Design Pattern

Unit: I

Introduction to Design Pattern

Course Details
(B. Tech. 5th Sem)

Ibrar Ahmed
(Asst. Professor)
CSE Department

Faculty Introduction

Name	**Ibrar Ahmed**
Qualification	M. Tech. (Computer Engineering)
Designation	Assistant Professor
Department	Computer Science & Engineering
Total Experience	4 years
NIET Experience	1 years
Subject Taught	Design & Analysis of Algorithm, Data Structures, Artificial Intelligence, Soft Computing, C Programming, Web Technology, Discrete Mathematics.

Evaluation Scheme

B. TECH (CSE)
Evaluation Scheme

Session 2020-21	Third Year	SEMESTER V								ESC (3)	PCC (14)	ELC (6)	PW (1)		

Sl. No.	Subject code	Subject	Periods			Evaluation Schemes				End Semester		Total	Credit	Course Type
			L	T	P	CT	TA	TOTAL	PS	TE	PE			
1		Design Thinking -II	2	1	0	30	20	50		100		150	3	ESC
2	20CS501	Database Management System	3	1	0	30	20	50		100		150	4	PCC
3	20CS502	Web Technology	3	0	0	30	20	50		100		150	3	PCC
4	20CS503	Compiler Design	3	1	0	30	20	50		100		150	4	PCC
5		Python Web development with Django	3	0	0	30	20	50		100		150	3	ELC
6		Design Pattern	3	0	0	30	20	50		100		150	3	ELC
7	P20CS501	Database Management System Lab	0	0	2				25		25	50	1	PCC
8	P20CS502	Web Technology Lab	0	0	2				25		25	50	1	PCC
9	P20CS503	Compiler Design Lab	0	0	2				25		25	50	1	PCC
10		Internship Assessment	0	0	2				50			50	1	PW
11		Constitution of India / Essence of Indian Traditional Knowledge	2	0	0	30	20	50		50		100	0	NC
12		MOOCs for Honors degree												

Syllabus

UNIT-I: Introduction of Design Pattern

Describing Design Patterns, Design Patterns in Smalltalk MVC, The Catalogue of Design Patterns, Organizing The Cato log, How Design Patterns solve, Design Problems, How to Select a Design pattern, How to Use a Design Pattern. Principle of least knowledge.

Syllabus

UNIT-II: Creational Design Pattern

A Case Study: Designing a Document Editor

Creational Patterns: Abstract Factory, Builder , Factory Method, Prototype , Singleton Pattern,

Syllabus

UNIT-III: Structural Design Pattern

Structural Pattern Part-I, Adapter, Bridge, Composite.

Structural Pattern Part-II, Decorator, Facade, Flyweight, Proxy.

Syllabus

UNIT-IV: Behavioral Design Patterns Part: I

Behavioral Patterns Part: I, Chain of Responsibility, Command, Interpreter, Iterator Pattern.

Behavioral Patterns Part: II, Mediator, Memento, Observer, Patterns.

Syllabus

UNIT-V: Behavioral Design Patterns Part: II

Behavioral Patterns Part: III, State, Strategy, Template Method, Visitor, What to Expect from Design Patterns.

Branch Wise Application

1. Real time web analytics
2. Digital Advertising
3. E-Commerce
4. Publishing
5. Massively Multiplayer Online Games
6. Backend Services and Messaging
7. Project Management & Collaboration
8. Real time Monitoring Services
9. Live Charting and Graphing
10. Group and Private Chat

Course Objective

In this semester, the students will

Study how to shows relationships and interactions between classes or objects..

Study to speed up the development process by providing well-tested, proven development/design paradigms.

Select a specific design pattern for the solution of a given design problem.

Create a catalogue entry for a simple design pattern whose purpose and application is understood.

Course Outcomes (COs)

At the end of course, the student will be able to:

CO1 : Construct a design consisting of collection of modules.

CO2 : Exploit well known design pattern such as Factory, visitor etc.

CO3 : Distinguish between different categories of design patterns.

CO4 : Ability to common design pattern for incremental development.

CO5 : Identify appropriate design pattern for a given problem and design the software using pattern oriented architecture.

Program Outcomes (POs)

Engineering Graduates will be able to:

PO1 : Engineering Knowledge

PO2 : Problem Analysis

PO3 : Design/Development of solutions

PO4 : Conduct Investigations of complex problems

PO5 : Modern tool usage

PO6 : The engineer and society

Program Outcomes (POs)

Engineering Graduates will be able to:

PO7 : Environment and sustainability

PO8 : Ethics

PO9 : Individual and teamwork

PO10 : Communication

PO11 : Project management and finance

PO12 : Life-long learning

COs - POs Mapping

CO.K	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	2	2	2	3	3	-	-	-	-	-	-	-
CO2	3	2	3	2	3	-	-	-	-	-	-	-
CO3	3	2	3	2	3	-	-	-	-	-	-	-
CO4	3	2	3	2	3	-	-	-	-	-	-	-
CO5	3	2	3	3	3	-	-	-	-	-	-	-
AVG	2.8	2.0	2.8	2.4	3.0	-	-	-	-	-	-	-

Program Specific Outcomes(PSOs)

S. No.	Program Specific Outcomes (PSO)	PSO Description
1	PSO1	Understand to shows relationships and interactions between classes or objects of a pattern.
2	PSO2	Study to speed up the development process by providing well-tested, proven development
3	PSO3	Select a specific design pattern for the solution of a given design problem
4	PSO4	Create a catalogue entry for a simple design pattern whose purpose and application is understood.

COs - PSOs Mapping

CO.K	PSO1	PSO2	PSO3	PSO4
CO1	3	-	-	-
CO2	3	3	-	-
CO3	3	3	-	-
CO4	3	3	-	-
CO5	3	3	-	-

Program Educational Objectives (PEOs)

Program Educational Objectives (PEOs)	PEOs Description
PEOs	To have an excellent scientific and engineering breadth so as to comprehend, analyze, design and provide sustainable solutions for real-life problems using state-of-the-art technologies.
PEOs	To have a successful career in industries, to pursue higher studies or to support entrepreneurial endeavors and to face the global challenges.
PEOs	To have an effective communication skills, professional attitude, ethical values and a desire to learn specific knowledge in emerging trends, technologies for research, innovation and product development and contribution to society.
PEOs	To have life-long learning for up-skilling and re-skilling for successful professional career as engineer, scientist, entrepreneur and bureaucrat for betterment of society.

Result Analysis(Department Result & Subject Result & Individual result

Name of the faculty	Subject code	Result % of clear passed
Mr. Sanjay Nayak		

Pattern of Online External Exam Question Paper (100 marks)

Printed page: .... **Subject Code: .....**

Roll No: | | | | | | | | | | | | |

NOIDA INSTITUTE OF ENGINEERING AND TECHNOLOGY, GREATER NOIDA

(An Autonomous Institute Affiliated to AKTU, Lucknow)

B.Tech./MBA/MCA/M.Tech (Integrated)

(SEM:....THEORY EXAMINATION(2020-2021)

Subject .....

Time: 2 Hours **Max. Marks: 100**

Pattern of Online External Exam Question Paper (100 marks)

		SECTION – A	**[30]**	**CO**
1.	**Attempt all parts- (MCQ, True False)Three Question From Each Unit**		**[15×2=30]**	
		UNIT-1		
	1-a.	**Question-**	**(2)**	
	1-b.	**Question-**	**(2)**	
	1-c.	**Question-**	**(2)**	
		UNIT-2		
	1-d.	**Question-**	**(2)**	
	1-e.	**Question-**	**(2)**	
	1-f.	**Question-**	**(2)**	
		UNIT-3		
	1-g.	**Question-**	**(2)**	
	1-h.	**Question-**	**(2)**	
	1-i.	**Question-**	**(2)**	
		UNIT-4		
	1-j.	**Question-**	**(2)**	
	1-k.	**Question-**	**(2)**	
	1-l.	**Question-**	**(2)**	
		UNIT-5		
	1-m.	**Question-**	**(2)**	
	1-n.	**Question-**	**(2)**	
	1-o.	**Question-**	**(2)**	

Pattern of Online External Exam Question Paper (100 marks)

		SECTION – B	**[20×2=40]**	**CO**
2.	**Attempt all Four parts. Fill in The Blanks, Match the pairs (From the Data Given in Glossary)) Question from Unseen passage - Four Question From Unit-I**		**[4×2=08]**	**CO**
	Glossary- (Required words to be written)			
	2-a.	**Question-**	**(2)**	
	2-b.	**Question-**	**(2)**	
	2-c.	**Question-**	**(2)**	
	2-d.	**Question-**	**(2)**	
3.	**Attempt all Four parts. Fill in The Blanks, Match the pairs (From the Data Given in Glossary) Question from Unseen passage - Four Question From Unit-II**		**[4×2=08]**	**CO**
		Glossary- (Required words to be written)		
	3-a.	**Question-**	**(2)**	
	3-b.	**Question-**	**(2)**	
	3-c.	**Question-**	**(2)**	
	3-d.	**Question-**	**(2)**	

Pattern of Online External Exam Question Paper (100 marks)

4.	**Attempt all Four parts. Fill in The Blanks, Match the pairs (From the Data Given in Glossary) Question from Unseen passage - Four Question From Unit-III**		**[4×2=08]**	**CO**
		Glossary- (Required words to be written)		
	4-a.	**Question-**	**(2)**	
	4-b.	**Question-**	**(2)**	
	4-c.	**Question-**	**(2)**	
	4-d.	**Question-**	**(2)**	
5.	**Attempt all Four parts. Fill in The Blanks, Match the pairs (From the Data Given in Glossary) Question from Unseen passage - Four Question From Unit-IV**		**[4×2=08]**	**CO**
		Glossary- (Required words to be written)		
	5-a.	**Question-**	**(2)**	
	5-b.	**Question-**	**(2)**	
	5-c.	**Question-**	**(2)**	
	5-d.	**Question-**	**(2)**	
6.	**Attempt all Four parts. Fill in The Blanks, Match the pairs (From the Data Given in Glossary) Question from Unseen passage - Four Question From Unit-V**		**[4×2=08]**	**CO**
		Glossary- (Required words to be written)		
	6-a.	**Question-**	**(2)**	
	6-b.	**Question-**	**(2)**	
	6-c.	**Question-**	**(2)**	
	6-d.	**Question-**	**(2)**	

Pattern of Online External Exam Question Paper (100 marks)

		SECTION – C		
7	**Answer any10 out 15 of the following, Subjective Type Question, Three Question from Each Unit**		**[10×3=30]**	**CO**
		UNIT-1		
	7-a.	**-Question-**	**(3)**	
	7-b.	**-Question-**	**(3)**	
	7-c.	**-Question-**	**(3)**	
		UNIT-2		
	7-d.	**-Question-**	**(3)**	
	7-e.	**-Question-**	**(3)**	
	7-f.	**-Question-**	**(3)**	
		UNIT-3		
	7-g.	**-Question-**	**(3)**	
	7-h.	**-Question-**	**(3)**	
	7-i.	**-Question-**	**(3)**	
		UNIT-4		
	7-j.	**-Question-**	**(3)**	
	7-k.	**-Question-**	**(3)**	
	7-l.	**-Question-**	**(3)**	
		UNIT-5		
	7-m	**-Question-**	**(3)**	
	7-n.	**-Question-**	**(3)**	
	7-o.	**-Question-**	**(3)**	

Prerequisite / Recap

- Student should have knowledge of object oriented analysis and design.
- Knowledge of Data structure and algorithm.
- knowledge of Programing language such as C/C++ etc.
- Good problem solving Skill .

Brief Introduction about the Subject with videos

YouTube /other Video Links

- https://youtu.be/rI4kdGLaUiQ?list=PL6n9fhu94yhUbctIoxoVTrklN3LMwTCmd
- https://youtu.be/v9ejT8FO-7I?list=PLrhzvIcii6GNjpARdnO4ueTUAVR9eMBpc
- https://youtu.be/VGLjQuEQgkI?list=PLt4nG7RVVk1h9lxOYSOGI9pcP3I5oblbx

Unit II Content

- A Case Study: Designing a Document Editor.
- Creational Patterns.
- Abstract Factory.
- Builder Factory Method,
- Prototype Patterns.
- Singleton Patterns.

Unit II Objective

In Unit II, the students will be able to find

- What are Creational Patterns and their types in detail.
- Learn the concept of Abstract Factory.
- How Builder Factory Method works.
- How and when we use Prototype Patterns.
- Simplest and easiest pattern is Singleton Patterns how .

Topic Objective

Topic : A Case Study - Designing a Document Editor

- ✓ This chapter presents a case study in the design of a “What-You-See-Is-What-You-Get” (or “WYSIWYG”) document editor called Lexi. We’ll see how design patterns capture solutions to design problems in Lexi and applications like it. By the end of this chapter you will have gained experience with eight patterns, learning them by example.

A Case Study - Designing a Document Editor

- A WYSIWYG representation of the document occupies the large rectangular area in the center.
- The document can mix text and graphics freely in a variety of formatting styles.
- Surrounding the document are the usual pull-down menus and scroll bars, and a collection of page icons for jumping to a particular page in the document.

A Case Study - Designing a Document Editor

Design Problems

Seven problems in Lexis's design:-

1. **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.
- The way we organize those impacts design information.

2. **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?

A Case Study - Designing a Document Editor

Design Problems

3. Embellishing the user interface:

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

4. Supporting multiple look-and-feel standards:

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

5. Supporting multiple window systems:

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

A Case Study - Designing a Document Editor

Design Problems

6. 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.

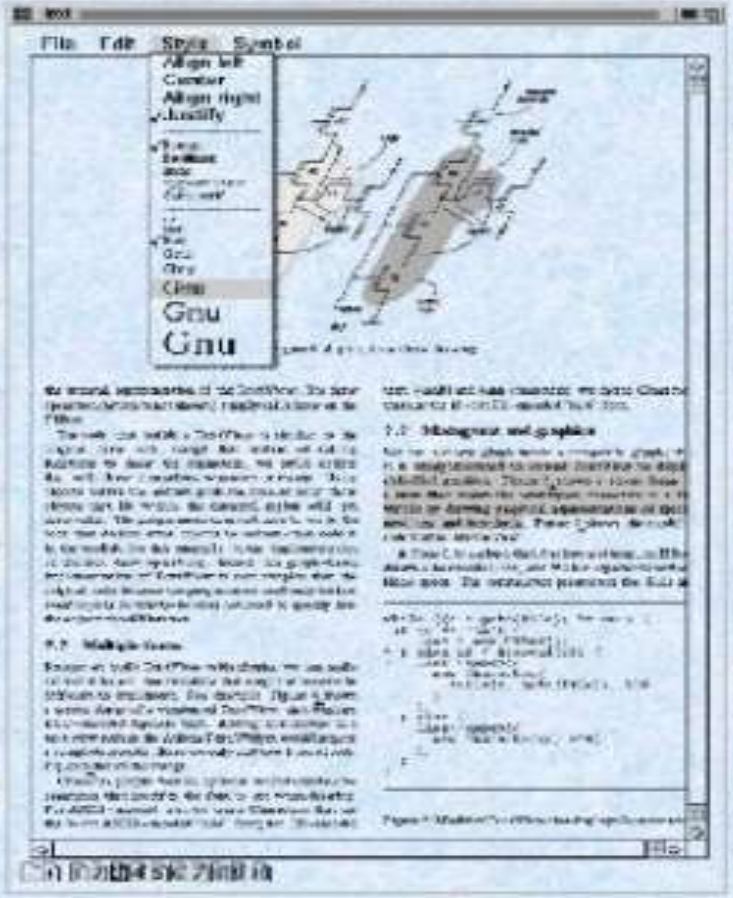
7. Spelling checking and Hyphenation (automated process of breaking words):

- 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?

A Case Study - Designing a Document Editor

L1

Part II: Application: Document Editor (Lexi)



7 Design Problems

1. Document structure
2. Formatting
3. Embellishment
4. Multiple look & feels
5. Multiple window systems
6. User operations
7. Spelling checking & hyphenation

A Case Study - Designing a Document Editor

Document Structure

- 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

A Case Study - Designing a Document Editor

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

A Case Study - Designing a Document Editor

Document Structure

- An arrangement of graphical elements (characters, lines, polygon, other shapes) and structural elements (lines, columns, figures, tables, and other substructures).
- Lexi's user interface should let users manipulate these substructures directly. That helps make the interface simple.
- For example, a user should be able to treat a diagram as a unit rather than as a collection of individual graphical primitives.

Goals

- The internal representation should support:
- Maintaining the document's physical structure.
- Generating and presenting the document's visual aspects.
- Mapping positions on the display to elements in the internal representations (drawing, hit detection, alignment, etc).

A Case Study - Designing a Document Editor

Document Structure (cont.)

- 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

A Case Study - Designing a Document Editor

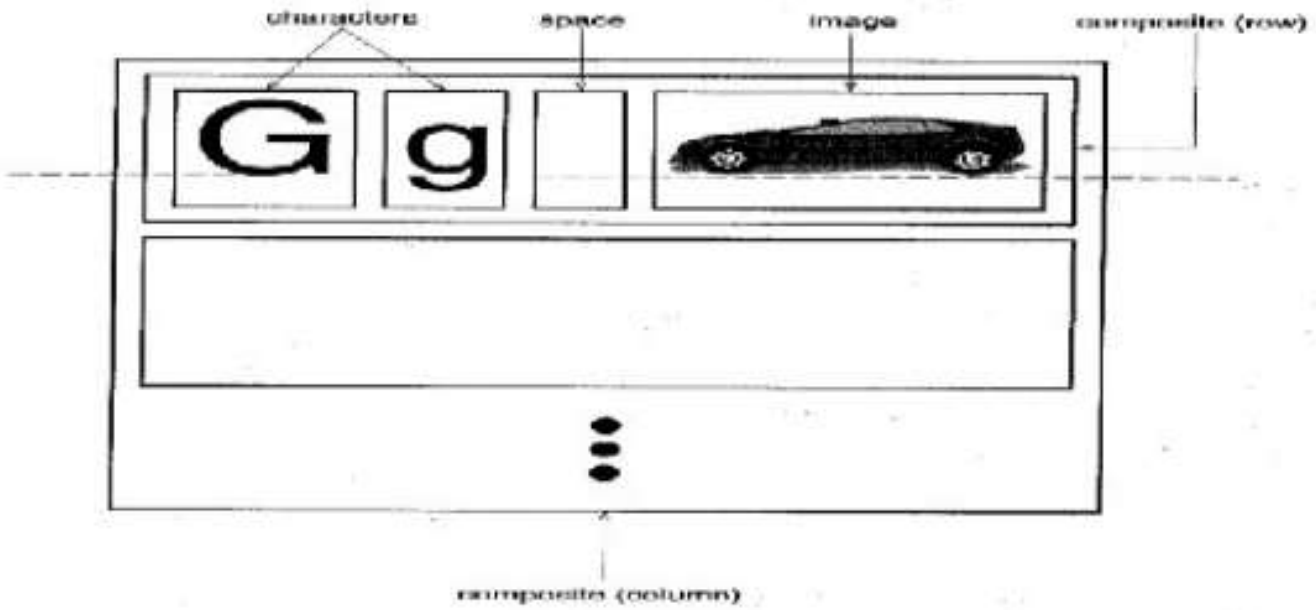


Figure 2.2: Recursive composition of text and graphics

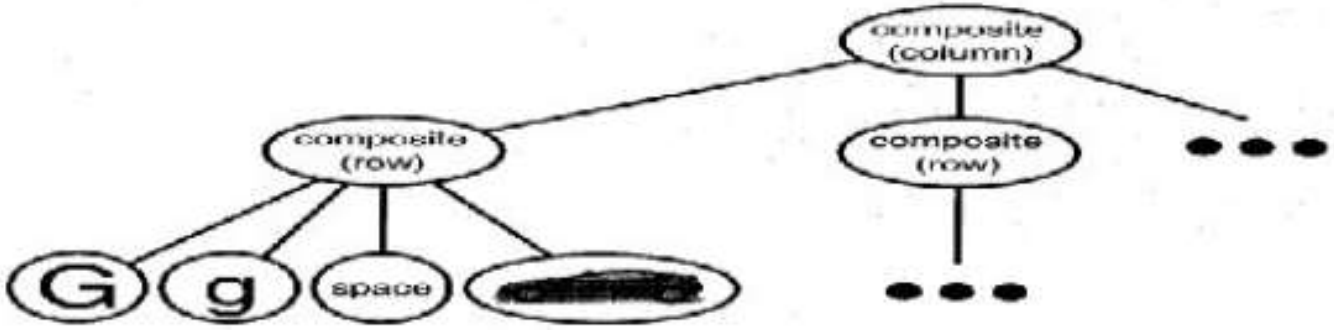


Figure 2.3: Object structure for recursive composition of text and graphics

Document Structure (cont.)

- 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, and their children and parent
- Composite Pattern
  - captures the essence of recursive composition in object-oriented terms

A Case Study - Designing a Document Editor

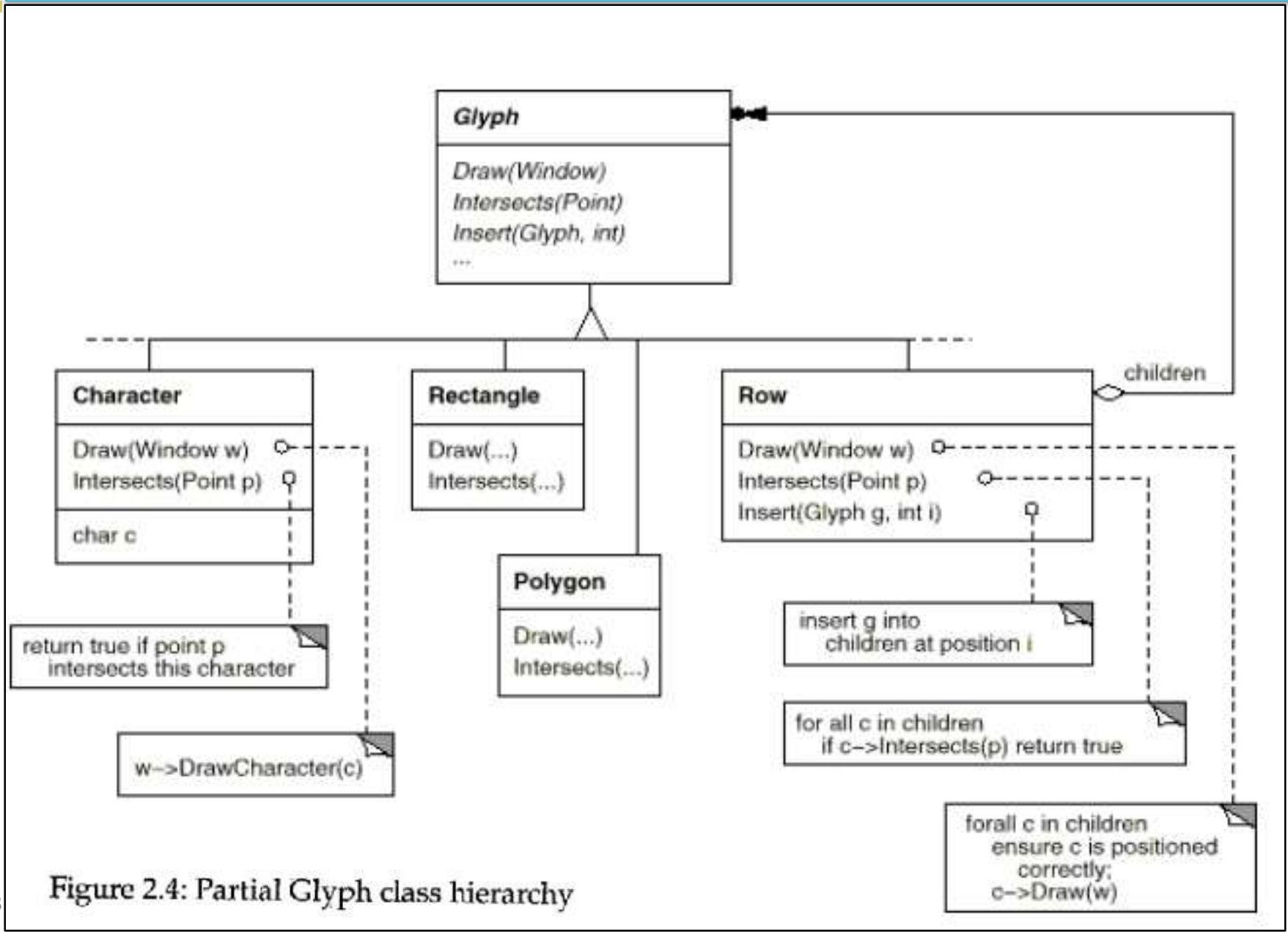


Figure 2.4: Partial Glyph class hierarchy

A Case Study - Designing a Document Editor

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

A Case Study - Designing a Document Editor

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

Formatting (cont.)

- 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

A Case Study - Designing a Document Editor

Formatting (cont.)

- 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

A Case Study - Designing a Document Editor

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

Table 2.2: Basic compositor interface

Formatting (cont.)

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

A Case Study - Designing a Document Editor

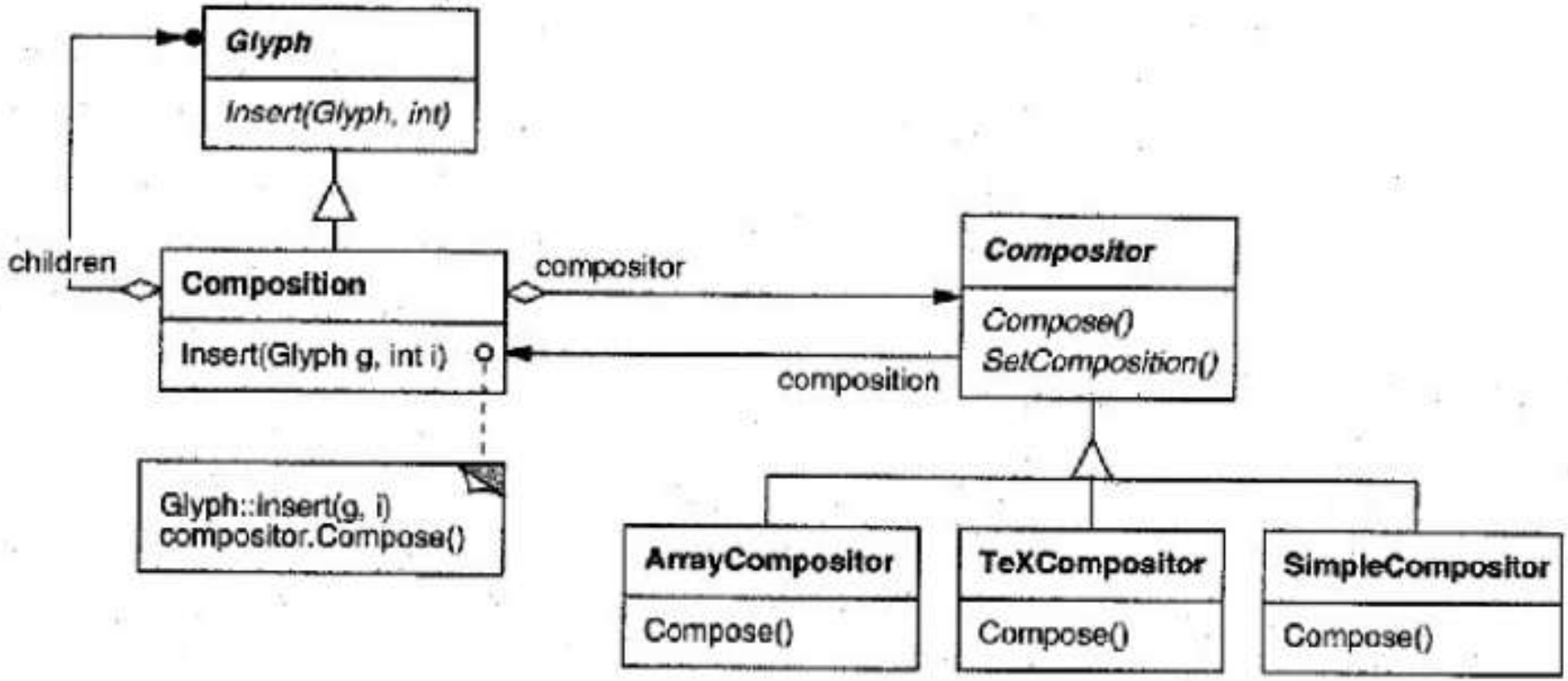


Figure 2.5: Composition and Compositor class relationships

A Case Study - Designing a Document Editor

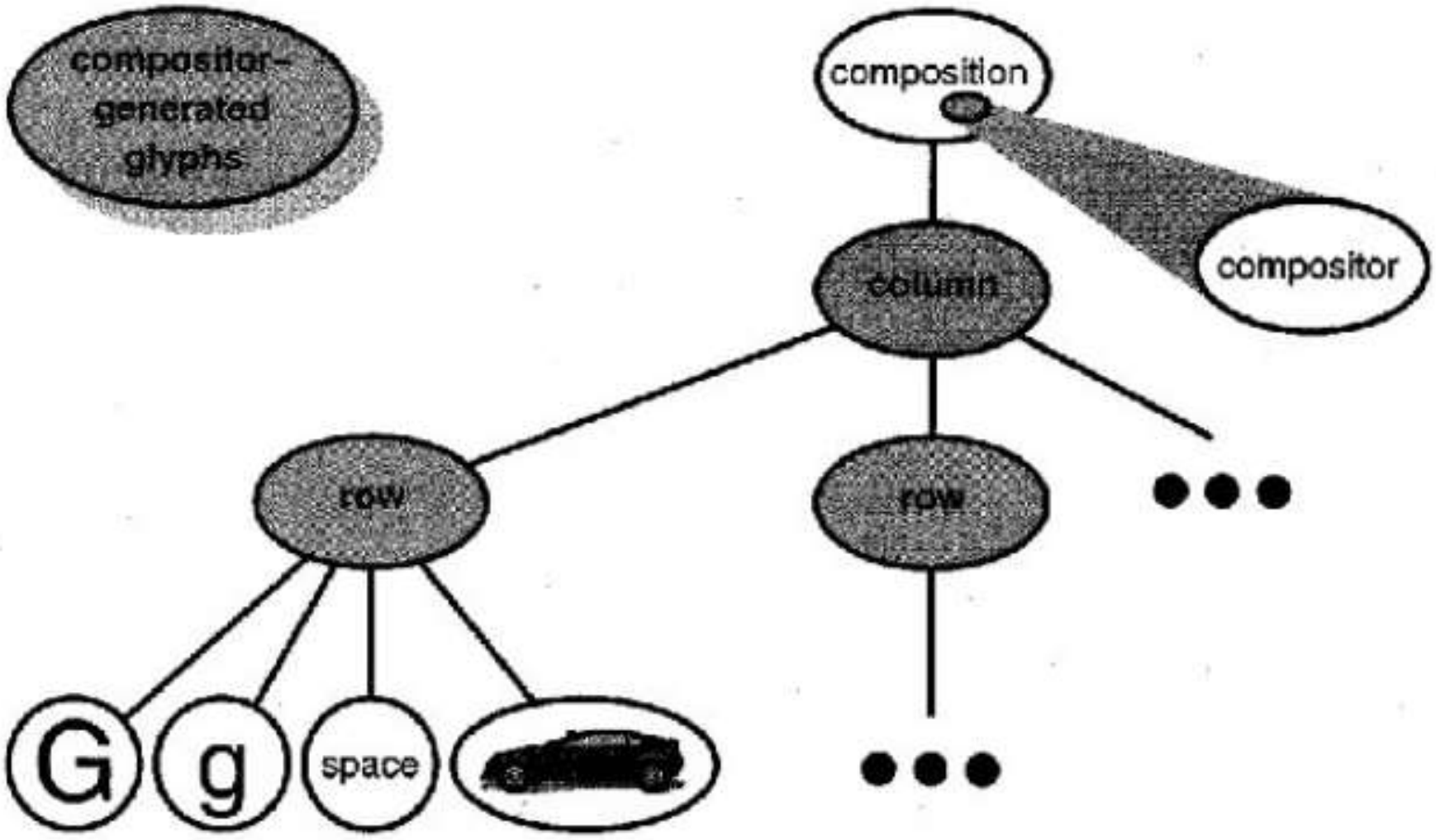


Figure 2.6: Object structure reflecting compositor-directed linebreaking

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

Embellishing the User Interface (cont.)

- Transparent enclosure (cont.)
  - object composition (cont.)
    - Border and Scroller should be a subclass of Glyph
  - two notions
    - single-child (single-component) composition
    - compatible interfaces

Embellishing the User Interface (cont.)

- Monoglyph
  - We can apply the concept of transparent enclosure to all glyphs that embellish other glyphs
  - the class, Monoglyph
- Decorator Pattern
  - captures class and object relationships that support embellishment by transparent enclosure

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

A Case Study - Designing a Document Editor

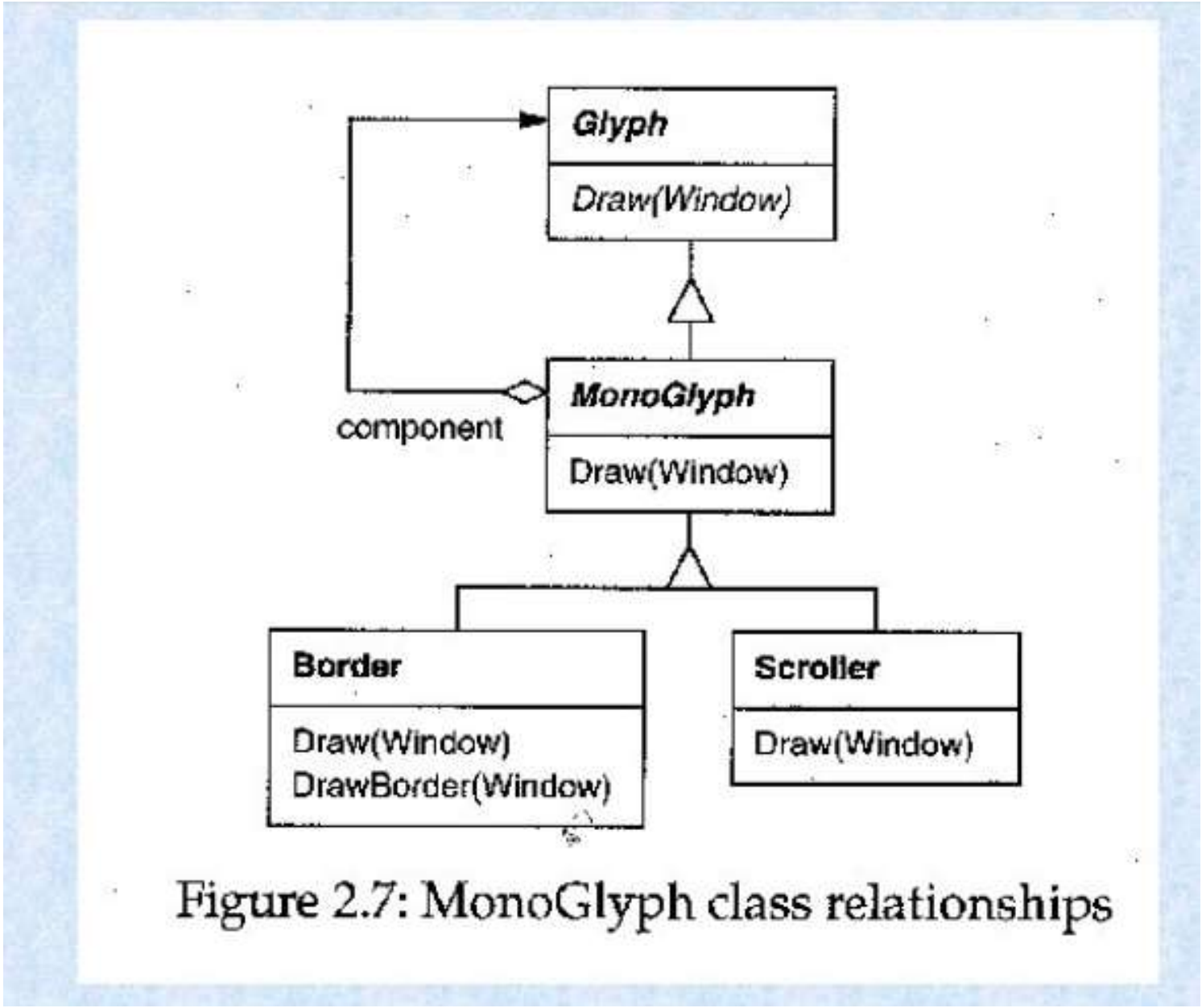


Figure 2.7: MonoGlyph class relationships

A Case Study - Designing a Document Editor

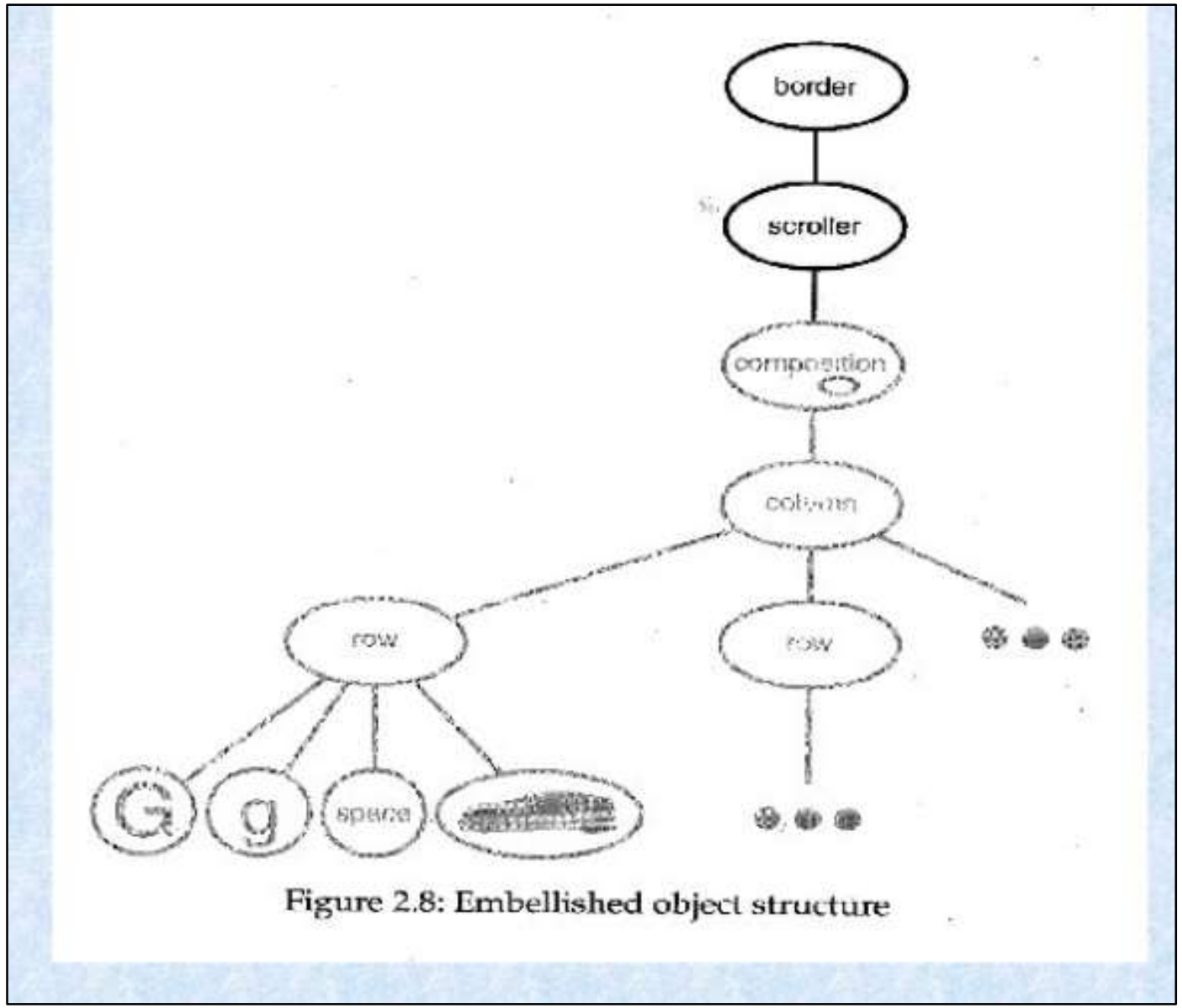


Figure 2.8: Embellished object structure

A Case Study - Designing a Document Editor

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)

A Case Study - Designing a Document Editor

Supporting Multiple Look-and-Feel Standards (cont.)

- 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*

Supporting Multiple Look-and-Feel Standards (cont.)

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

A Case Study - Designing a Document Editor

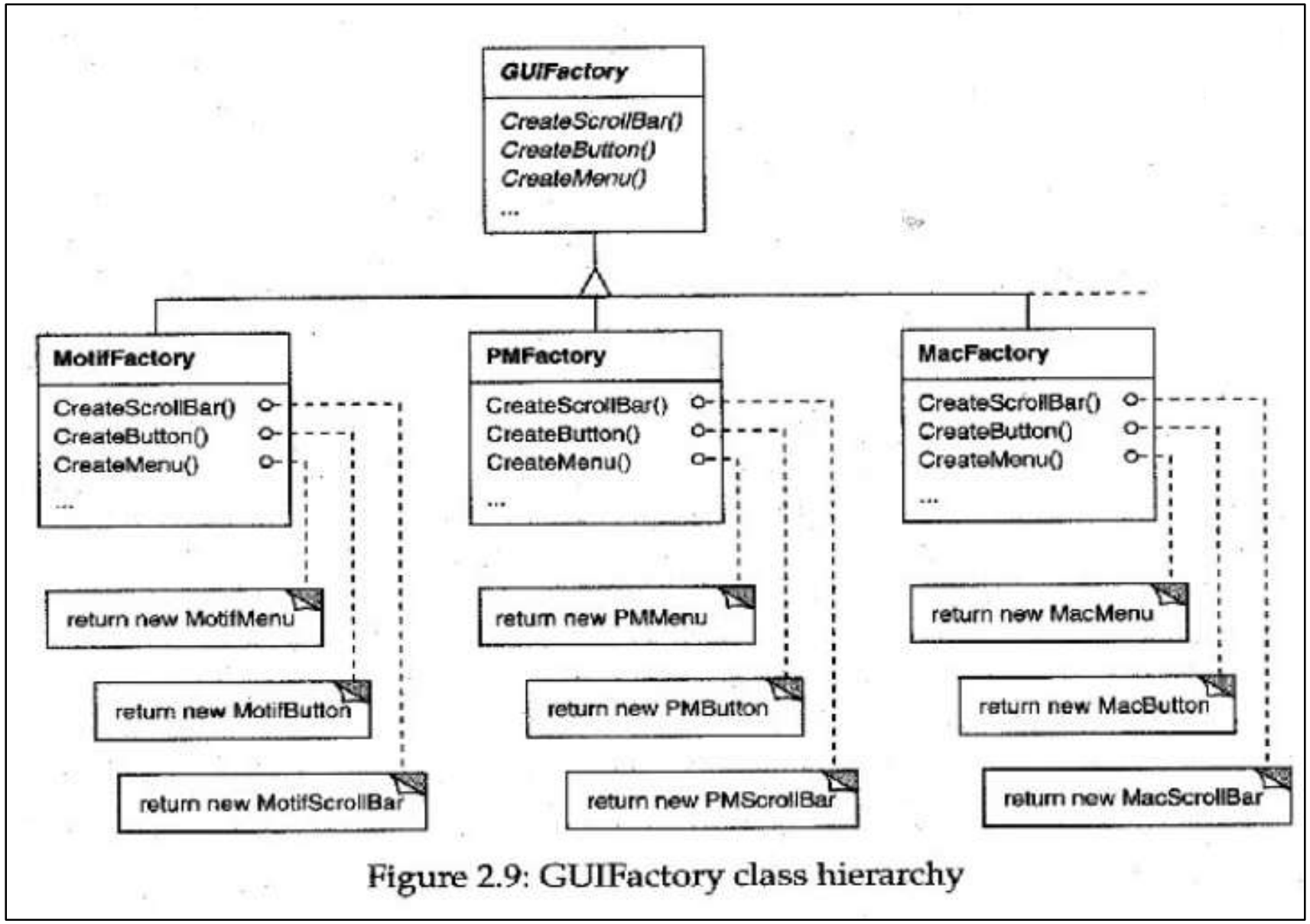


Figure 2.9: GUIFactory class hierarchy

A Case Study - Designing a Document Editor

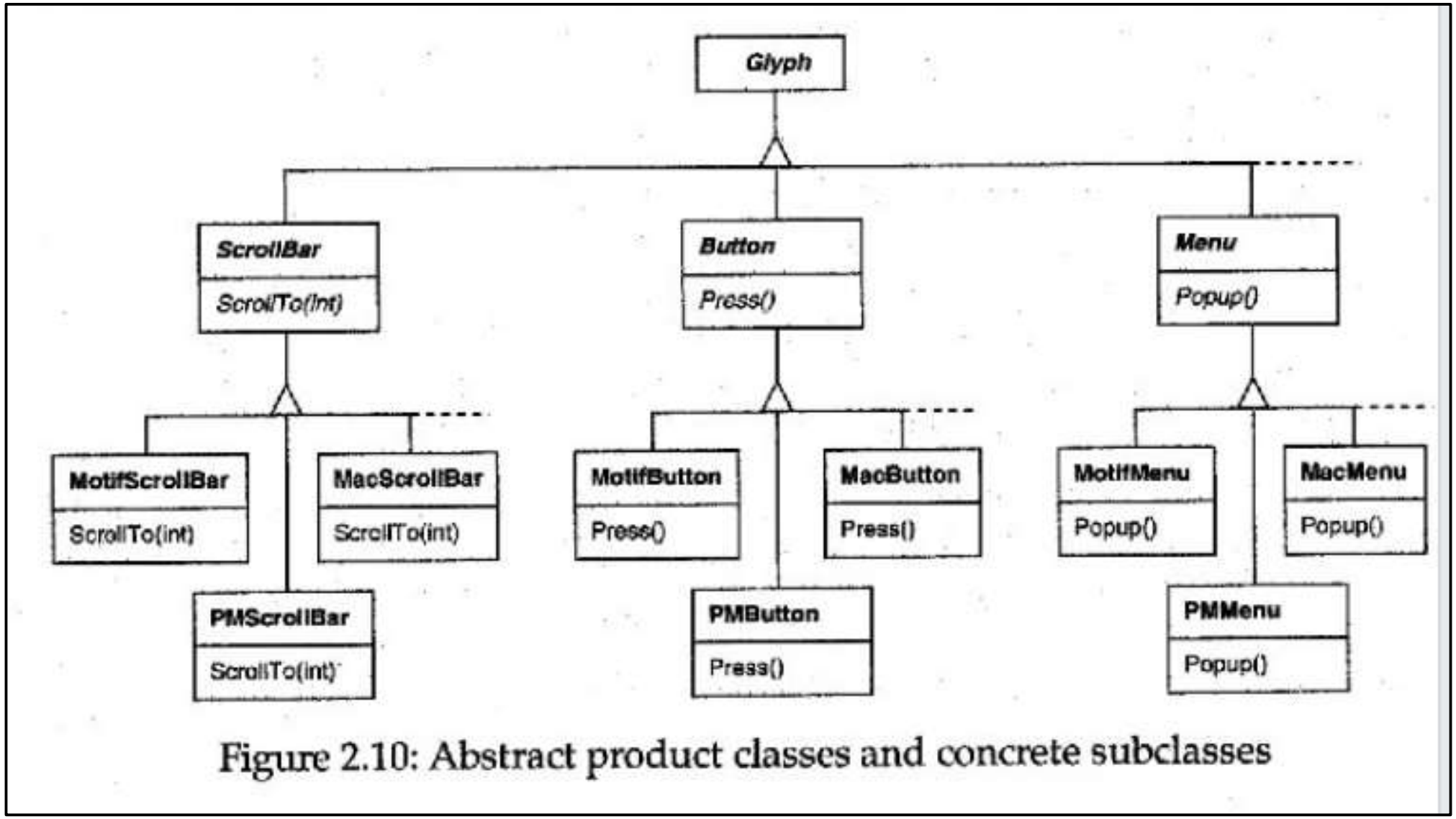


Figure 2.10: Abstract product classes and concrete subclasses

Supporting Multiple Window Systems

L5

- 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

A Case Study - Designing a Document Editor

Supporting Multiple Window Systems (cont.)

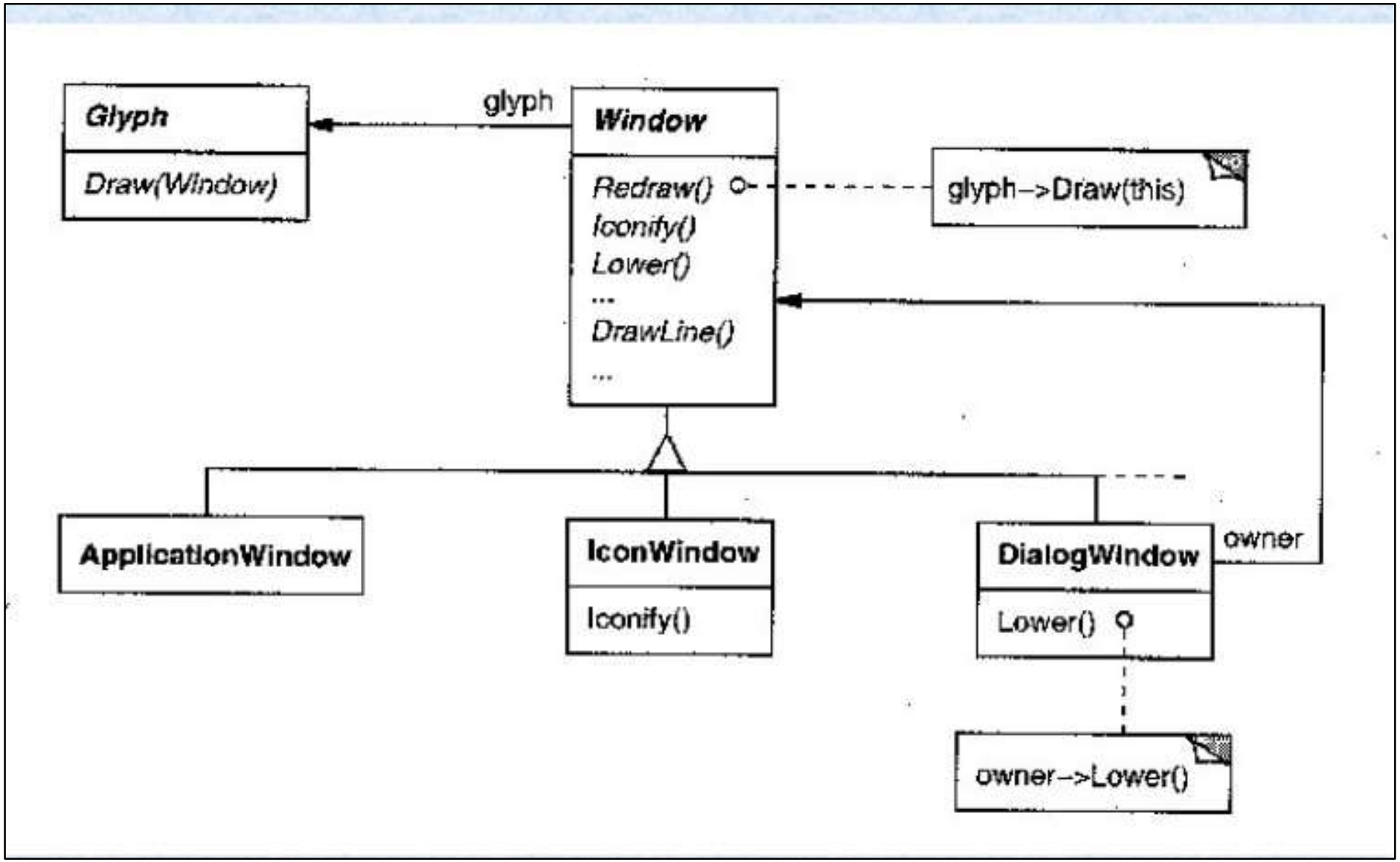
- Encapsulating Implementation Dependencies
  - The Window class interface encapsulates the things windows tend to do across window systems
  - The Window class is an abstract class
  - Where does the implementation live?
- Window and WindowImp
- Bridge Pattern
  - to allow separate class hierarchies to work together even as they evolve independently

A Case Study - Designing a Document Editor

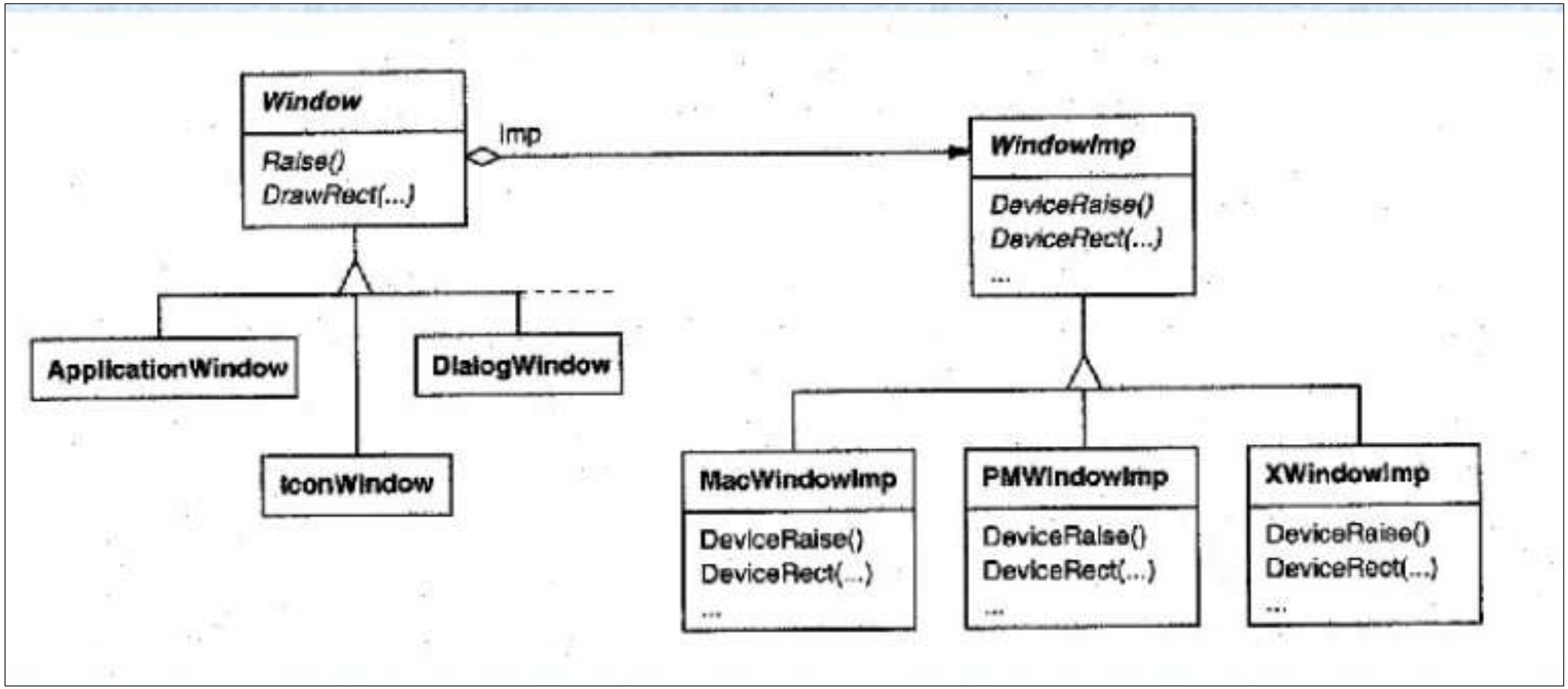
Responsibility	Operations
window management	virtual void Redraw() virtual void Raise() virtual void Lower() virtual void Iconify() virtual void Deiconify() ...
graphics	virtual void DrawLine(...) virtual void DrawRect(...) virtual void DrawPolygon(...) virtual void DrawText(...) ...

Table 2.3: Window class interface

A Case Study - Designing a Document Editor



A Case Study - Designing a Document Editor



A Case Study - Designing a Document Editor

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

A Case Study - Designing a Document Editor

User Operations (cont.)

- 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

A Case Study - Designing a Document Editor

User Operations (cont.)

- 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

A Case Study - Designing a Document Editor

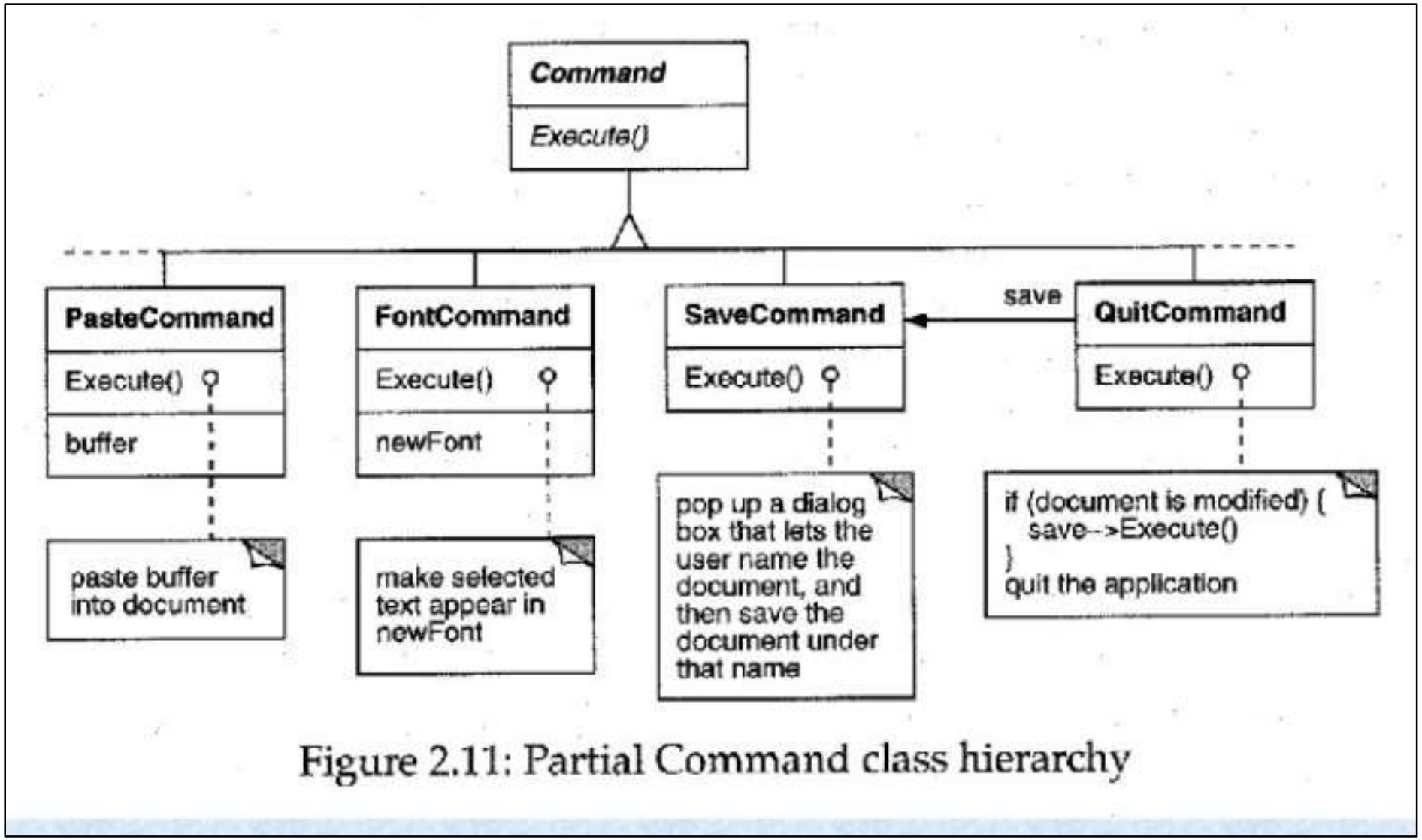


Figure 2.11: Partial Command class hierarchy

A Case Study - Designing a Document Editor

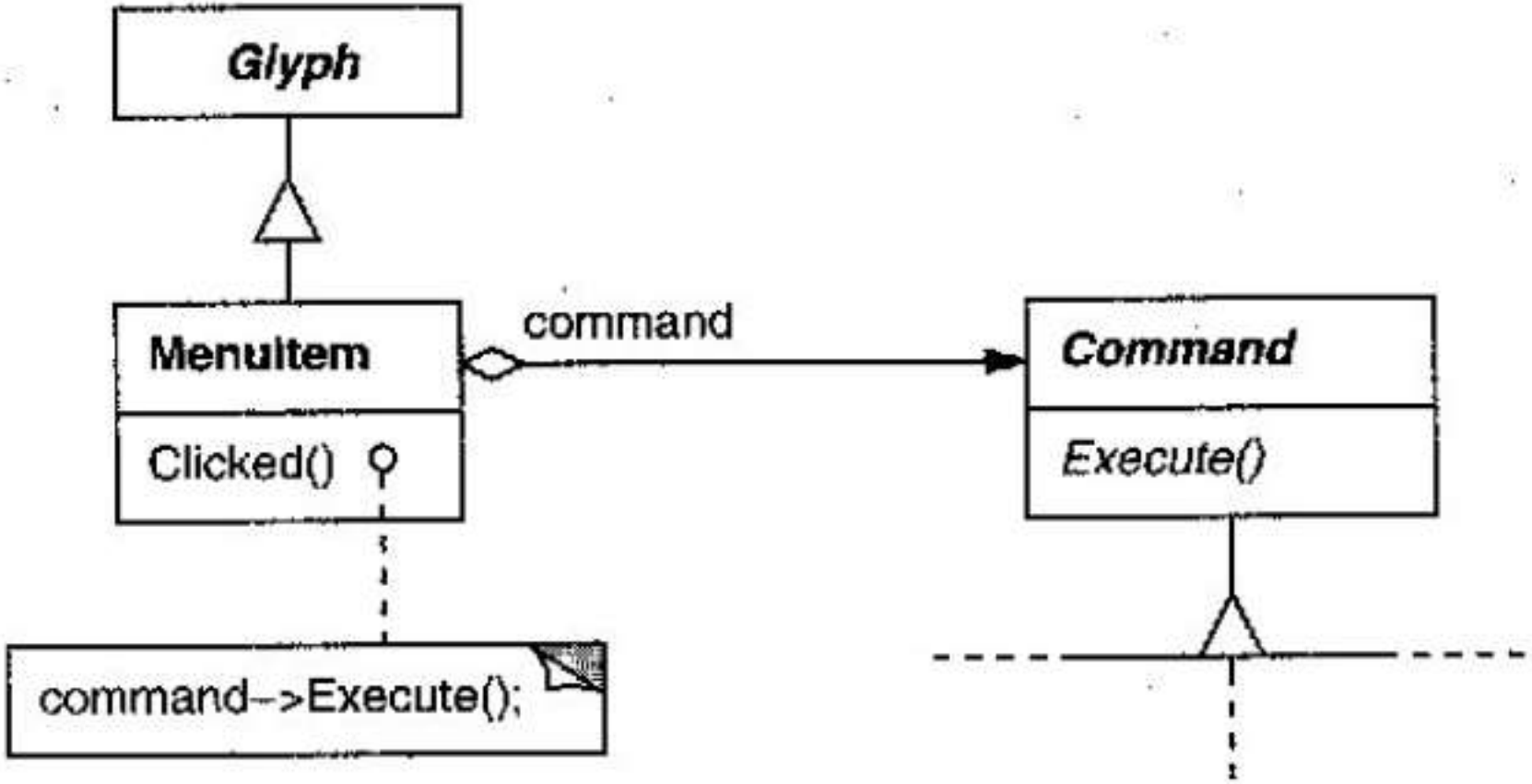


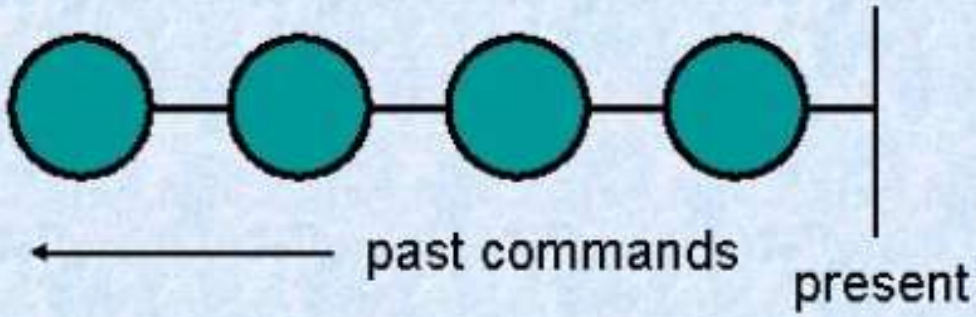
Figure 2.12: MenuItem-Command relationship

User Operations (cont.)

- 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

A Case Study - Designing a Document Editor

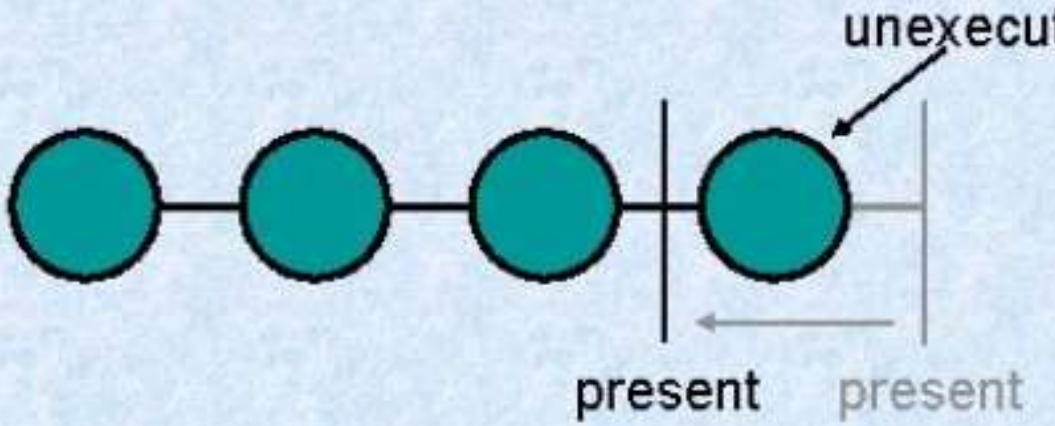
Implementing a Command History



- 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

A Case Study - Designing a Document Editor

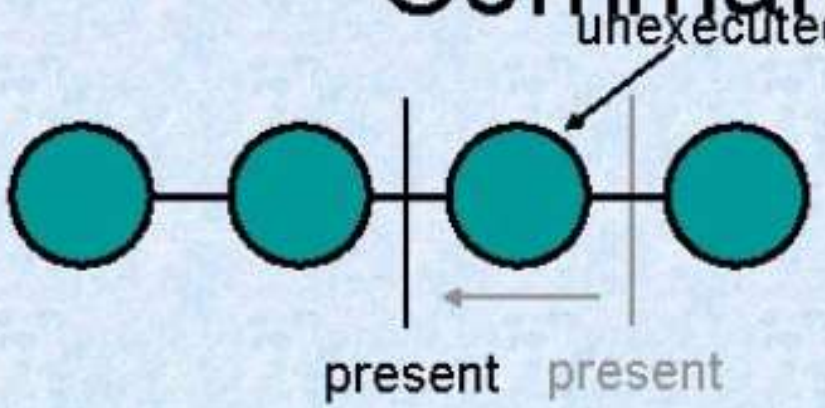
Undoing the Last Command



- 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

A Case Study - Designing a Document Editor

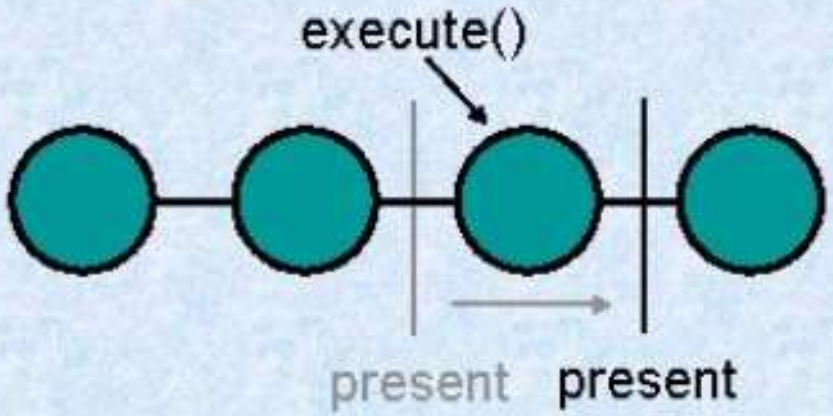
Undoing the Previous Command



- To undo the previous command, unexecute() is called on the next command in the history
- The present pointer is moved to point before that command

A Case Study - Designing a Document Editor

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

A Case Study - Designing a Document Editor

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
- Covered on pg. 233 in the book

A Case Study - Designing a Document Editor

Spelling Checking & Hyphenation

Goals:

- analyze text for spelling errors
- introduce potential hyphenation sites

Constraints/forces:

- support multiple algorithms
- don't tightly couple algorithms with document structure

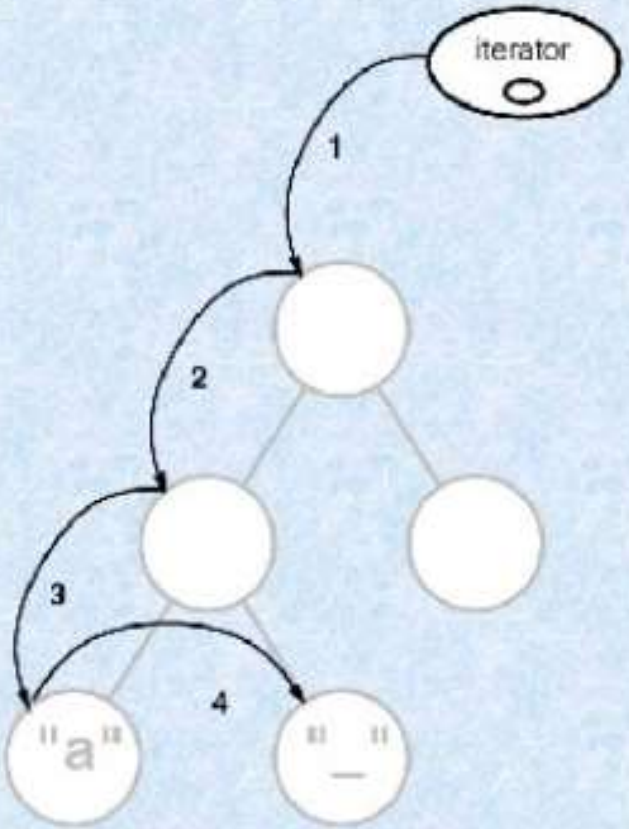
A Case Study - Designing a Document Editor

Spelling Checking & Hyphenation (cont'd)

Solution: Encapsulate Traversal

Iterator

- encapsulates a traversal algorithm without exposing representation details to callers
- uses Glyph's child enumeration operation
- This is an example of a "preorder iterator"



A Case Study - Designing a Document Editor

Spelling Checking & Hyphenation (cont'd)

TERATOR object behavioral

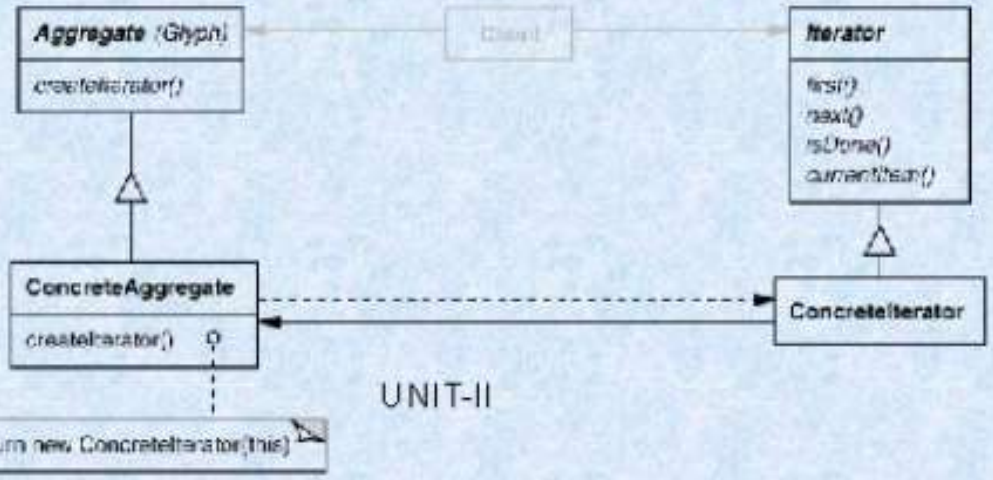
Intent

access elements of a container without exposing its representation

Applicability

- require multiple traversal algorithms over a container
- require a uniform traversal interface over different containers
- when container classes & traversal algorithm must vary independently

Structure



UNIT-II 49

A Case Study - Designing a Document Editor

Spelling Checking & Hyphenation (cont'd)

TERATOR (cont'd) object behavioral

Iterators are used heavily in the C++ Standard Template Library (STL)

```
int main (int argc, char *argv[]) {
   vector<string> args;
   for (int i = 0; i < argc; i++)
       args.push_back (string (argv[i]));
   for (vector<string>::iterator i (args.begin ());
        i != args.end ();
        i++)
     cout << *i;
   cout << endl;
   return 0;
}
```

The same iterator pattern can be applied to any STL container!

```
for (Glyph::iterator  i = glyphs.begin ();
     i != glyphs.end ();
     i++)
   ...
```

A Case Study - Designing a Document Editor

Spelling Checking & Hyphenation (cont'd)

TERATOR (cont'd) object behavioral

Consequences

+ flexibility: aggregate & traversal are independent
+ multiple iterators & multiple traversal algorithms
– additional communication overhead between iterator & aggregate

Implementation

– internal versus external iterators
– violating the object structure's encapsulation
– robust iterators
– synchronization overhead in multi-threaded programs
– batching in distributed & concurrent programs

Known Uses

– C++ STL iterators
– JDK Enumeration, Iterator
– Unidraw Iterator

A Case Study - Designing a Document Editor

Spelling Checking & Hyphenation (cont'd) L7

Visitor

- defines action(s) at each step of traversal
- avoids wiring action(s) into Glyphs
- iterator calls glyph's `accept(Visitor)` at each node
- `accept()` calls back on visitor (a form of "static polymorphism" based on method overloading by type)

```
void Character::accept (Visitor &v) { v.visit (*this); }

class Visitor {
public:
    virtual void visit (Character &);
    virtual void visit (Rectangle &);
    virtual void visit (Row &);
    // etc. for all relevant Glyph subclasses
};
```

UNIT-II 52

A Case Study - Designing a Document Editor

Spelling Checking & Hyphenation (cont'd)

SpellingCheckerVisitor

- gets character code from each character glyph
  Can define `getCharCode()` operation just on `Character()` class
- checks words accumulated from character glyphs
- combine with **PreorderIterator**

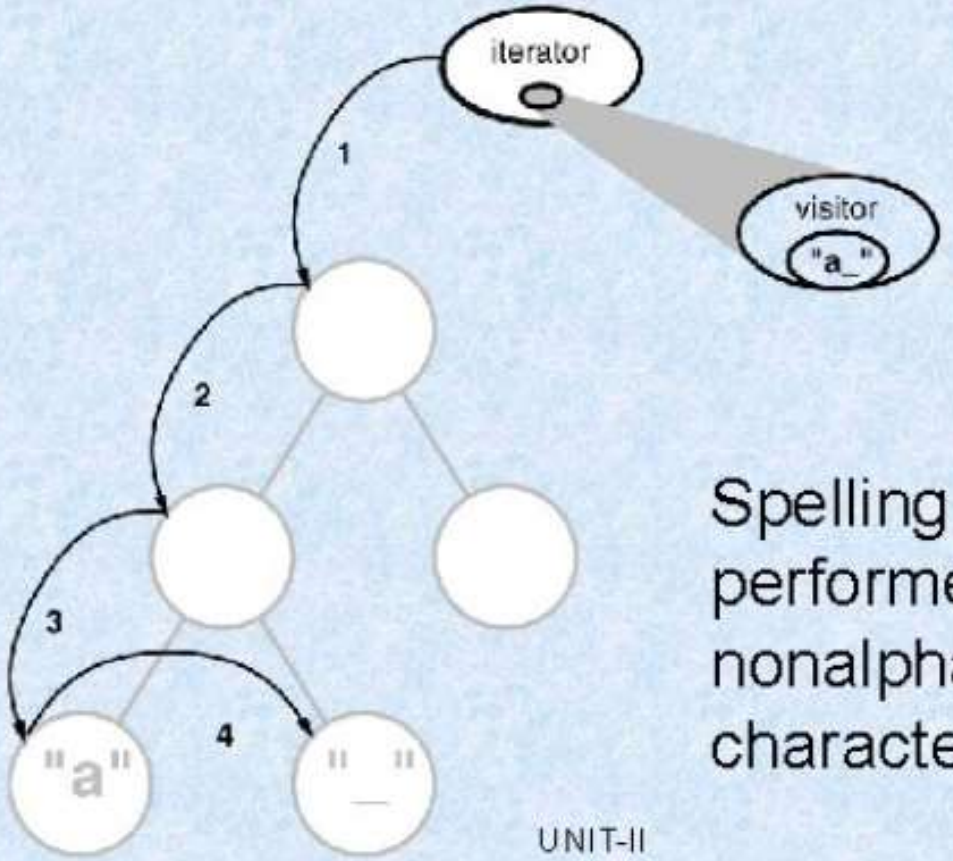
```
class SpellCheckerVisitor : public Visitor {
public:
    virtual void visit (Character &);
    virtual void visit (Rectangle &);
    virtual void visit (Row &);
    // etc. for all relevant Glyph subclasses
Private:
  std::string accumulator_;
};
```

UNIT-II 53

A Case Study - Designing a Document Editor

Spelling Checking & Hyphenation (cont'd)

Accumulating Words



Spelling check performed when a nonalphabetic character it reached

UNIT-II

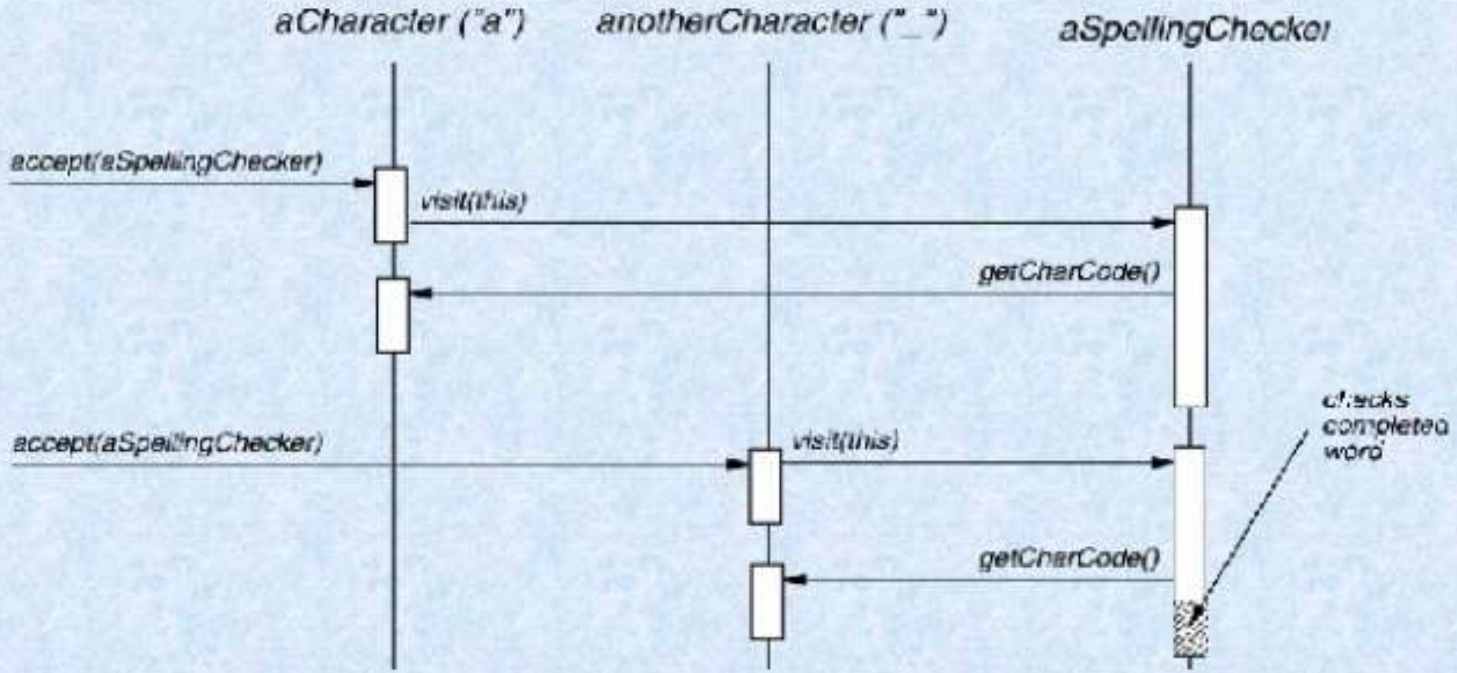
54

A Case Study - Designing a Document Editor

Spelling Checking & Hyphenation (cont'd)

Interaction Diagram

- The iterator controls the order in which accept() is called on each glyph in the composition
- accept() then "visits" the glyph to perform the desired action
- The Visitor can be sub-classed to implement various desired actions



A Case Study - Designing a Document Editor

Spelling Checking & Hyphenation (cont'd)

HyphenationVisitor

- gets character code from each character glyph
- examines words accumulated from character glyphs
- at potential hyphenation point, inserts a...

```
class HyphenationVisitor : public Visitor {
public:
    void visit (Character &);
    void visit (Rectangle &);
    void visit (Row &);
    // etc. for all relevant Glyph subclasses
};
```

A Case Study - Designing a Document Editor

Spelling Checking & Hyphenation (cont'd)

Discretionary Glyph

- looks like a hyphen when at end of a line
- has no appearance otherwise
- Compositor considers its presence when determining linebreaks

aluminum alloy

or

aluminum al-
loy

A Case Study - Designing a Document Editor

Spelling Checking & Hyphenation (cont'd)

VISITOR — object behavioral

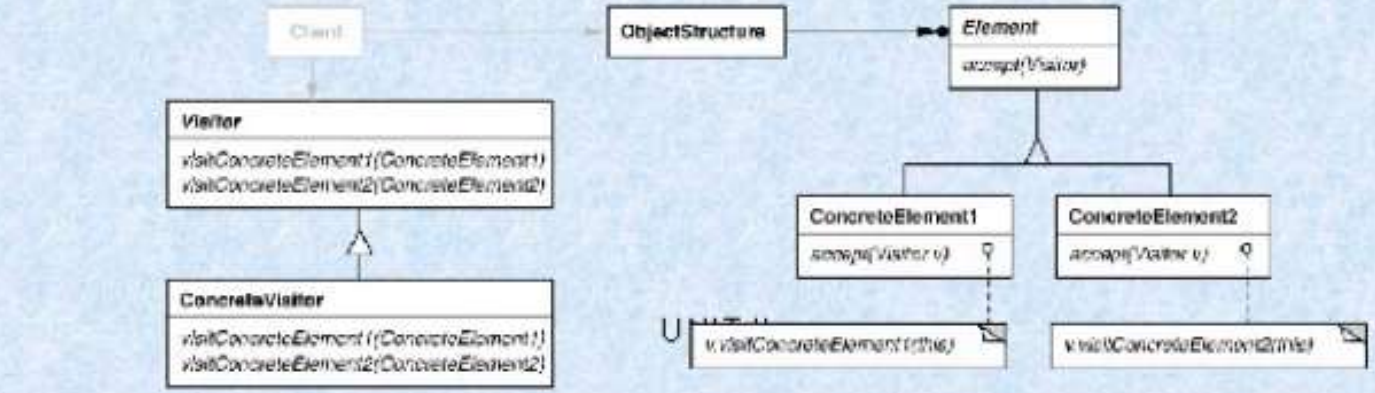
Intent

centralize operations on an object structure so that they can vary independently but still behave polymorphically

Applicability

- when classes define many unrelated operations
- class relationships of objects in the structure rarely change, but the operations on them change often
- algorithms keep state that's updated during traversal

Structure



A Case Study - Designing a Document Editor

Spelling Checking & Hyphenation (cont'd)

VISITOR (cont'd) object behavioral

```
SpellCheckerVisitor spell_check_visitor;

for (Glyph::iterator i = glyphs.begin ();
     i != glyphs.end ();
     i++) {
   (*i)->accept (spell_check_visitor);
}

HyphenationVisitor hyphenation_visitor;

for (Glyph::iterator i = glyphs.begin ();
     i != glyphs.end ();
     i++) {
   (*i)->accept (hyphenation_visitor);
}
```

A Case Study - Designing a Document Editor

Spelling Checking & Hyphenation (cont'd)

VISITOR (cont'd) object behavioral

Consequences

- \+ flexibility: visitor & object structure are independent
- \+ localized functionality
- – circular dependency between Visitor & Element interfaces
- – Visitor brittle to new ConcreteElement classes

Implementation

- – double dispatch
- – general interface to elements of object structure

Known Uses

- – ProgramNodeEnumerator in Smalltalk-80 compiler
- – IRIS Inventor scene rendering
- – TAO IDL compiler to handle different backends

Topic Objective

Topic : Creational Patterns (Abstract Factory)

In this topic, the students will learn what are Creational Patterns that help in real world problem and what are abstract factory method and how Creates an instance of several families of classes.

Creational Patterns

Creational patterns:-

- In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

- creational pattern provides one of the best ways to create an object.

- Creational design patterns are concerned with the way of creating objects. These design patterns are used when a decision must be made at the time of instantiation of a class (i.e. creating an object of a class).

Abstract Factory Pattern

Abstract Factory patterns:-

- Abstract Factory Pattern says that just define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes. That means Abstract Factory lets a class returns a factory of classes. So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.

- An Abstract Factory Pattern is also known as Kit.

Abstract Factory Pattern

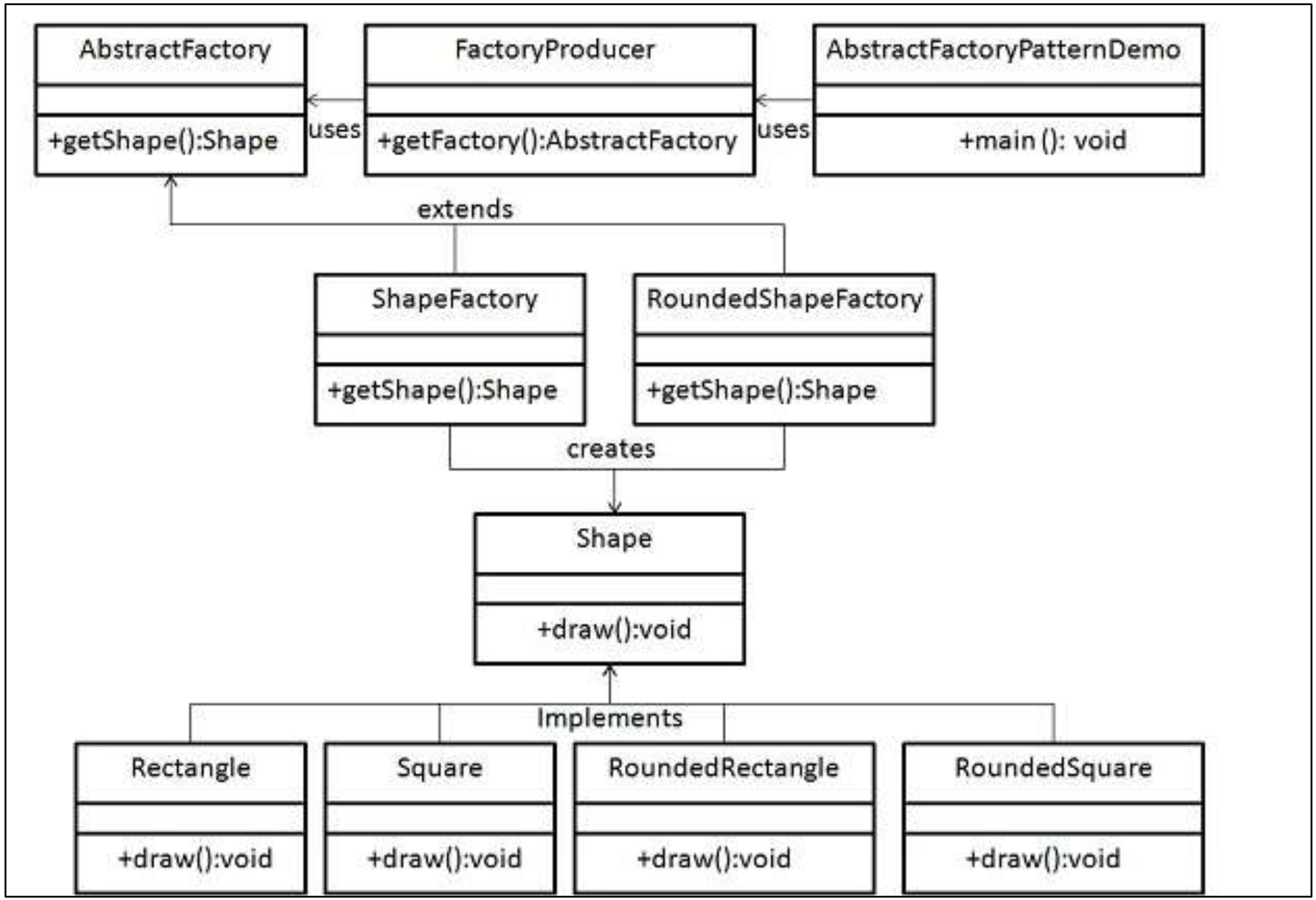
Advantage of Abstract Factory Pattern:-

- Abstract Factory Pattern isolates the client code from concrete (implementation) classes.
- It eases the exchanging of object families.
- It promotes consistency among objects.

Usage of Abstract Factory Pattern:-

- When the system needs to be independent of how its object are created, composed, and represented.
- When the family of related objects has to be used together, then this constraint needs to be enforced.
- When you want to provide a library of objects that does not show implementations and only reveals interfaces.

UML\Structure for Abstract Factory Pattern



Implementation

- We are going to create a Shape interface and a concrete class implementing it. We create an abstract factory class AbstractFactory as next step. Factory class ShapeFactory is defined, which extends AbstractFactory. A factory creator/generator class FactoryProducer is created.

- AbstractFactoryPatternDemo, our demo class uses FactoryProducer to get a AbstractFactory object. It will pass information (CIRCLE / RECTANGLE / SQUARE for Shape) to AbstractFactory to get the type of object it needs.

Implementation of Abstract Factory Pattern

Step 1

Create an interface for Shapes.

(Shape.java)

```
public interface Shape {
   void draw();
}
```

Implementation of Abstract Factory Pattern

Step 2

Create concrete classes implementing the same interface.

(RoundedRectangle.java)

```
public class RoundedRectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside RoundedRectangle::draw() method.");
   }
}
```

Implementation of Abstract Factory Pattern

Step 2 CONT…….

(RoundedSquare.java)

```
public class RoundedSquare implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside RoundedSquare::draw() method.");
   }
}
```

(Rectangle.java)

```
public class Rectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Implementation of Abstract Factory Pattern

Step 3

Create an Abstract class to get factories for Normal and Rounded Shape Objects.

AbstractFactory.java

```
public abstract class AbstractFactory {
   abstract Shape getShape(String shapeType) ;
}
```

Implementation of Abstract Factory Pattern

Step 4

Create Factory classes extending AbstractFactory to generate object of concrete class based on given information.

(ShapeFactory.java)

```
public class ShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}
```

Implementation of Abstract Factory Pattern

Step 4

(RoundedShapeFactory.java)

```
public class RoundedShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new RoundedRectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new RoundedSquare();
      }
      return null;
   }
}
```

Implementation of Abstract Factory Pattern

Step 5

Create a Factory generator/producer class to get factories by passing an information such as Shape

(FactoryProducer.java)

```
public class FactoryProducer {
   public static AbstractFactory getFactory(boolean rounded){
      if(rounded){
         return new RoundedShapeFactory();
      }else{
         return new ShapeFactory();
      }
   }
}
```

Implementation of Abstract Factory Pattern

Step 6

Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

(AbstractFactoryPatternDemo.java)

```
public class AbstractFactoryPatternDemo {
   public static void main(String[] args) {
      //get shape factory
      AbstractFactory shapeFactory = FactoryProducer.getFactory(false);
      //get an object of Shape Rectangle
      Shape shape1 = shapeFactory.getShape("RECTANGLE");
      //call draw method of Shape Rectangle
      shape1.draw();
```

Implementation of Abstract Factory Pattern

Cont……

```
        //get an object of Shape Square
        Shape shape2 = shapeFactory.getShape("SQUARE");
        //call draw method of Shape Square
        shape2.draw();
        //get shape factory
        AbstractFactory shapeFactory1 = FactoryProducer.getFactory(true);
        //get an object of Shape Rectangle
        Shape shape3 = shapeFactory1.getShape("RECTANGLE");
        //call draw method of Shape Rectangle
        shape3.draw();
        //get an object of Shape Square
        Shape shape4 = shapeFactory1.getShape("SQUARE");
        //call draw method of Shape Square
        shape4.draw();

    }
}
```

Implementation of Abstract Factory Pattern

Step 7

Step 7

Verify the output.

```
Inside Rectangle::draw() method.
Inside Square::draw() method.
Inside RoundedRectangle::draw() method.
Inside RoundedSquare::draw() method.
```

Topic Objective

Topic : Creational Patterns (Builder Design Pattern)

In this topic, the students will learn what are Creational Patterns that help in real world problem and what are Builder Design Pattern and how construct a complex object from simple objects using step-by-step approach.

Builder Design Pattern

Builder Design Pattern:-

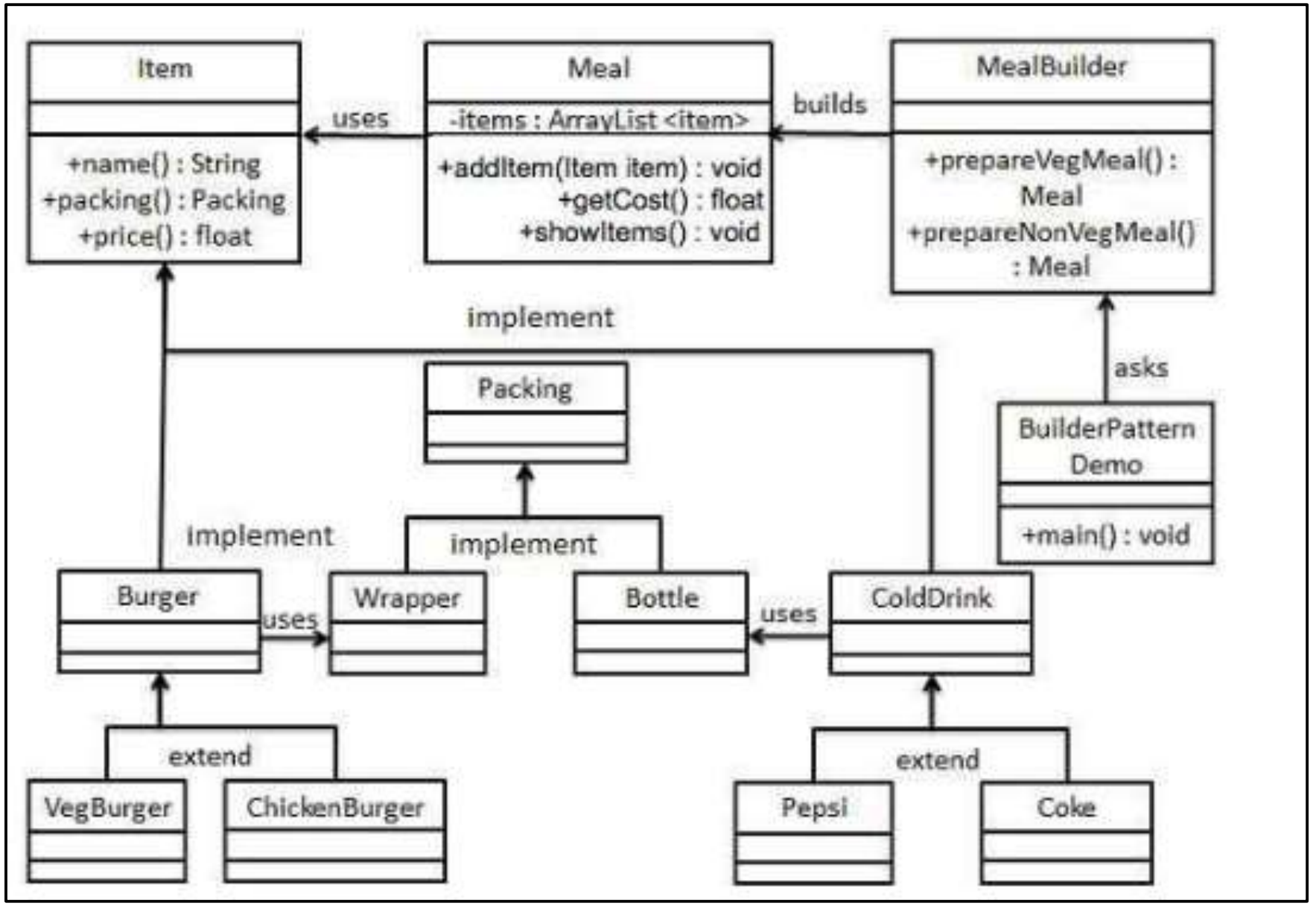
- Builder Pattern says that "construct a complex object from simple objects using step-by-step approach“.
- It is mostly used when object can't be created in single step like in the de-serialization of a complex object.

Advantage of Builder Design Pattern:-

The main advantages of Builder Pattern are as follows:

- It provides clear separation between the construction and representation of an object.
- It provides better control over construction process.
- It supports to change the internal representation of objects.

UML\Structure for Builder Design Pattern



Implementation

- We have considered a business case of fast-food restaurant where a typical meal could be a burger and a cold drink. Burger could be either a Veg Burger or Chicken Burger and will be packed by a wrapper. Cold drink could be either a coke or Pepsi and will be packed in a bottle.

- We are going to create an Item interface representing food items such as burgers and cold drinks and concrete classes implementing the Item interface and a Packing interface representing packaging of food items and concrete classes implementing the Packing interface as burger would be packed in wrapper and cold drink would be packed as bottle.

- We then create a Meal class having ArrayList of Item and a MealBuilder to build different types of Meal objects by combining Item. BuilderPatternDemo, our demo class will use MealBuilder to build a Meal.

Implementation of Builder Design Pattern

Step 1

Create an interface Item representing food item and packing.

Item.java

```
public interface Item {
   public String name();
   public Packing packing();
   public float price();
}
```

Packing.java

```
public interface Packing {
   public String pack();
}
```

Implementation of Builder Design Pattern

Step 2

Create concrete classes implementing the Packing interface.

Wrapper.java

```
public class Wrapper implements Packing {

   @Override
   public String pack() {
      return "Wrapper";
   }
}
```

Bottle.java

```
public class Bottle implements Packing {

   @Override
   public String pack() {
      return "Bottle";
   }
}
```

Step 3

Create abstract classes implementing the item interface providing default functionalities.

Burger.java

```
public abstract class Burger implements Item {

   @Override
   public Packing packing() {
      return new Wrapper();
   }

   @Override
   public abstract float price();
}
```

ColdDrink.java

```
public abstract class ColdDrink implements Item {

        @Override
        public Packing packing() {
       return new Bottle();
        }

        @Override
        public abstract float price();
}
```

Implementation of Builder Design Pattern

Step 4

Create concrete classes extending Burger and ColdDrink classes

VegBurger.java

```
public class VegBurger extends Burger {

   @Override
   public float price() {
      return 25.0f;
   }

   @Override
   public String name() {
      return "Veg Burger";
   }
}
```

ChickenBurger.java

```
public class ChickenBurger extends Burger {

   @Override
   public float price() {
      return 50.5f;
   }

   @Override
   public String name() {
      return "Chicken Burger";
   }
}
```

Coke.java

```
public class Coke extends ColdDrink {

   @Override
   public float price() {
      return 30.0f;
   }

   @Override
   public String name() {
      return "Coke";
   }
}
```

Pepsi.java

```
public class Pepsi extends ColdDrink {

   @Override
   public float price() {
      return 35.0f;
   }

   @Override
   public String name() {
      return "Pepsi";
   }
}
```

Implementation of Builder Design Pattern

Step 5

Create a Meal class having Item objects defined above.

Meal.java

```
import java.util.ArrayList;
import java.util.List;

public class Meal {
   private List<Item> items = new ArrayList<Item>();

   public void addItem(Item item){
      items.add(item);
   }
   public float getCost(){
      float cost = 0.0f;

      for (Item item : items) {
         cost += item.price();
      }
      return cost;
   }

   public void showItems(){

      for (Item item : items) {
         System.out.print("Item : " + item.name());
         System.out.print(", Packing : " + item.packing().pack());
         System.out.println(", Price : " + item.price());
      }
   }
}
```

Implementation of Builder Design Pattern

Step 6

Create a MealBuilder class, the actual builder class responsible to create Meal objects.

MealBuilder.java

```
public class MealBuilder {

   public Meal prepareVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new VegBurger());
      meal.addItem(new Coke());
      return meal;
   }
```

```
   public Meal prepareNonVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new ChickenBurger());
      meal.addItem(new Pepsi());
      return meal;
   }
}
```

Implementation of Builder Design Pattern

Step 7

BuiderPatternDemo uses MealBuider to demonstrate builder pattern.

BuilderPatternDemo.java

```
public class BuilderPatternDemo {
   public static void main(String[] args) {

      MealBuilder mealBuilder = new MealBuilder();

      Meal vegMeal = mealBuilder.prepareVegMeal();
      System.out.println("Veg Meal");
      vegMeal.showItems();
      System.out.println("Total Cost: " + vegMeal.getCost());

      Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
      System.out.println("\n\nNon-Veg Meal");
      nonVegMeal.showItems();
      System.out.println("Total Cost: " + nonVegMeal.getCost());
   }
}
```

Implementation of Builder Design Pattern

Step 8

Verify the output.

```
Veg Meal
Item : Veg Burger, Packing : Wrapper, Price : 25.0
Item : Coke, Packing : Bottle, Price : 30.0
Total Cost: 55.0


Non-Veg Meal
Item : Chicken Burger, Packing : Wrapper, Price : 50.5
Item : Pepsi, Packing : Bottle, Price : 35.0
Total Cost: 85.5
```

Topic Objective

Topic : Creational Patterns (Factory Design Pattern)

In this topic, the students will learn what are Creational Patterns that help in real world problem and what are Factory Pattern and how to define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.

Factory Design Pattern

Factory Design Pattern:-

- A Factory Pattern or Factory Method Pattern says that just define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate. In other words, subclasses are responsible to create the instance of the class.
- The Factory Method Pattern is also known as Virtual Constructor.

Advantage of Factory Design Pattern:-

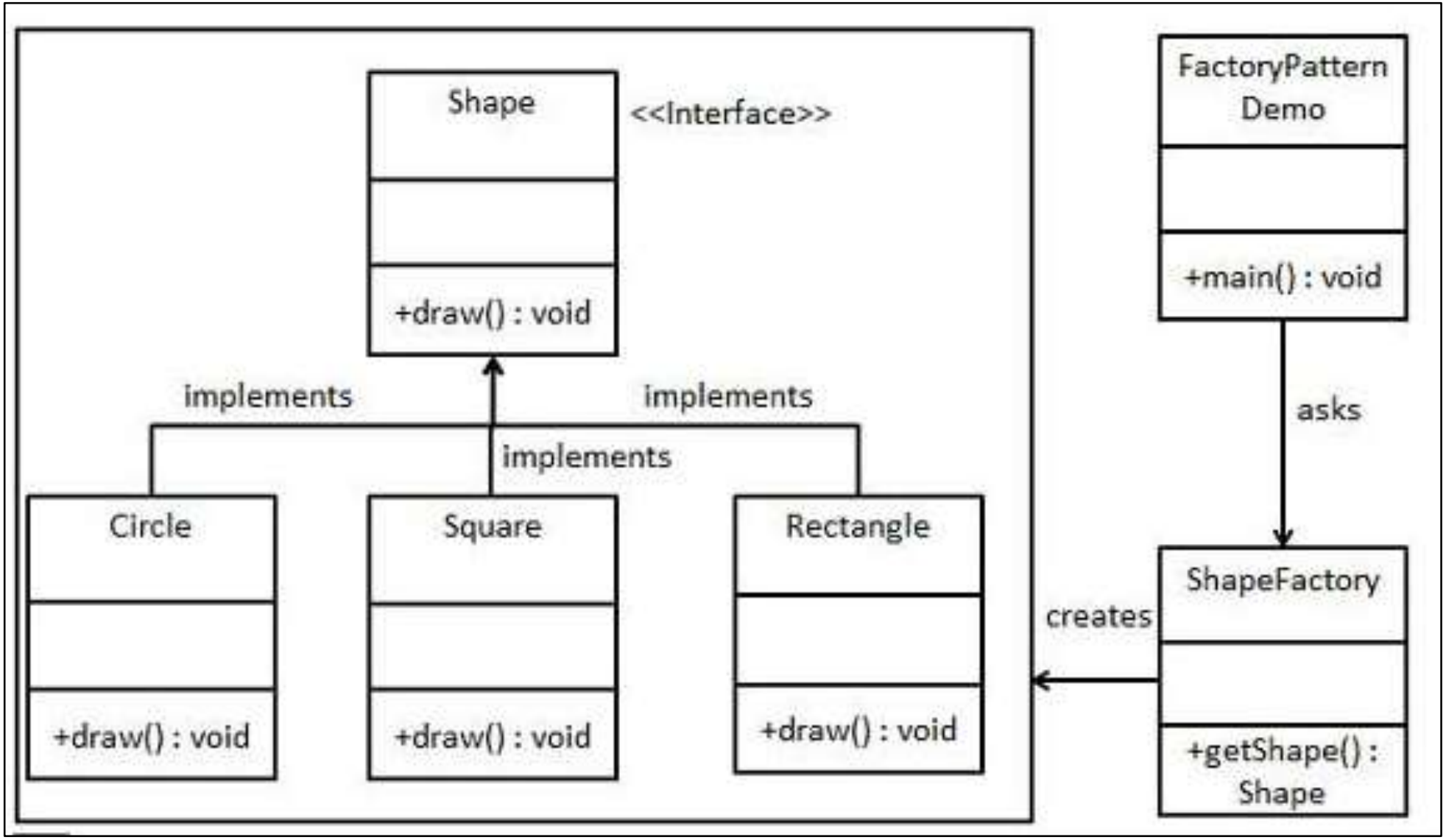
- Factory Method Pattern allows the sub-classes to choose the type of objects to create.
- It promotes the loose-coupling by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

Factory Design Pattern

Usage of Factory Design Pattern:-

- When a class doesn't know what sub-classes will be required to create.
- When a class wants that its sub-classes specify the objects to be created.
- When the parent classes choose the creation of objects to its sub-classes.
- Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

UML\Structure for Factory Design Pattern



Implementation

- We're going to create a Shape interface and concrete classes implementing the Shape interface. A factory class ShapeFactory is defined as a next step.
- FactoryPatternDemo, our demo class will use ShapeFactory to get a Shape object. It will pass information (CIRCLE / RECTANGLE / SQUARE) to ShapeFactory to get the type of object it needs.

Step 1

Create an interface.

Shape.java

```
public interface Shape {
   void draw();
}
```

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Implementation

Step 2

Square.java

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

Implementation

Step 3

Create a Factory to generate object of concrete class based on given information.

ShapeFactory.java

```
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();

      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }

      return null;
   }
}
```

Implementation

Step 4

Use the Factory to get object of concrete class by passing an information such as type.

FactoryPatternDemo.java

```
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of square
      shape3.draw();
   }
}
```

Implementation

Step 5

Verify the output.

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

Topic Objective

Topic : Creational Patterns (Prototype Design Pattern)

In this topic, the students will learn what are Creational Patterns that help in real world problem and what are Prototype Pattern and how to cloning of an existing object instead of creating new one and can also be customized as per the requirement.

Prototype Design Pattern

Prototype Design Pattern:-

- Prototype Pattern says that cloning of an existing object instead of creating new one and can also be customized as per the requirement.

- This pattern should be followed, if the cost of creating a new object is expensive and resource intensive.

Advantage of Prototype Design Pattern:-

The main advantages of prototype pattern are as follows:

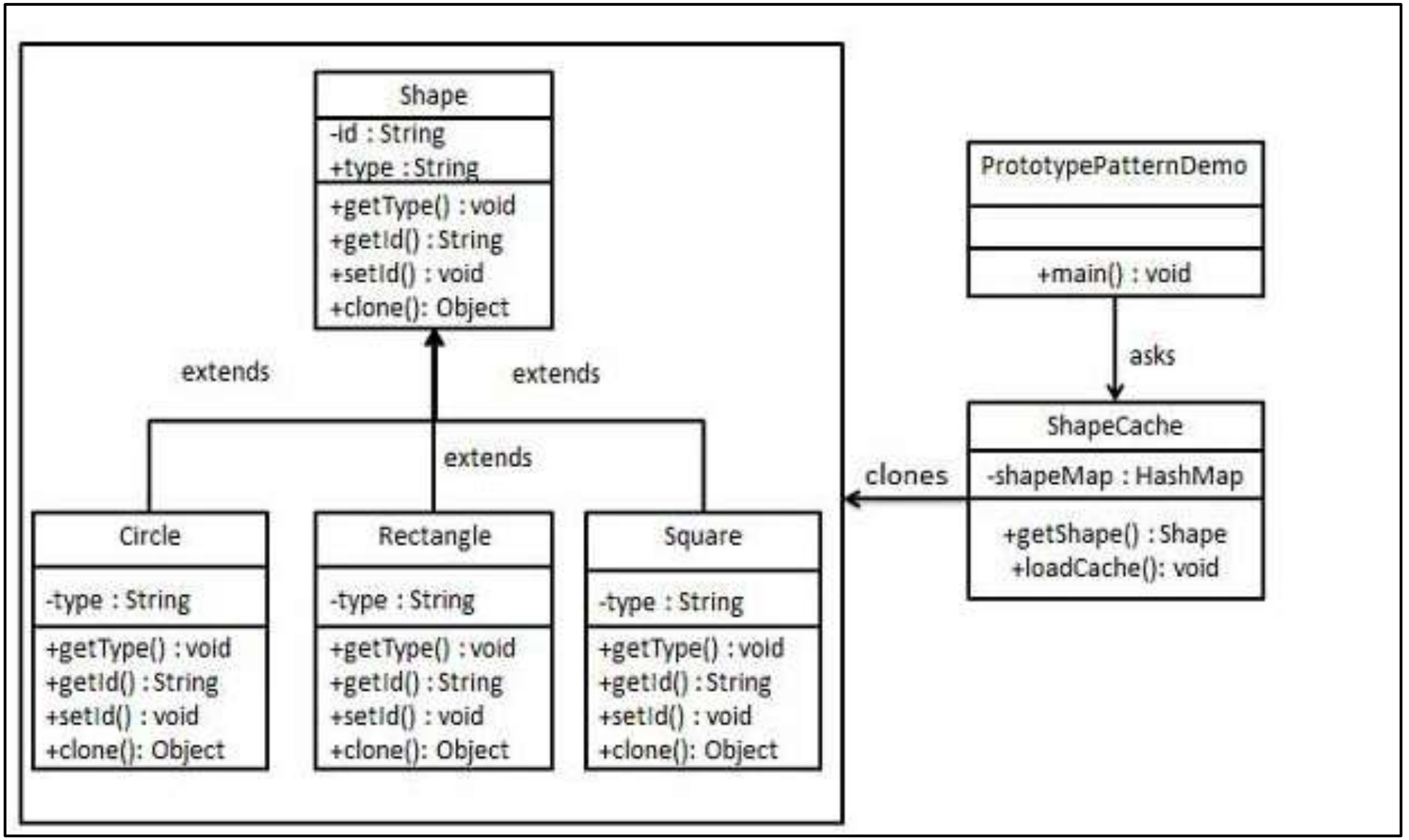
- ❑ It reduces the need of sub-classing.
- ❑ It hides complexities of creating objects.
- ❑ The clients can get new objects without knowing which type of object it will be.
- ❑ It lets you add or remove objects at runtime.

Prototype Design Pattern

Usage of Prototype Design Pattern:-

- When the classes are instantiated at runtime.
- When the cost of creating an object is expensive or complicated.
- When you want to keep the number of classes in an application minimum.
- When the client application needs to be unaware of object creation and representation.
- This pattern is used when creation of object directly is costly. For example, an object is to be created after a costly database operation. We can cache the object, returns its clone on next request and update the database as and when needed thus reducing database calls.

UML\Structure for Prototype Design Pattern



Implementation of Prototype Design Pattern

- Prototype pattern refers to creating duplicate object while keeping performance in mind. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
- This pattern involves implementing a prototype interface which tells to create a clone of the current object.
- We're going to create an abstract class Shape and concrete classes extending the Shape class. A class ShapeCache is defined as a next step which stores shape objects in a Hashtable and returns their clone when requested.
- PrototypPatternDemo, our demo class will use ShapeCache class to get a Shape object.

Implementation of Prototype Design Pattern

Step 1

Create an abstract class implementing *Clonable* interface.

Shape.java

```
public abstract class Shape implements Cloneable {

   private String id;
   protected String type;

   abstract void draw();

   public String getType(){
      return type;
   }

   public String getId() {
      return id;
   }

   public void setId(String id) {
      this.id = id;
   }

   public Object clone() {
      Object clone = null;

      try {
         clone = super.clone();

      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }

      return clone;
   }
}
```

Implementation of Prototype Design Pattern

Step 2

Create concrete classes extending the above class.

Rectangle.java

```
public class Rectangle extends Shape {

   public Rectangle(){
     type = "Rectangle";
   }

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square.java

```
public class Square extends Shape {

   public Square(){
     type = "Square";
   }

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Implementation of Prototype Design Pattern

Circle.java

```
public class Circle extends Shape {

   public Circle(){
     type = "Circle";
   }

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

Implementation of Prototype Design Pattern

Step 3

Create a class to get concrete classes from database and store them in a *Hashtable*.

ShapeCache.java

```
import java.util.Hashtable;

public class ShapeCache {

   private static Hashtable<String, Shape> shapeMap  = new Hashtable<String, Shape>();

   public static Shape getShape(String shapeId) {
      Shape cachedShape = shapeMap.get(shapeId);
      return (Shape) cachedShape.clone();
   }
```

Implementation of Prototype Design Pattern

```
    // for each shape run database query and create shape
    // shapeMap.put(shapeKey, shape);
    // for example, we are adding three shapes

    public static void loadCache() {
       Circle circle = new Circle();
       circle.setId("1");
       shapeMap.put(circle.getId(),circle);

       Square square = new Square();
       square.setId("2");
       shapeMap.put(square.getId(),square);

       Rectangle rectangle = new Rectangle();
       rectangle.setId("3");
       shapeMap.put(rectangle.getId(), rectangle);
    }
}
```

Implementation of Prototype Design Pattern

Step 4

PrototypePatternDemo uses *ShapeCache* class to get clones of shapes stored in a *Hashtable*.

PrototypePatternDemo.java

```
public class PrototypePatternDemo {
   public static void main(String[] args) {
      ShapeCache.loadCache();

      Shape clonedShape = (Shape) ShapeCache.getShape("1");
      System.out.println("Shape : " + clonedShape.getType());

      Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
      System.out.println("Shape : " + clonedShape2.getType());

      Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
      System.out.println("Shape : " + clonedShape3.getType());
   }
}
```

Implementation of Prototype Design Pattern

Step 5

Verify the output.

```
Shape : Circle
Shape : Square
Shape : Rectangle
```

Topic Objective

Topic : Creational Patterns (Singleton Design Pattern)

In this topic, the students will learn what are Creational Patterns that help in real world problem and what are Singleton Pattern and how a Singleton Pattern says that just "define a class that has only one instance and provides a global point of access to it".

Singleton Design Patterns

Singleton Design patterns:-

- Singleton Pattern says that just“ define a class that has only one instance and provides a global point of access to it".
- In other words, a class must ensure that only single instance should be created and single object can be used by all other classes.
- **There are two forms of singleton design pattern:-**

- Early Instantiation: creation of instance at load time.
- Lazy Instantiation: creation of instance when required.

Singleton Design Pattern

Advantage of Singleton Design Pattern:-

Saves memory because object is not created at each request. Only single instance is reused again and again.

Usage of Singleton design pattern :-

Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

Singleton Design Pattern

How to create Singleton design pattern:-

To create the singleton class, we need to have static member of class, private constructor and static factory method.

- **Static member:** It gets memory only once because of static, it contains the instance of the Singleton class.
- **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
- **Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

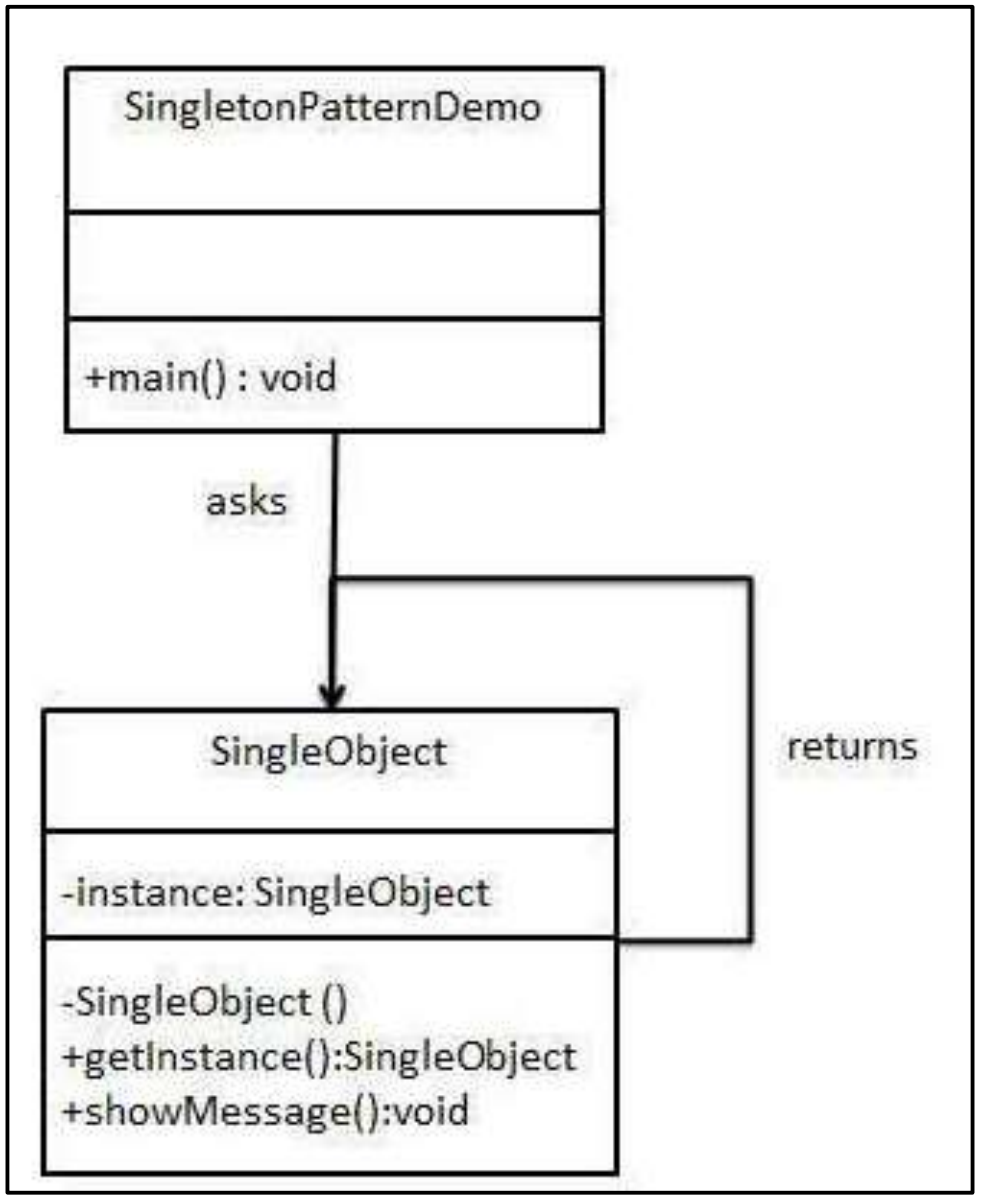
Implementation

Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

We're going to create a SingleObject class. SingleObject class have its constructor as private and have a static instance of itself.

- SingleObject class provides a static method to get its static instance to outside world. SingletonPatternDemo, our demo class will use SingleObject class to get a SingleObject object.

UML\Structure for Singleton Design Pattern



Implementation of Singleton Design Pattern

Step 1

Create a Singleton Class.

SingleObject.java

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

Implementation of Singleton Design Pattern

Step 2

Get the only object from the singleton class.

SingletonPatternDemo.java

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Step 3

Verify the output.

```
Hello World!
```

Daily Quiz

Which of the following pattern refers to creating duplicate object while keeping performance in mind.

A. Builder Pattern
B. Bridge Pattern
C. Prototype Pattern
D. Filter Pattern.

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. It lets subclasses redefine certain steps of an algorithm without changing the algorithm structure.

A. Chain of responsibility
B. Template method
C. Interpreter
D. Prototype

Daily Quiz

Which design pattern suggest multiple classes through which request is passed and multiple but only relevant classes carry out operations on the request.

A. Singleton pattern
B. Chain of responsibility pattern
C. State pattern
D. Bridge pattern

Most user interface design patterns fall with in one of ____ categories of patterns.

A. 5
B. 10
C. 25
D. 100

Weekly Assignment

1. What are Design Patterns in Java? What are the types of design patterns in Java.
2. What are the Creational Patterns and What Is Factory Pattern.
3. What Is Abstract Factory Pattern.
4. Explain Structural Patterns in Java.
5. Explain the Singleton pattern.

Topic Link (YouTube & NPTEL Video Links)

YouTube /other Video Links

- https://youtu.be/rI4kdGLaUiQ?list=PL6n9fhu94yhUbctIoxoVTrkIN3LMwTCmd
- https://youtu.be/v9ejT8FO-7I?list=PLrhzvIcii6GNjpARdnO4ueTUAVR9eMBpc
- https://youtu.be/VGLjQuEQgkI?list=PLt4nG7RVVk1h9lxOYSOGI9pcP3I5oblbx

MCQ (End of Unit)

1. Design patterns can be classified in _____ categories.

- ❑ 1
- ❑ 2
- ❑ 3
- ❑ 4

2. Which design patterns are specifically concerned with communication between objects?

- ❑ Creational Patterns
- ❑ Structural Patterns
- ❑ Behavioral Patterns
- ❑ J2EE Patterns

MCQ (End of Unit)

3. Which design pattern provides a single class which provides simplified methods required by client and delegates call to those methods?

- ❑ Adapter pattern
- ❑ Builder pattern
- ❑ Facade pattern
- ❑ Prototype pattern

4. Which design pattern suggests multiple classes through which request is passed and multiple but only relevant classes carry out operations on the request?

- ❑ Singleton pattern
- ❑ Chain of responsibility pattern
- ❑ State pattern
- ❑ Bridge pattern

Glossary Questions

Top 10 design pattern interview questions

1. What are design patterns?
2. How are design patterns categorized?
3. Explain the benefits of design patterns in Java.
4. Describe the factory pattern.
5. Differentiate ordinary and abstract factory design patterns.
6. What do you think are the advantages of builder design patterns?
7. How is the bridge pattern different from the adapter pattern?
8. What is a command pattern?
9. Describe the singleton pattern along with its advantages and disadvantages.
10. What are anti patterns?

Expected Questions for University Exam

- What are design patterns?
- How are design patterns categorized?
- Explain the benefits of design patterns in Java.
- Describe the factory pattern.
- Differentiate ordinary and abstract factory design patterns.
- What do you think are the advantages of builder design patterns?
- How is the bridge pattern different from the adapter pattern?
- What is a command pattern?
- Describe the singleton pattern along with its advantages and disadvantages.
- What are anti patterns?

Recap of Unit

- **Till Now we understand,** a case study in the design of a "What-You-See-Is-What-You-Get" (or "WYSIWYG") document editor called Lexi. We'll see how design patterns capture solutions to design problems in Lexi and applications like it.
- By the end of this chapter you will have gained experience with eight patterns, learning them by example.
- You also learn what are Creational Patterns that help in real world problem and what are abstract factory method and how Creates an instance of several families of classes.
- what are Prototype Pattern and how to cloning of an existing object instead of creating new one and can also be customized as per the requirement.