
  - As the different programming interfaces on these existing window systems, the Abstract Factory pattern doesn't work.
  - We need a uniform set of windowing abstractions that lets us take different window system impelementations and slide any one of them under a common interface.
- Encapsulating Implementation Dependencies
  - The Window class interface encapsulates the things windows tend to do across window systems
  - The Window class is an abstract class
  - Where does the implementation live?

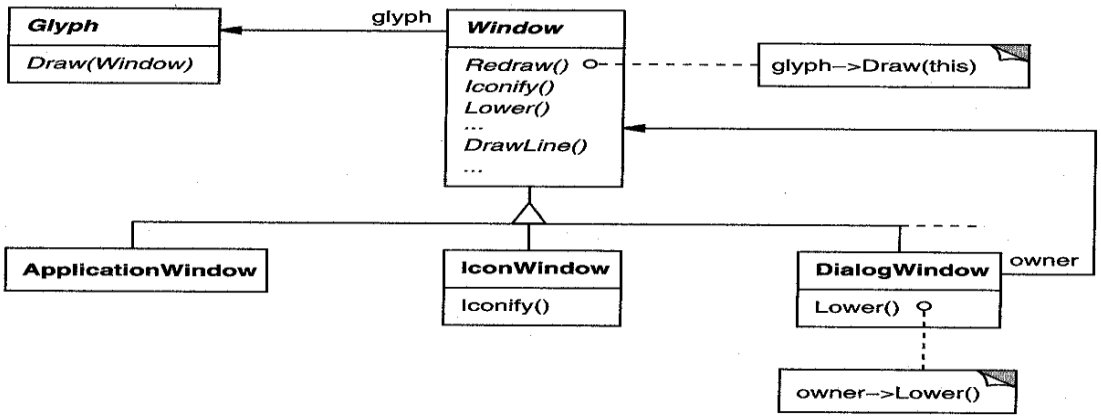
- Window and WindowImp :

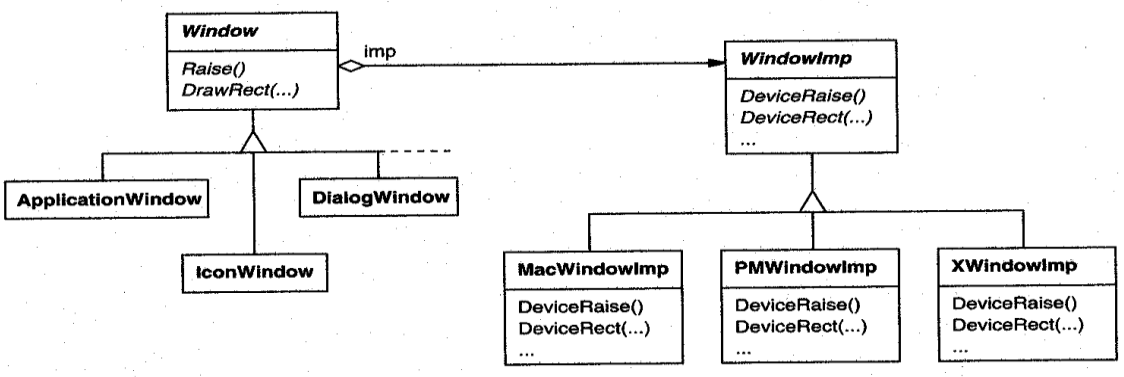
Responsibility	Operations

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(...)`
	`...`

Table 2.3: Window class interface





- Bridge Pattern
  - to allow separate class hierarchies to work together even as they evolve independently.

User Operations:

- Requirements
  - Lexi provides different user interfaces for the operations it supported.
  - These operations are implemented in many different classes.
  - Lexi supports undo and redo.
- The challenge is to come up with a simple and extensible mechanism that satisfies all of these needs.
- Encapsulating a Request
  - We could parameterize MenuItem with a *function* to call, but that's not a complete solution.
    - it doesn't address the undo/redo problem.
    - it's hard to associate state with a function.
    - functions are hard to extent, and it's hard to reuse part of them.
  - We should parameterize MenuItems with an ***object***, not a function.
- Command Class and Subclasses
  - The Command abstract class consists of a single abstract operation called "Execute".
  - MenuItem can store a Command object that encapsulates a request.
  - When a user choose a particular menu item, the MenuItem simply calls Execute on its Command object to carry out the request.

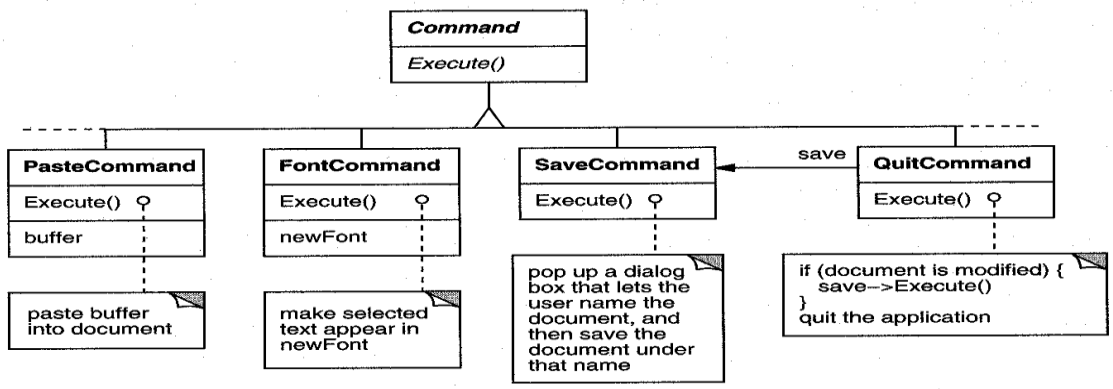


Figure 2.11: Partial Command class hierarchy

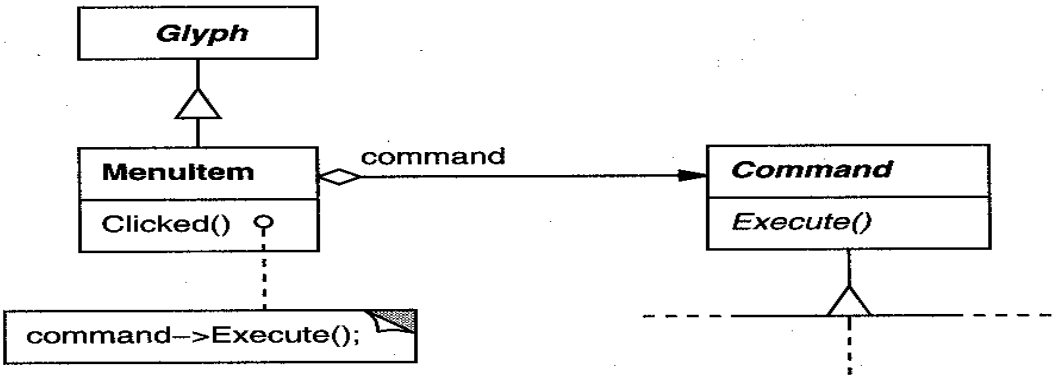
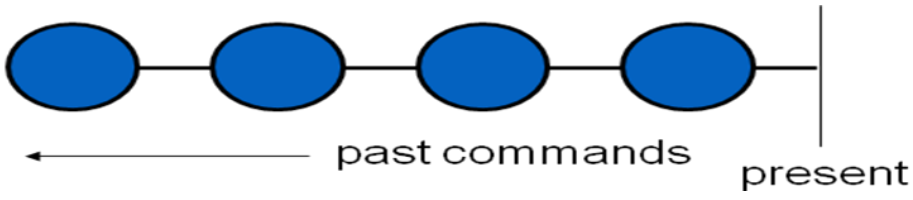


Figure 2.12: MenuItem-Command relationship

- Undoability
  - To undo and redo commands, we add an *Unexecute* operation to Command's interface.
  - A concrete Command would store the state of the Command for Unexecute .
  - Reversible operation returns a Boolean value to determine if a command is undoable.

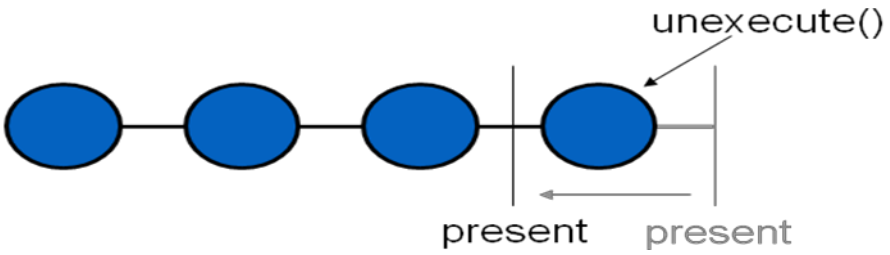
Command History

- a list of commands that have been executed.



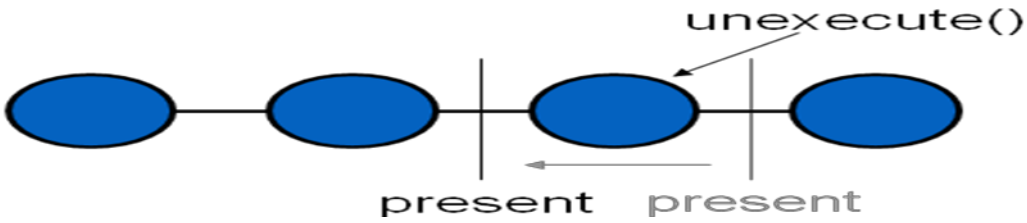
- The command history can be seen as a list of past commands commands .
- As new commands are executed they are added to the front of the history.

Undoing the Last Command:



- To undo a command, unexecute() is called on the command on the front of the list.
- The “present” position is moved past the last command.

Undoing Previous command:



Redoing the Next Command:

- To redo the command that was just undone, execute() is called on that command.
- The present pointer is moved up past that command.

The Command Pattern:

- Encapsulate a request as an object
- The Command Patterns lets you
  - parameterize clients with different requests
  - queue or log requests
  - support undoable operations
- Also Known As: Action, Transaction.