

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**COURSE FILE**

# **Design Patterns**



**(ESTD - 1999)**

**Vidya Jyothi Institute of Technology**

**(An AUTONOMOUS Institution)**

**(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi,  
Permanently Affiliated to**

**J.N.T. University, Hyderabad)**

**(Aziz Nagar, C.B.Post, Hyderabad -500075)**



# Vidya Jyothi Institute of Technology

(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi, permanently affiliated to JNTUH)

(An AUTONOMOUS Institution)

## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**REGULATION:** R15

**BATCH:** 2016-2020

**ACADEMIC YEAR:** 2019-2020

**PROGRAM:** B.Tech (COMPUTER SCIENCE AND ENGINEERING)

**YEAR/SEM:** IV/II

**COURSE NAME:** Design Patterns

**COURSE CODE:** A18543

**PRE REQUISITE:** Object Oriented Analysis and Design

**COURSE COORDINATOR:** R.R.S.Ravi kumar

### **COURSE INSTRUCTORS:**

1. K.S.R.K.Sarma
2. R.Yogesh
3. M.Kavya

*R.R.S. Ravi Kumar*

Course Coordinator

  
HOD-CSE



# Vidya Jyothi Institute of Technology

(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi, permanently affiliated to JNTUH)

(An AUTONOMOUS Institution)

## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

### COURSE FILE INDEX

S.NO.	DESCRIPTION
1.	Syllabus
2.	Text Books & Other References
3.	Time Table
4.	Program Outcomes (PO's) , Program Specific Outcomes (PSO's) & PEO's
5.	Mapping of Course Outcomes (CO's) With Program Outcomes (PO's) & Program Specific Outcomes (PSO's)
6.	Academic Calendar
7.	Course Schedule
8.	Lesson Plan
9.	Assignment Questions
10.	Mid Question Papers I & II
11.	Unit Wise Questions
12.	Minutes of Course Review Meeting
13.	Lecture Notes
14.	Power Point Presentations
15.	Semester End Question Papers
16.	Extra Topics delivered (if any)
17.	Innovations In Teaching and Learning
18.	Assessment Sheet – CO Wise (Direct Attainment)
19.	Course End Survey Form

# Syllabus



## Design Patterns

### B. Tech. IV Year II Semester

#### Course Outcomes:

L	T	P	C
3	0	0	3

At the end of this course, the student would be able to

1. Understand the Design patterns in software applications.
2. Discuss the Creational Patterns
3. Categorize the Structural Patterns.
4. Investigate Behavior Patterns
5. Construct the 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.

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

#### UNIT-II

##### **Creational Patterns**

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

#### UNIT-III

##### **Structural Patterns**

Adaptor, Bridge and Composite, Decorator, Facade, Flyweight, proxy

#### UNIT-IV

##### **Behavior Patterns**

Chain of Responsibility, Command, Interpreter, and Iterator, Mediator, Memento, Observer, State, strategy, Template Method, Visitor, Discussion on Behavioral Patterns.

#### UNIT-V

##### **What to Expect from Design Patterns**

A brief History, The Pattern Community, An Invitation, A Parting Thought.

#### **Text Books:**

1. Design Patterns By Erich Gamma, Pearson Education.

#### **Reference Books:**

1. Pattern's in JAVA Vol-I By Mark Grand, Wiley Dream Tech.
2. Pattern's in JAVA Vol – II BY Mark Grand, Wiley Dream Tech.

# **Text Books & other References**

## Text Books & other References

Text Books	
1	Design Patterns By Erich Gamma, Pearson Education.
Suggested / Reference Books	
1	Pattern's in JAVA Vol-I By Mark Grand, Wiley Dream Tech.
2	Pattern's in JAVA Vol – II BY Mark Grand, Wiley Dream Tech.
Other Resources	
1	<a href="https://sourcemaking.com/design_patterns">https://sourcemaking.com/design_patterns</a>
2	<a href="http://www.oodeesign.com">http://www.oodeesign.com</a>

# Vidya Jyothi Institute of Technology

Department of Computer Science and Engineering

Sec: **CSE-D**

Year/Sem: **IV - II**

**W.E.F: 09/12/2019**

ROOM NO: **N307**

DAY	9.00 – 9.55	9.55 – 10.50	10.50 – 11.45	11.45 – 12.30	12.30 – 1.25	1.25 – 2.20	2.20 – 3.15	3.15-4.05
MON	SWSN	DP	E-COM	LUNCH	E-COM	SWSN	DP	
TUE	E-COM	SWSN	E-COM		DP		SWSN	
WED	DP	SWSN	E-COM		SEMINAR			
THU	←Project Work→				←Project Work→			
FRI	←Project Work→				←Project Work→			
SAT	Department Events				Department Events			

## Subject

## Name of the Faculty

DP	Design Patterns	Mr. R.R.S.Ravi Kumar
E-Commerce	E-Commerce	Dr.Moulali
SWSN	Semantic Web and Social Networks	Ms.K.Keerthi
SEMINAR		Mr.R.R.S.Ravi Kumar

Class Incharge  
IV YEAR Coordinator

Mr. R.R.S. Ravi Kumar  
Mr. Y.Prabhu Kumar

  
**Time Table I/C**

  
**H.O.D**

**Program  
Outcomes(PO's),  
Program Specific  
Outcomes (PSO's)  
& PEO's**



## Program Outcomes (PO's):

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization for the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools, including prediction and modelling to complex engineering activities, with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with the society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

## Program Specific Outcomes (PSO's):

**PSO 1:** The ability to design and develop Algorithms to provide optimized solutions for societal needs

**PSO 2:** Apply standard approaches and practices in Software Project Development through trending technologies



### **Program Educational Objectives (PEO's)**

**PEO1:** Enhance the employability of the graduate in software industries/Public sector/Research organizations

**PEO2:** Acquire analytical and computational abilities to pursue higher studies for professional growth

**PEO3:** Work in multidisciplinary project teams with effective communication skills and leadership qualities

**PEO4:** Develop professional ethics among the students and promote entrepreneurial abilities

**Mapping of  
Course Outcomes(CO's)  
With  
Program Outcomes(PO's)  
&  
Program Specific  
Outcomes(PSO's)**



**VIDYA JYOTHI INSTITUTE OF TECHNOLOGY**  
**Department of Computer Science & Engineering**

Year & Sem: IV Year II Sem

Course name: Design Patterns

Course Code: A18543

Regulation: R15

**COURSE OUTCOMES:**

After completing this course the student must demonstrate the knowledge and ability to	
CO1	Understand the Design patterns in software applications.
CO2	Discuss the Creational Patterns.
CO3	Categorize the Structural Pattern.
CO4	Investigate Behavioral Patterns.
CO5	Construct the good design pattern structures.

**CO -PO MAPPING:**

	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12
CO 1	3	3	1	2	1	2	1	1	3	3	2	3
CO 2	3	2	3	3	2	1	1	2	3	3	2	3
CO 3	3	2	3	3	3	2	1	1	3	3	2	3
CO 4	3	2	3	2	3	2	2	2	2	3	2	3
CO 5	2	2	2	1	2	1	1	2	2	2	2	2
Avg	3	2	3	2	3	2	1	2	3	3	2	3

**CO - PSO MAPPING:**

	PSO1	PSO2
CO1	3	3
CO2	3	2
CO3	2	3
CO4	3	3
CO5	3	3
Avg	3	3

*RRS Ravi Kumar*

Signature of Coordinator

Signature of HOD



# **Academic Calendar**



## Academic Calendar



### Vidya Jyothi Institute of Technology (Autonomous)

(Accredited by NAAC & NBA, Approved by A.I.C.T.E., New Delhi, Permanently Affiliated to JNTU, Hyderabad)  
(Aziz Nagar, C.B.Post, Hyderabad -500075)

#### II/III/IV B.Tech I & II Semester Academic Calendar for the Academic Year 2019-20

II/III/IV YEAR I SEMESTER		Commencement of Class Work 17.06.2019	
	From	To	Duration
I Spell of Instruction	17.06.2019	10.08.2019	8 WEEKS
I Mid Examinations	13.08.2019	17.08.2019	4 DAYS
II Spell of Instruction	19.08.2019	05.10.2019	7 WEEKS
Dussehra Holidays	07.10.2019	19.10.2019	2 WEEKS
II Spell of Instruction Continuation	21.10.2019	26.10.2019	1 WEEK
II Mid Examinations	28.10.2019	31.10.2019	4 DAYS
Practical Examinations	01.11.2019	03.11.2019	3 DAYS
End Semester Examinations	04.11.2019	20.11.2019	2 WEEKS 3 DAYS
Betterment Examinations	21.11.2019	23.11.2019	3 DAYS
Supplementary Examinations	25.11.2019	07.12.2019	2 WEEKS
II/III/IV YEAR II SEMESTER		Commencement of Class Work 09.12.2019	
I Spell of Instruction	09.12.2019	10.01.2020	5 WEEKS
Sankranti Holidays	11.01.2020	15.01.2020	5 DAYS
Technical/Sports fest	16.01.2020	18.01.2020	3 DAYS
I Spell of Instruction Continuation	20.01.2020	08.02.2020	3 WEEKS
I Mid Examinations	10.02.2020	15.02.2020	1 WEEK
II Spell of Instruction	17.02.2020	11.04.2020	8 WEEKS
II Mid Examinations	13.04.2020	17.04.2020	4 DAYS
Practical Examinations	18.04.2020	22.04.2020	4 DAYS
Betterment Examinations	23.04.2020	25.04.2020	4 DAYS
End Semester Examinations	27.04.2020	12.05.2020	2 WEEKS 2 DAYS
Supplementary Examinations	13.05.2020	30.05.2020	2 WEEKS 4 DAYS
Commencement of classes will be from 15.06.2020			



 DIRECTOR



# **Course Schedule**

## Course Schedule

### Distribution of Hours in Unit – Wise

Unit	Topic	Book1	Total No. of Hours
I	<p><b>Introduction:</b> 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.</p> <p><b>A Case Study:</b> Designing a Document Editor, Design Problems, Document Structure, Formatting Embellishing the User Interface, Supporting Multiple Look and Feel Standards, Supporting Multiple Window systems, User Operations Spelling Checking and Hyphenation, Summary.</p>	T1/1-76	16
II	<p><b>Creational Patterns</b> Abstract Factory, Builder, Factory Method, Prototype, Singleton, Discussion of Creational Patterns.</p>	T1/81-135	6
III	<p><b>Structural Patterns</b> Adaptor, Bridge and Composite, Decorator, Facade, Flyweight, proxy.</p>	T1/137-219	8
IV	<p><b>Behavior Patterns</b> Chain of Responsibility, Command, Interpreter, and Iterator, Mediator, Memento, Observer, State, strategy, Template Method, Visitor, Discussion on Behavioral Patterns.</p>	T1/221-345	12
V	<p><b>What to Expect from Design Patterns</b> A brief History, The Pattern Community, An Invitation, A Parting Thought.</p>	T1/351-359	6
Total contact classes for syllabus coverage			48



# Vidya Jyothi Institute of Technology

(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi, permanently affiliated to JNTUH)

(An AUTONOMOUS Institution)

## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**SUBJECT:** Design Patterns

**ACADEMIC YEAR:** 2019-20

**NAME:** R R S Ravi kumar

**YEAR/SEM/SECTION:** IV B.TECH/II SEM

### LESSON PLAN

S.No	Topic Name	Teaching Learning Process
Unit-I		
1.	Introduction. What is a Design  Pattern?	From Known to Unknown, Chalk & Board
2.	Design Patterns in Small Talk MVC.	Chalk & Board
3.	Describing Design Patterns	From Known to Unknown, Chalk & Board
4.	The Catalog of Design Patterns	Chalk & Board
5.	Organizing the catalog	Chalk & Board
6.	How Design Patterns Solve design Problems	Chalk & Board
7.	How to Select a Design Pattern	Chalk & Board
8.	How to Use a Design Pattern	Chalk & Board
9.	A Case Study: Designing a Document Editor	From Known to Unknown, Chalk & Board
10.	Design Problems,	PPT
11.	Document Structure,	PPT
12.	Formatting Embellishing the User Interface,	PPT
13.	Supporting Multiple Look and Feel Standards,	PPT
14.	Supporting Multiple Window systems,	PPT
15.	User Operations	PPT

16.	Spelling Checking and Hyphenation, Summary	PPT,Online quiz
<b>Unit-II</b>		
17.	Creational Patterns,Introduction	PPT
18.	Abstract Factory	PPT
19.	Builder	From Known to Unknown, PPT
20.	Factory Method	Chalk & Board
21.	Prototype	Chalk & Board,Online quiz
<b>Unit-III</b>		
22.	Structural Patterns.Adapter	From Known to Unknown, Chalk & Board
23.	Bridge	Chalk & Board
24.	Composite	Chalk & Board
25.	Decorator	Chalk & Board
26.	Façade	Chalk & Board
27.	Flyweight	Chalk & Board
28.	Proxy	Chalk & Board
29.	Structural Patterns.Adapter	Chalk & Board
<b>Unit-IV</b>		
30.	Behavioral Patterns.Chain of Responsibility	From Known to Unknown ,Chalk & Board
31.	Command	Chalk & Board
32.	Interpreter	Chalk & Board
33.	Iterator	Chalk & Board
34.	Mediator	Chalk & Board
35.	Memento	Chalk & Board
36.	Observer	Chalk & Board
37.	State	Chalk & Board

38.	Strategy	PPT
39.	Template Method	PPT
40.	Visitor	PPT
41.	Discussion of Behavioral Patterns	PPT
<b>Unit-V</b>		
42.	What to Expect from Design Patterns.	Chalk & Board
43.	A Brief History,	Chalk & Board
44.	The Pattern Community An Invitation	Chalk & Board
45.	A Parting Thought	PPT

*R R S Ravikumar*

**Course Coordinator**

*[Red Signature]*

**HOD-CSE**



# **Assignment Questions**





# Vidya Jyothi Institute of Technology

(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi, permanently affiliated to JNTUH)

(An AUTONOMOUS Institution)

## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

### ASSIGNMENT 1

**Branch:** CSE

**Year&Sem:** IV-II

**SUB:** Design Patterns

**Academic Year:** 2019-20

**Faculty Name:** R.R.S.Ravi Kumar

**Marks:** 25M

S.No	Question	Marks	CO	BL	PO's
1	What is a Design Pattern?Discuss about Design Patterns in Smalltalk MVC.	5	1	L2	1-12
2	List the Catalog of Design Patterns.	5	1	L2	1-12
3	Explain in detail about abstract factory design patterns	5	2	L2	1-12
4	Write the intent of Singleton.Describe the motivation,structure and participants of singleton.	5	2	L1,L2	1-12
5	Discuss about Bridge design pattern	5	3	L1,L2	1-12



# Vidya Jyothi Institute of Technology

(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi, permanently affiliated to JNTUH)

(An AUTONOMOUS Institution)

## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

### ASSIGNMENT 2

**Branch:** CSE

**Year&Sem:** IV-II

**SUB:** Design Patterns

**Academic Year:** 2019-20

**Faculty Name:** R.R.S.Ravi kumar

**Marks:** 25M

S.No	Question	Marks	CO	BL	PO's
1	What is a composite design pattern? Discuss about the collaborations and consequences of composite.	5	3	L2	1-12
2	List the behavioral design patterns. Compare state and strategy design patterns	5	4	L1,L2	1-12
3.	Illustrate about Observer and Factory design Patterns.	5	4	L4	1-12
4.	Discuss about what to expect from design patterns.	5	5	L1,L2	1-12
5.	Discuss about Alexander patterns for languages	5	5	L1,L2	1-12

# • **Mid Question Papers**





# Vidya Jyothi Institute of Technology (Autonomous)

(Accredited by NAAC & NBA, Approved By A.I.C.T.E., New Delhi, Permanently Affiliated to JNTU, Hyderabad)  
(Aziz Nagar, C.B.Post, Hyderabad -500075)

## IV Year B.Tech II Semester I Mid Examination

Branch: CSE

Duration: 90Min

Subject: Design Patterns

Marks: 20

Date: 17.02.2020

Session: FN

### Course Outcomes:

1. Understand the Design Patterns in Software Applications
2. Discuss the Creational Patterns
3. Categorize the Structural Patterns
4. Investigate Behavioral Patterns
5. Construct the good design pattern structures

### Bloom's Level:

Remember	I
Understand	II
Apply	III
Analyze	IV
Evaluate	V
Create	VI

### PART-A (3Q×2M =6Marks)

		Outcomes		BL	Marks
ANSWER ALL THE QUESTIONS		CO	PO		
1.i)	What is a design pattern?	1	1 to 12	1	2
[OR]					
ii)	List the various behavioral patterns.	1	1 to 12	1	2
2.i)	What is the intent of abstract factory?	2	1 to 12	1	2
[OR]					
ii)	List the participants of prototype.	2	1 to 12	1	2
3.i)	Write the collaborations of composite.	3	1 to 12	2	2
[OR]					
ii)	What is the intent of bridge?	3	1 to 12	2	2

### PART-B (5+5+4= 14 Marks)

		Outcomes		BL	Marks
ANSWER ALL THE QUESTIONS		CO	PO		
4.i.a)	Write the general template used for describing the design patterns.	1	1 to 12	2	3
b)	List and Explain various ways how to organizing the Catalog.	1	1 to 12	3	2
[OR]					
ii.a)	Explain the Spell checking and Hyphenation in Lexi.	1	1 to 12	3	3
b)	Discuss the various patterns in design of Lexi.	1	1 to 12	2	2
5. i.a)	Write the Intent, Motivation and structure of the Factory method Pattern.	2	1 to 12	2	3
b)	Write the Consequences, sample code and implementation of the Builder.	2	1 to 12	2	2
[OR]					
ii.a)	Discuss the structure, Participants and Collaborations of Singleton.	2	1 to 12	2	3
b)	What are the consequences and applicability of the abstract factory Pattern?	2	1 to 12	2	2
6.i)	What are the Intent, motivation and structure of the Composite Pattern?	3	1 to 12	1	4
[OR]					
ii)	Discuss about the Adapter Pattern.	3	1 to 12	4	4





# Vidya Jyothi Institute of Technology (Autonomous)

(Accredited by NAAC & NBA, Approved By A.I.C.T.E., New Delhi, Permanently Affiliated to JNTU, Hyderabad)  
(Aziz Nagar, C.B.Post, Hyderabad -500075)

## IV Year B.Tech II Semester 2<sup>nd</sup> Mid Exam

Branch: Computer Science and Engineering

Duration: 90Min

Sub:Design Patterns

Marks: 20

Date:

Session:

### Course Outcomes:

- CO1.Understand the Design Patterns in Software Applications
- CO2.Discuss the Creational Patterns
- CO3.Categorize the Structural Patterns
- CO4.Investigate Behavior Patterns
- CO5.Construct the good design pattern structures

### Bloom Levels:

Remember	I
Understand	II
Apply	III
Analyze	IV
Evaluate	V
Create	VI

PART-A (3Q×2M =6 Marks)		Course Outcomes		Bloom Levels
ANSWER ALL THE QUESTIONS		CO	PO	
1	What is Flyweight pattern.	CO3	P1 to P12	L1
2	What are the Known uses of Mediator Pattern.	CO4	P1 to P12	L2
3	Write A Parting Thought.	CO5	P1 to P12	L1
PART-B (5+5+4= 14 Marks)		Course Outcomes		Bloom Levels
ANSWER ALL THE QUESTIONS		CO	PO	
4.i)	Write the Intent Motivation of Façade Patterns and List the Uses.	CO3	P1 to P12	L2
[OR]				
ii)	Draw the Structure of the Decorator and Write the Implementation of it.	CO3	P1 to P12	L4
5.i)	What are the Consequences and Sample code of the Template method Pattern.	CO4	P1 to P12	L5
[OR]				
ii.)	Discuss the Intent, Applicability and Implementation of State Pattern.	CO4	P1 to P12	L4
6.i)	Explain what to expect from Design Patterns	CO5	P1 to P12	L2
[OR]				
ii)	Write about Alexander's Pattern Languages.	CO5	P1 to P12	L2

\*\*\*VJIT(A)\*\*\*

Controller of Examinations

DIRECTOR

# **Unit wise Questions**



Design Patterns unit wise questions

**Unit-I**

**Short Answer Questions.**

1. List the GOF in design patterns.
2. What is Design Pattern?
3. List down the four essential elements of patterns.
4. Describe the MVC architecture
5. Define encapsulation
6. Write notes on Operation's signature in an object.
7. Write notes on interfaces in object oriented systems.
8. Describe dynamic binding.
9. Write notes on abstract classes.
10. Define concrete class.
11. Define White-box reuse.
12. Define black box reuse.
13. List disadvantages of class inheritance.
14. Describe delegation
15. What is Acquaintance relationship of objects?
16. Describe the use Framework.
17. Outline the different Toolkits.
18. Discuss Aggregation.
19. What is object granularity?
20. Briefly describe how to Use a Design Pattern

**Long Answer Questions.**

1. Illustrate about Describing Design Patterns.
2. Compose and describe the MVC architecture in Smalltalk.
3. List and Identify various ways how to organizing the design patterns.
4. Write the intents of various design patterns in the catalog of design Patterns.
5. Explain how to describe design Patterns.
6. Examine how Design Patterns Solve Design Problems.
7. Identify the process of how to Select a Design Pattern and how to Use a Design Pattern.

**UNIT-II**

**Short Answer Questions.**

1. What is creational design pattern?
2. Define singleton
3. List the Examples of creational design patterns
4. Write about factory method
5. List the uses of factory method.

6. Write the main disadvantage of factory method.
7. Define Prototype method
8. Write the intent of Builder.
9. Identify the use Builder.
10. Identify the use of singleton.
11. List the benefits of singleton
12. List the benefits of Builder.
13. Write the disadvantages of Prototype method.
14. List the Related patterns of Builder.
15. Define the intent Abstract factory.
16. What is Abstract factory?
17. Write the Intent, structure of Singleton Pattern.
18. Describe Applicability and Implementation of Singleton Pattern.
19. Write the Consequences of the Builder Pattern
20. Write the Sample code of the Builder Pattern

### **Long Answer Questions.**

1. Compare and contrast two techniques for embellishing user interface.
2. Discuss about creational Patterns
3. Write the Intent, Motivation and structure of the Abstract Factory Pattern.
4. Compose the Consequences and Sample code of the Builder Pattern.
5. Write the structure, sample code and implementation of the Factory Method.
6. Illustrate the Intent, Applicability, Motivation of Prototype design pattern.
7. Discuss the Intent, structure, Applicability and Implementation of Singleton Pattern.

### **UNIT-III**

#### **Short Answer Questions.**

1. Write notes on Structural Pattern.
2. Identify the use of Adapter Pattern.
3. Write the use of Bridge Pattern.
4. Define composite Pattern.
5. Define decorator pattern.
6. Write notes on applicability of decorator pattern.
7. Identify when to use Façade Pattern.
8. What is Flyweight pattern?
9. How Decorator Pattern considered as structural pattern.
10. What is the other name of Proxy?
11. Name the related Patterns of Proxy.
12. List the related Patterns of Flyweight
13. List the related Patterns of Adapter
14. Find the related Patterns of Composite
15. List the Known uses of Adapter
16. Write the Known uses of Bridge pattern
17. Write about Intent and Motivation of the Flyweight pattern.
18. How to Implement the Flyweight pattern.
19. Restate the consequences, Collaborations of Proxy pattern

20. Write the sample code of Proxy pattern

### **Long Answer Questions**

1. Write the Intent, Motivation and structure of the Adapter Pattern.
2. Illustrate the Consequences and Sample code of the Bridge Pattern.
3. Discuss about the structure, sample code and implementation of the Composite.
4. Explain the Intent, Applicability, Motivation of Decorator design pattern.
5. Discuss the Intent, structure, Applicability and Implementation of Singleton Pattern.
6. Write the Motivation and Sample code of Façade Patterns and List the Uses.
7. Design the Structure of the Decorator and Write the Implementation of it.

## **UNIT-IV**

### **Short answer questions**

1. Define the Behavioral Patterns.
2. List the Different Behavioral Patterns.
3. Write the intent of Chain of Responsibility.
4. Identify the Known uses of Chain of responsibility.
5. Define the intent of command Pattern.
6. Describe the Known uses of command Pattern.
7. Write the intent of Iterator pattern.
8. Describe the Application of Strategy Pattern.
9. Define the intent of Visitor.
10. List down the uses mediator pattern.
11. Define the intent of Memento pattern.
12. Define the intent of Observer.
13. Write down the Known uses of Mediator Pattern.
14. List the Known uses of Memento Pattern.
15. Write the Known uses of Observer Pattern.
16. Identify the Collaborations of Command Pattern.
17. Differentiate Mediator and Memento.
18. State the Applicability of Iterator Pattern
19. Write the Collaborations of State Pattern
20. List the Advantages of State Pattern

### **Long Answer Questions.**

1. Describe the Implementation and sample code of Chain of responsibility.
2. Discuss the Intent, Applicability and Implementation of Command Pattern.
3. Explain the Motivation, Structure and Consequences of Interpreter
4. Compare and Contrast between State and strategy.
5. State the Consequences and Write the Sample code of the Template method Pattern.
6. Construct the sample code and implementation of the Visitor
7. Discuss the Intent, Applicability, Motivation Strategy design pattern.



## **UNIT-V**

### **Short answer questions**

1. List the Phases in object oriented software.
2. Define Refactoring.
3. Describe Prototyping.
4. What is expansionary?
5. Write about consolidating.
6. What can you do if you are interested in Patterns?
7. Write A Parting Thought.
8. Describe Reuse.
9. Write about Documentation.
10. Write the four ways of in which our works are different.

### **Long Answer Questions.**





1. Discuss what to expect from Design Patterns?
2. Discuss the Alexander's Patterns and Patterns in software in the Pattern Community.
3. Explain the Patterns in software.
4. Illustrate the Brief History of Design Patterns.
5. Illustrate concept of The Pattern Community.

# **Course Review Meeting**



## Meeting 2

Date: 18/03/2020




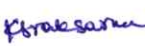
Details of Meeting No – 2	
Date of Meeting	18/03/2020
Member's Present	1. Mr. R.R.S.Ravi Kumar 2. Ms. M.Kavya 3. Mr. R.Yogesh 4. Mr.K.S.R.K.Sarma
Details	<ul style="list-style-type: none"><li>• Preparation of Unit wise questions and give assignment to students</li><li>• Status of Syllabus coverage of Mid II and instructions to complete the syllabus</li></ul>
Signatures	1.  2.  3.  4. 

R-R-S. Ravi Kumar  
Course Coordinator

  
CSE-HOD

## Meeting 1

Date: 02/02/2020

Details of Meeting No – 1	
Date of Meeting	02/02/2020
Member's Present	1. Mr. R.R.S.Ravi Kumar 2. Ms. M.Kavya 3. Mr. R.Yogesh 4. Mr.K.S.R.K.Sarma
Details	Points discussed in the meeting: <ul style="list-style-type: none"><li>• Preparation of Unit wise questions and give assignment to students</li><li>• Discussion of Teaching Learning Practices</li><li>• Status of Syllabus coverage of Mid I and instructions to complete the syllabus</li></ul>
Signatures	1.  2.  3.  4. 

R.R.S. Ravi Kumar  
Course Coordinator

  
CSE-HOD

# Lecture Notes

# CONTENTS

## UNIT-I

### Introduction:

1. What is a design pattern?
2. Design patterns in Smalltalk MVC
3. Describing Design Patterns
4. The Catalog of Design Patterns
5. Organizing the Catalog
6. How Design Patterns Solve Design Problems
7. How to Select a Design Pattern
8. How to use a Design Pattern.

## UNIT-II

### A Case Study:

1. Designing a Document Editor Design Problems,
2. Document Structure
3. Formatting
4. Embellishing the User Interface
5. Supporting Multiple Look-and Feel Standards
6. Supporting Multiple Window Systems,
7. User Operations Spelling Checking and Hyphenation,
8. Summary.
9. Creational Patterns: Abstract Factory
10. Builder
11. Factory Method
12. Prototype
13. Singleton
14. Discussion of Creational Patterns

## UNIT-III

### Structural Pattern Part-I :

1. Adapter
2. Bridge
3. Composite

### Structural Pattern Part-II :

1. Decorator
2. Façade
3. Flyweight
4. Proxy

## **UNIT-IV**

### **Behavioural Patterns Part-I :**

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator

### **Behavioural Patterns Part-II :**

1. Mediator
2. Memento
3. Observer

## **UNIT-V**

### **Behavioural Patterns Part-II (cont'd):**

1. State
2. Strategy
3. Template Method
4. Visitor
5. Discussion of Behavioral Patterns
6. What to Expect from Design Patterns
7. A Brief History
8. The Pattern Community An Invitation
9. A Parting Thought



## **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 reflect the state of the model.

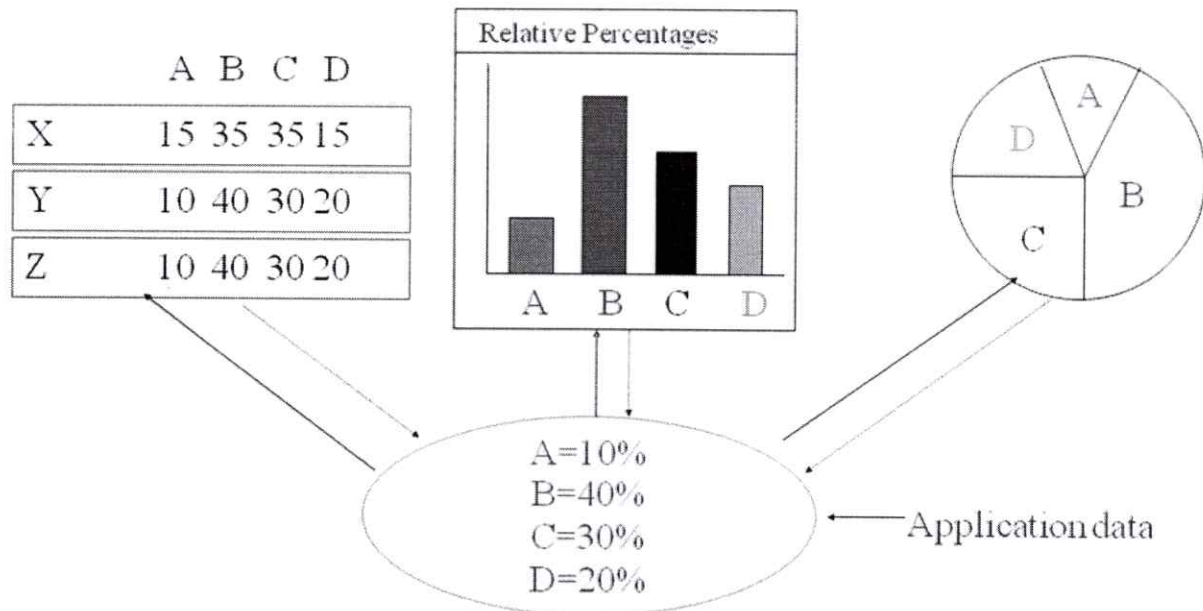
Whenever the model's data changes, the model notifies views that depends on it.

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 its 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.

### 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 construction process can create different representations.
- Chain of Responsibility: Avoid coupling the sender of a request to its 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 object 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 subclassing 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 its 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*

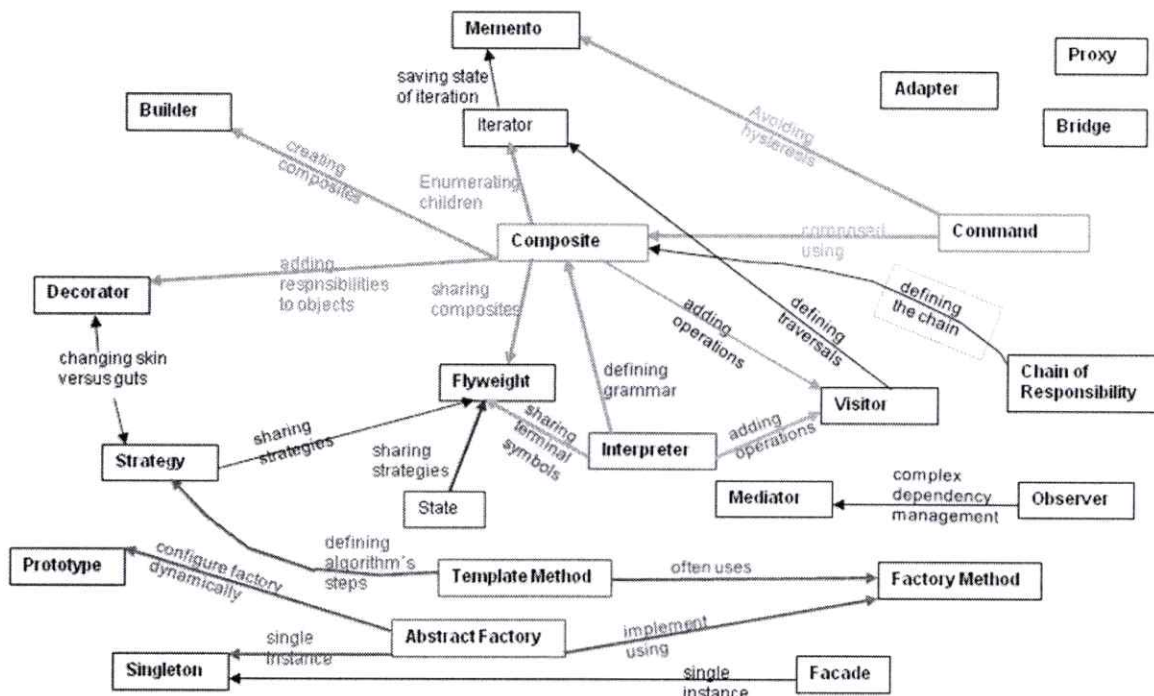
## How Design Patterns Solve Design Problems:

- Finding Appropriate Objects
  - Decomposing a system into objects is the hard part
  - 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



## 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**

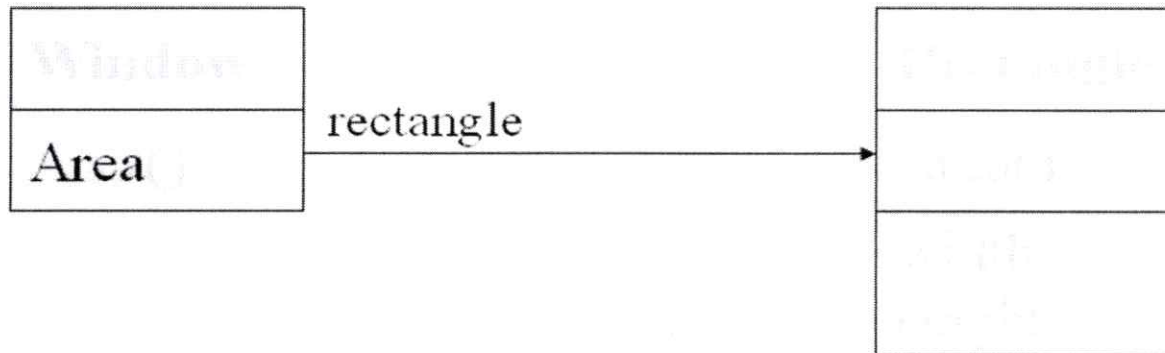
- **White-box** reuse:
  - Reuse by subclassing (class inheritance)
  - Internals of parent classes are often visible to subclasses
  - works statically, compile-time approach
  - Inheritance breaks encapsulation
- **Black-box** reuse:
  - Reuse by object composition
  - Requires objects to have well-defined 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.

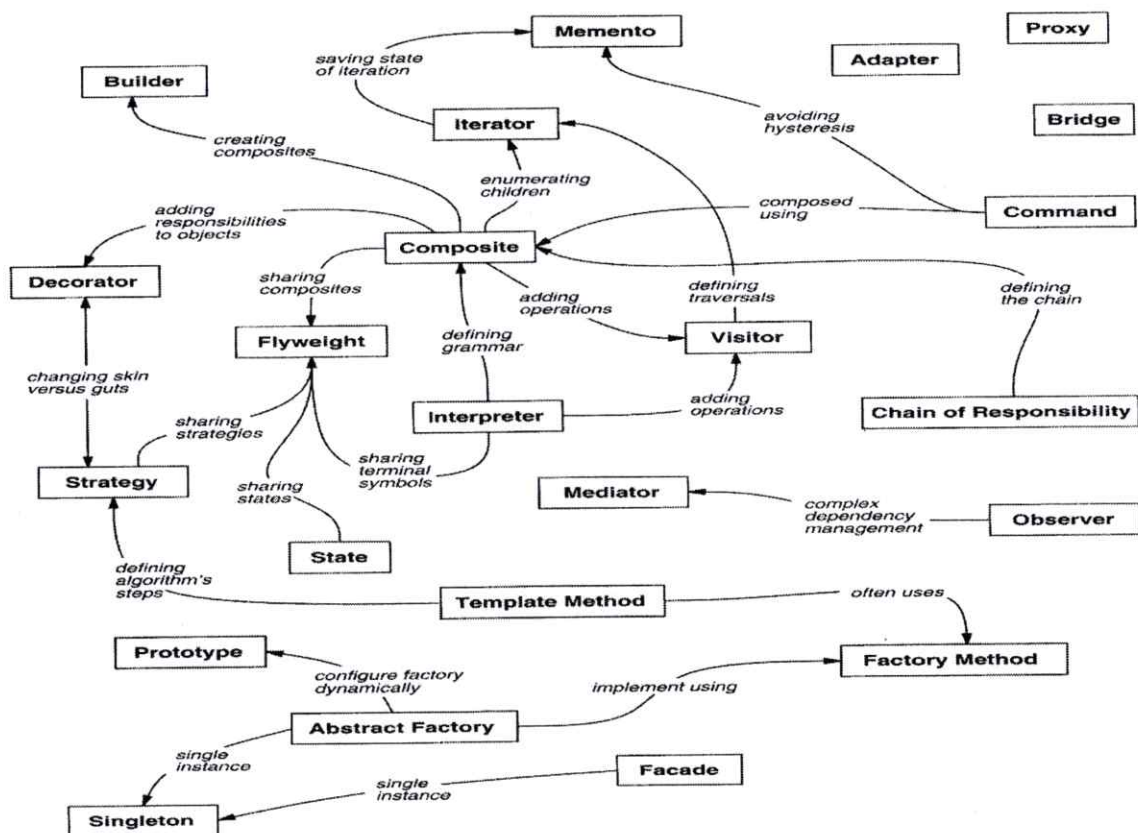


Figure 1.1: Design pattern relationships

- 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
<b>Creational</b>	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
<b>Structural</b>	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
<b>Behavioral</b>	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.

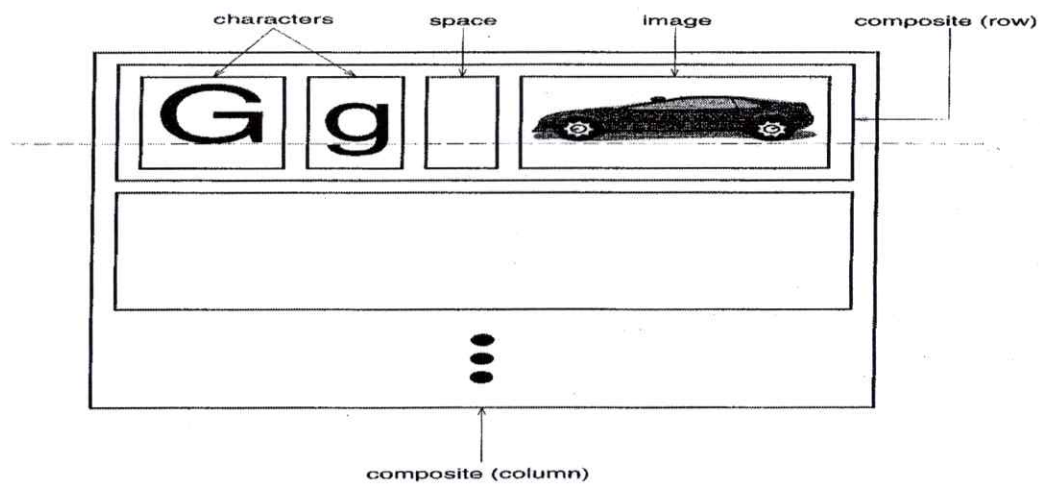


Figure 2.2: Recursive composition of text and graphics

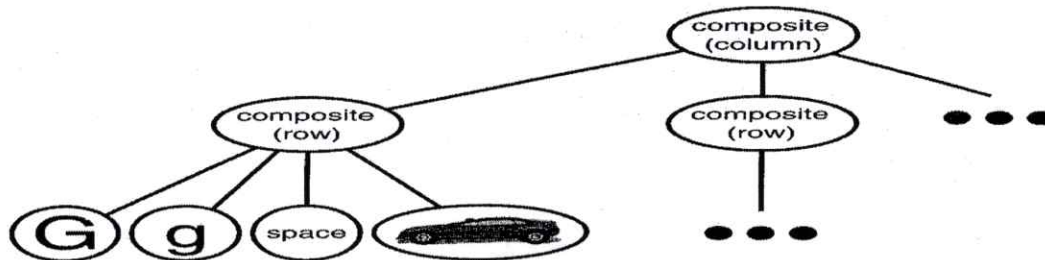


Figure 2.3: Object structure for recursive composition of text and graphics

- **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.

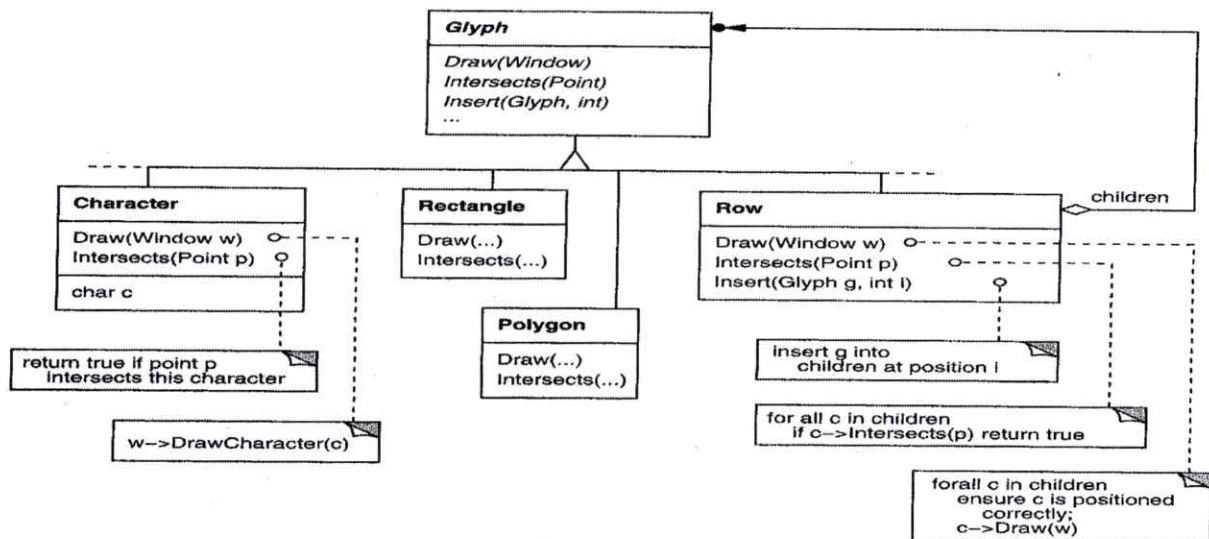


Figure 2.4: Partial Glyph class hierarchy

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	<code>void SetComposition(Composition*)</code>
when to format	<code>virtual void Compose()</code>

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.

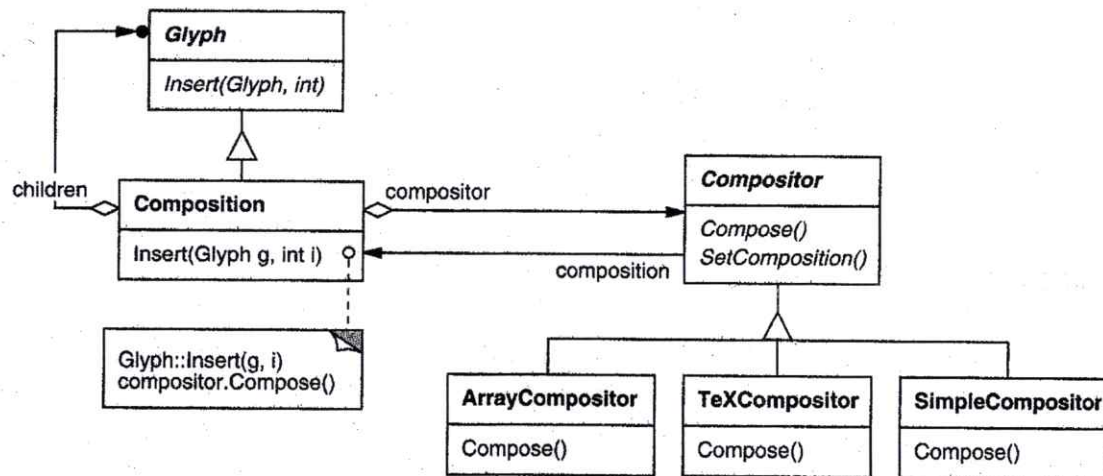


Figure 2.5: Composition and Compositor class relationships

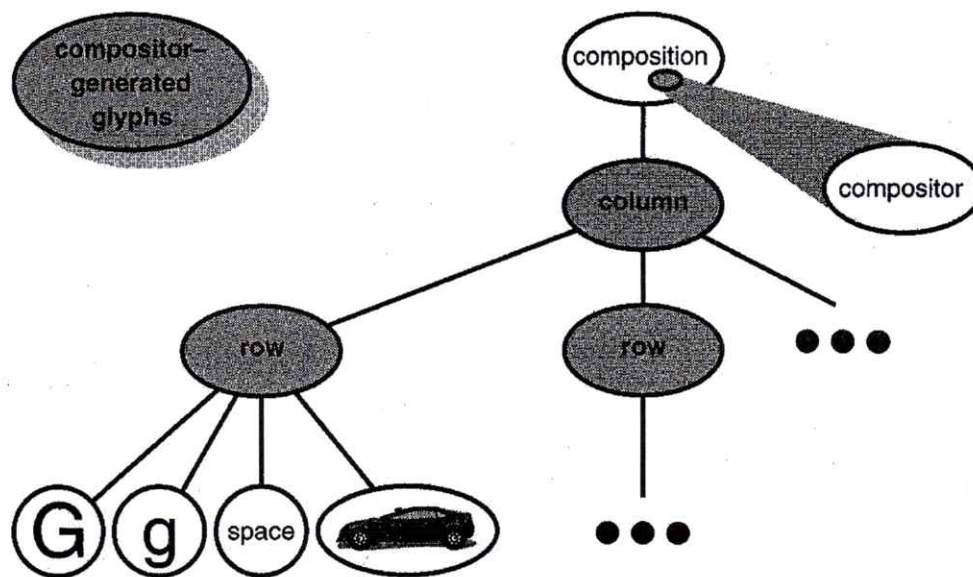


Figure 2.6: Object structure reflecting compositor-directed linebreaking

- **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);  
}
```

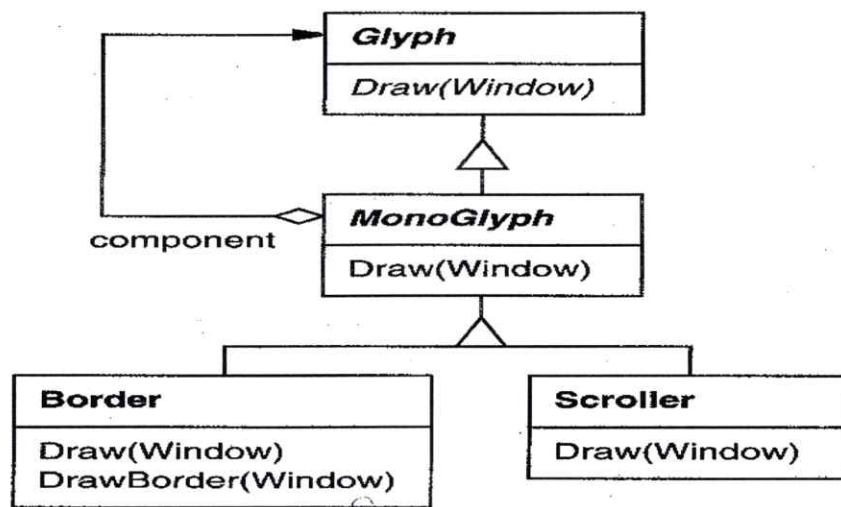


Figure 2.7: MonoGlyph class relationships

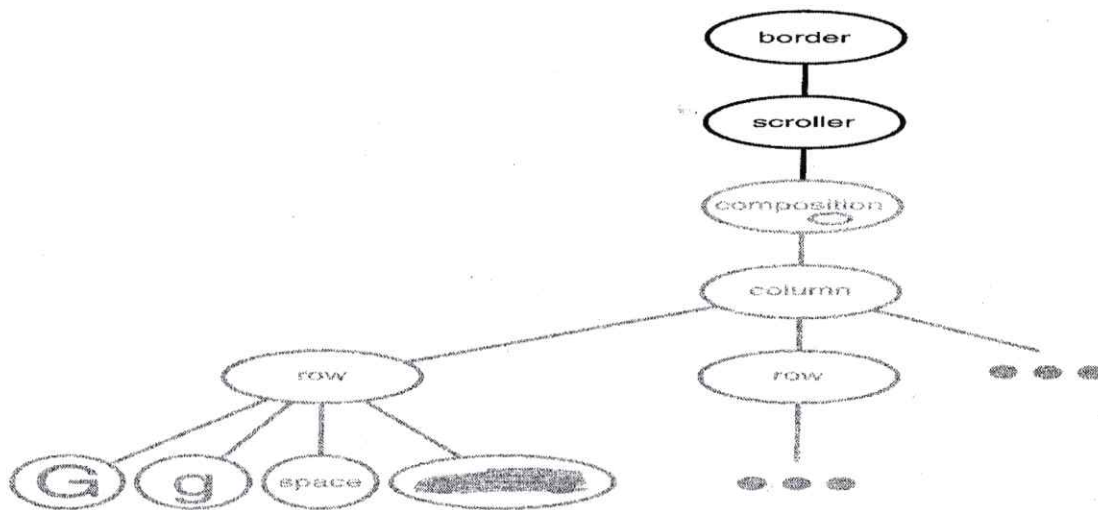
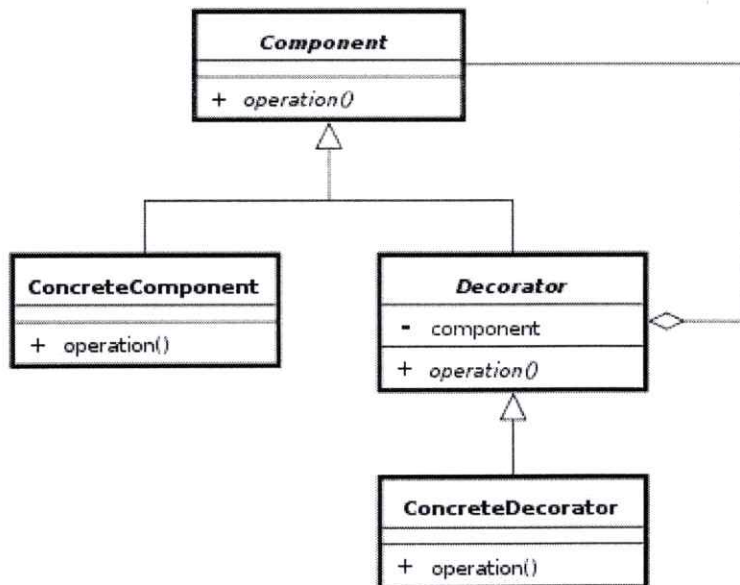


Figure 2.8: Embellished object structure

- **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.

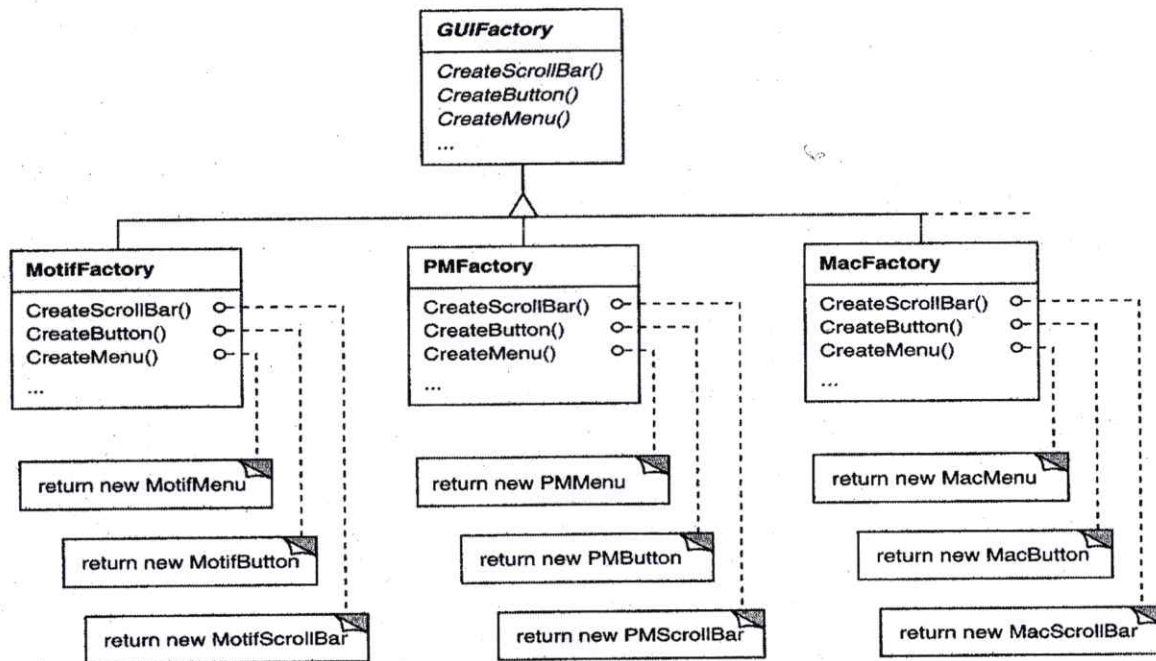


Figure 2.9: GUIFactory class hierarchy

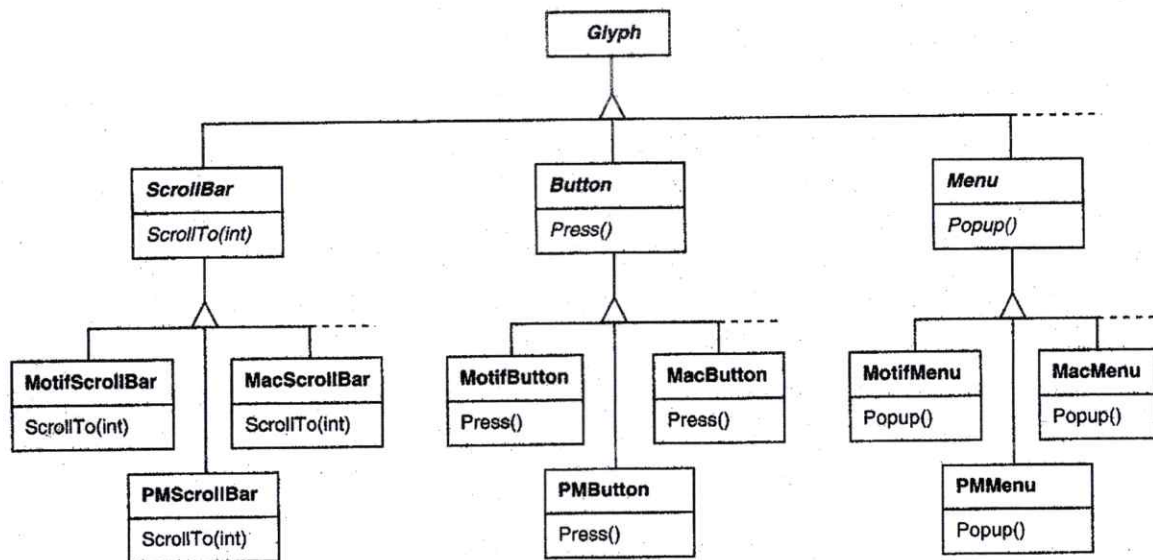


Figure 2.10: Abstract product classes and concrete subclasses

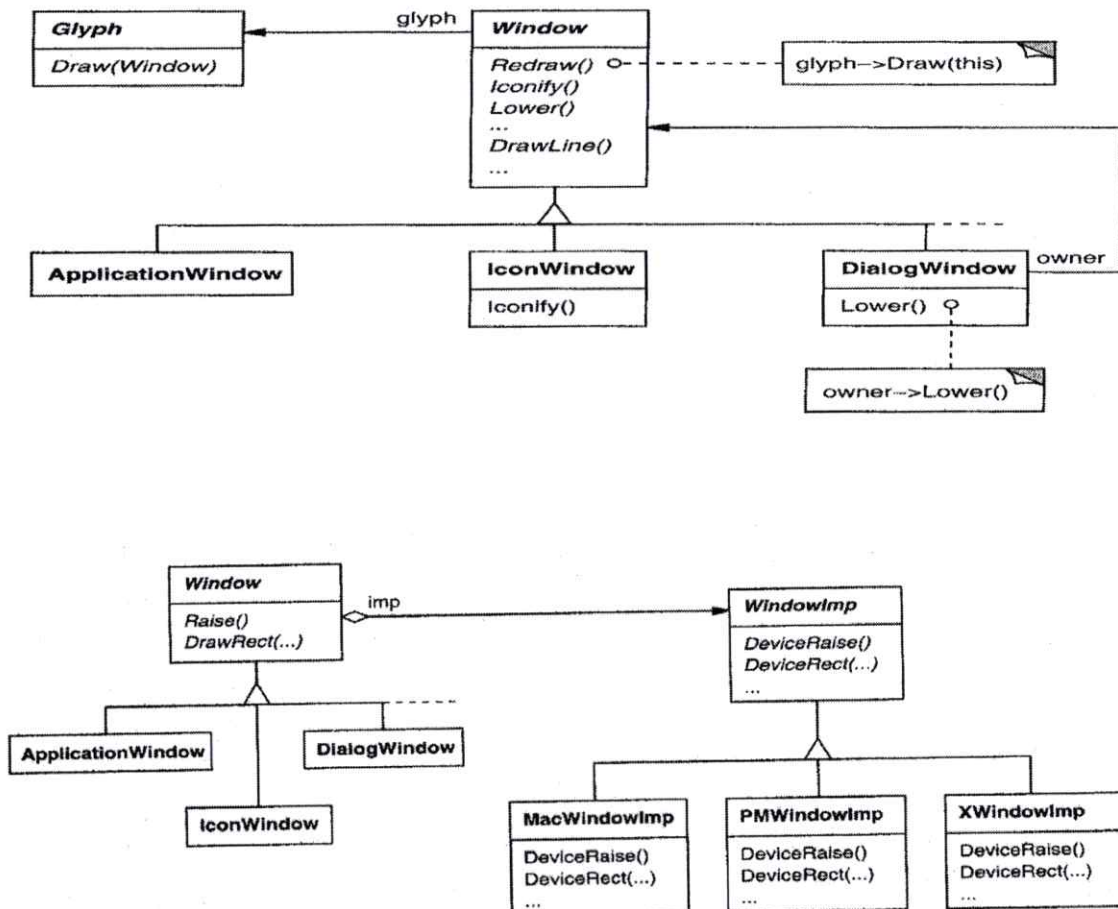
### Supporting Multiple Window Systems:

- 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 implementations 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.



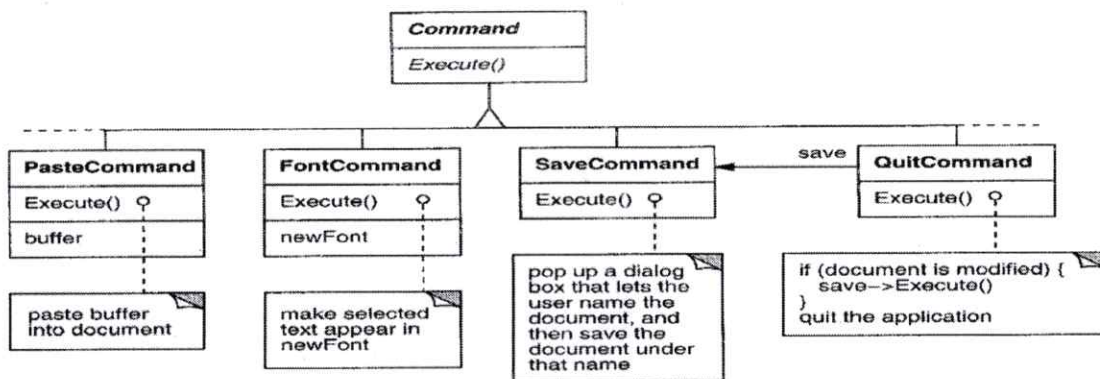


Figure 2.11: Partial Command class hierarchy

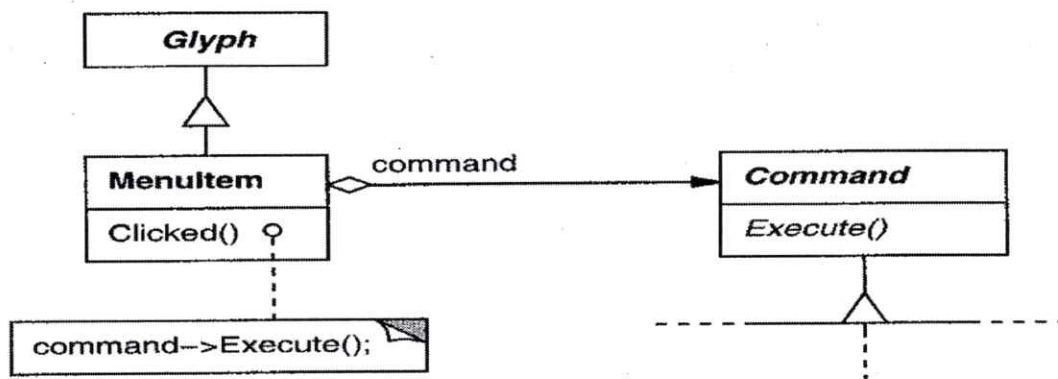


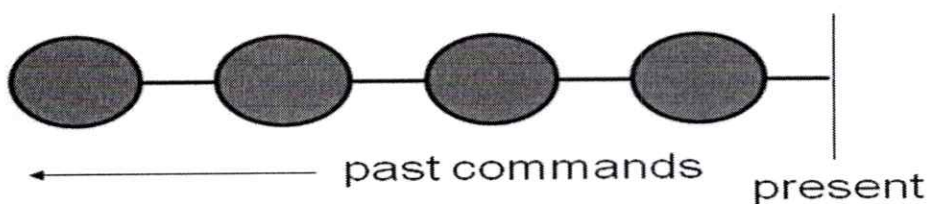
Figure 2.12: MenuItem-Command relationship

- 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.

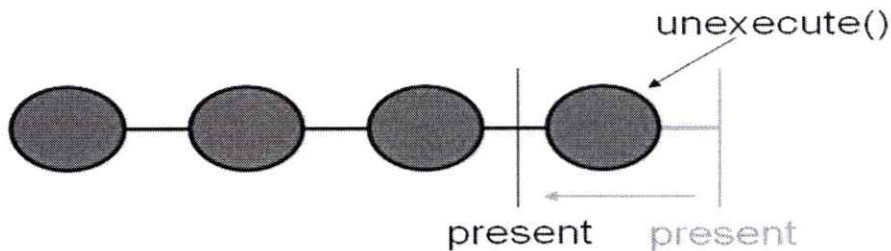
### Command History

- a list of commands that have been executed.



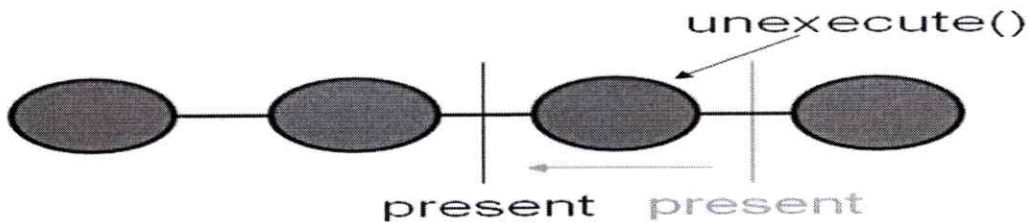
- The command history can be seen as a list of past commands.
- As new commands are executed they are added to the front of the history.

#### 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 Pattern lets you
  - parameterize clients with different requests
  - queue or log requests
  - support undoable operations
- Also Known As: Action, Transaction.

## Spelling Checking & Hyphenation:

Goals:

- analyze text for spelling errors.
- introduce potential hyphenation sites.

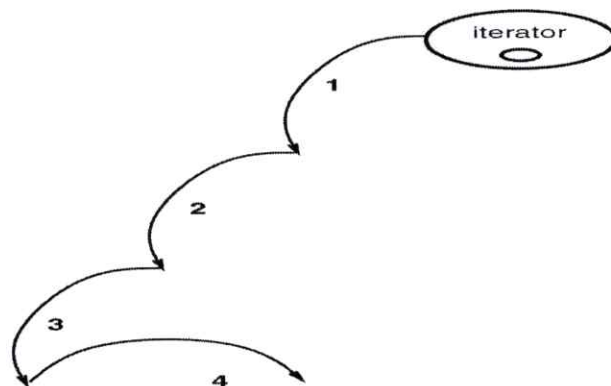
Constraints/forces:

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

**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".



## ITERATOR

object behavioral

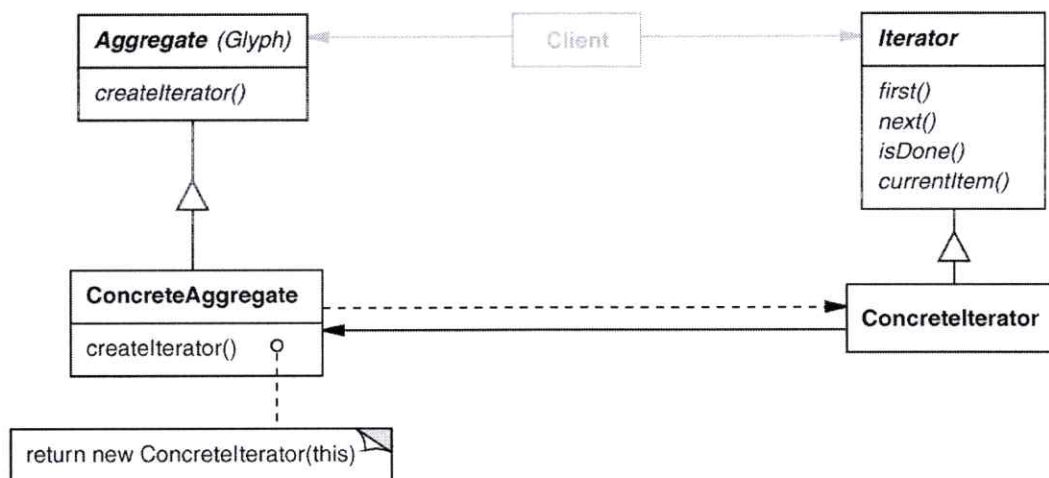
### Intent

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:



## 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.

## 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
};

```

### **SpellingCheckerVisitor :**

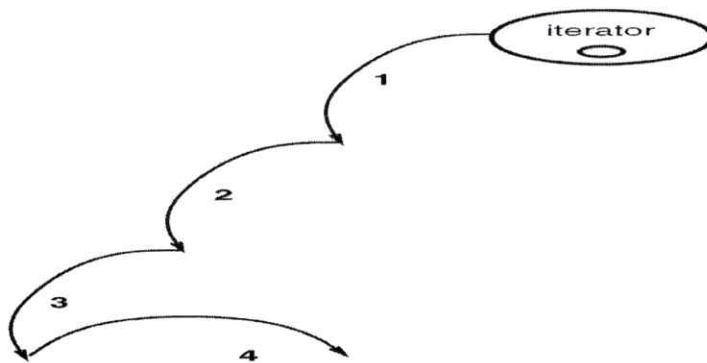
- gets character code from each character glyph.  
Can define getCode() operation just on Character() class
- checks words accumulated from character glyphs.
- combine with PreorderIterator .

```

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_;
};

```

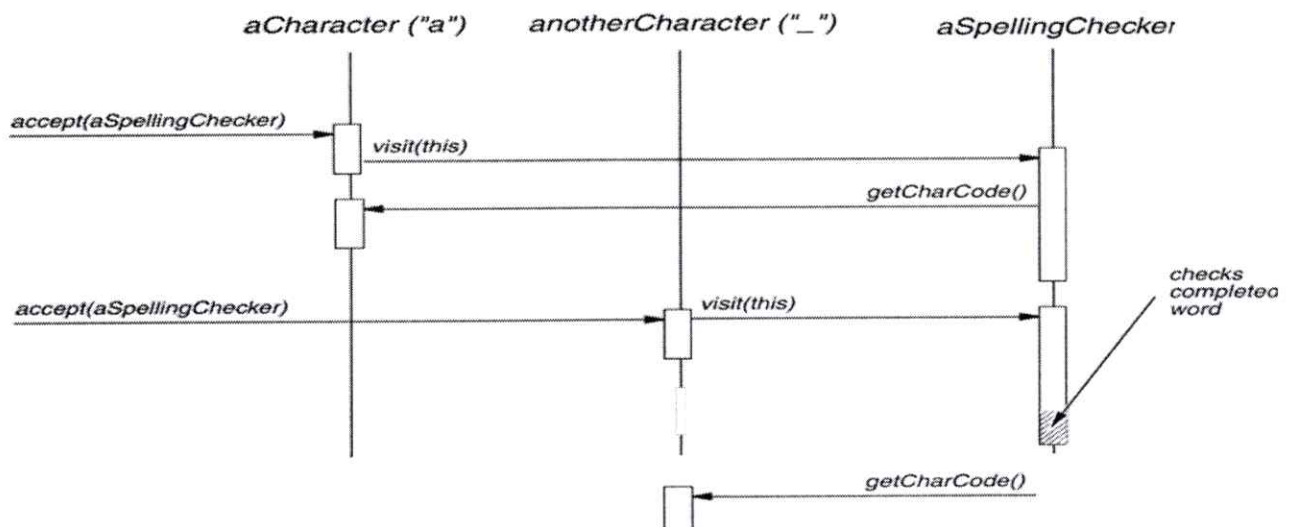
## Accumulating Words:



Spelling check performed when a nonalphabetic character is reached.

## 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.



### **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  
};
```

### **Concluding Remarks:**

- design reuse.
- uniform design vocabulary.
- understanding, restructuring, & team communication.
- provides the basis for automation.
- a “new” way to think about design.

## **Creational Patterns :**

- Abstracts instantiation process
- Makes system independent of how its objects are—
  - created
  - composed
  - represented
- Creational patterns encapsulates knowledge about which concrete classes the system uses
- Hides how instances of these classes are created and put together
- Important if systems evolve to depend more on object composition than on class inheritance
- Emphasis shifts from hardcoding fixed sets of behaviors towards a smaller set of composable fundamental behaviors
- Encapsulate knowledge about concrete classes a system— uses
- Hide how instances of classes are created and put together

## **What are creational patterns?**

- Design patterns that deal with object creation— mechanisms, trying to create objects in a manner suitable to the situation
- Make a system independent of the way in which— objects are created, composed and represented

## **Recurring themes :**

- Encapsulate knowledge about which concrete classes the system uses (so we can change them easily later)
- Hide how instances of these classes are created and put together (so we can change it easily later)

## **Benefits of creational patterns :**

Creational patterns let you program to an interface defined by an abstract class that lets you configure a system with “product” objects that vary widely in structure and functionality



**Example:**

GUI systems.

Interviews GUI class library.

Multiple look-and-feels.

Abstract Factories for different screen components.

**Generic instantiation** – Objects are instantiated→ without having to identify a specific class type in client code (Abstract Factory, Factory) .

**Simplicity** – Make instantiation easier: callers do not→ have to write long complex code to instantiate and set up an object (Builder, Prototype pattern).

**Creation constraints** – Creational patterns can put→ bounds on who can create objects, how they are created, and when they are created .

**Abstract Factory Pattern**

Abstract factory provide an interface for creating families of related or dependent objects without specifying their concrete classes

- Intent:
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes

**Also Known As:** Kit.

**Motivation:**

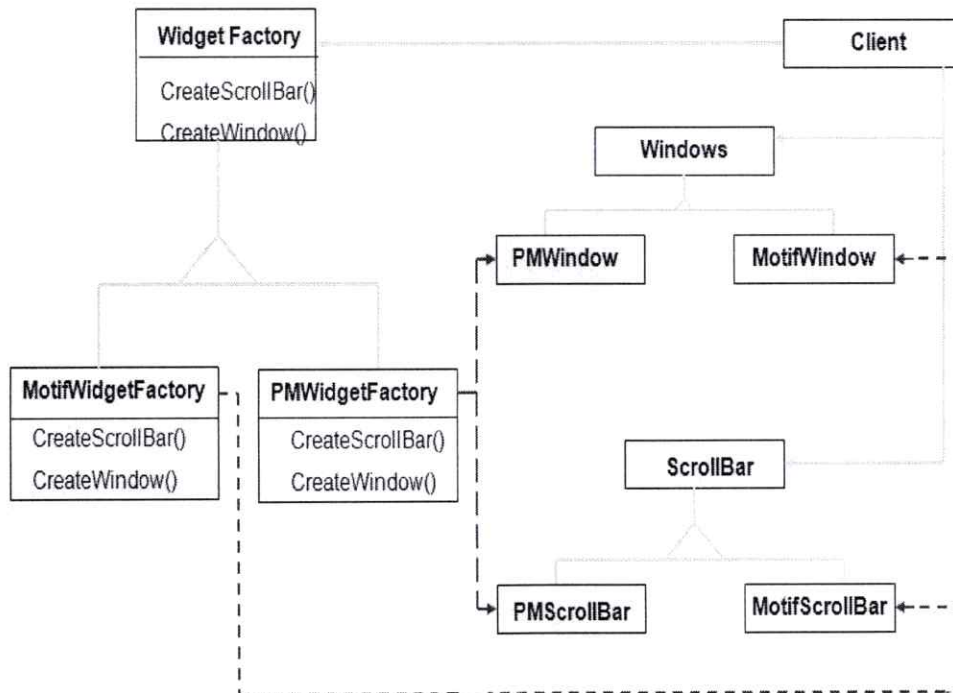
User interface toolkit supports multiple look-and-feel standards (Motif, Presentation Manager).

Different appearances and behaviors for UI widgets

Apps should not hard-code its widgets

# ABSTRACT FACTORY

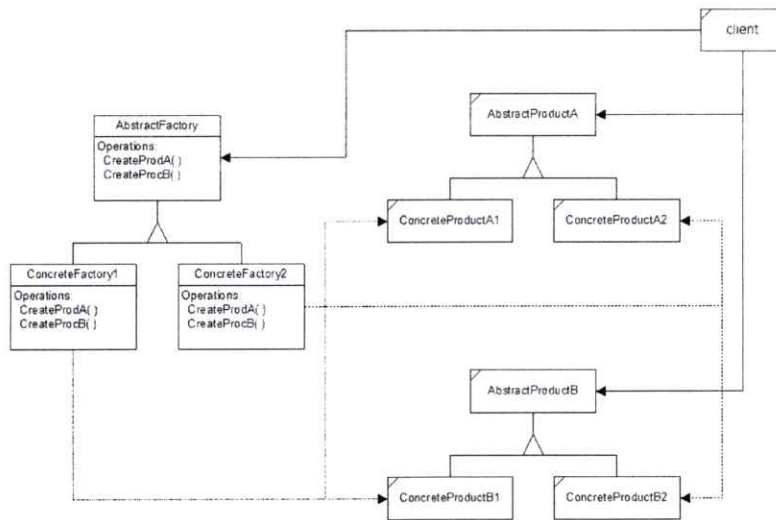
## Motivation



### Solution:

- Abstract Widget Factory class
- Interfaces for creating each basic kind of widget
- Abstract class for each kind of widgets,
- Concrete classes implement specific look-and-feel.

# Abstract Factory Structure



## Abstract Factory :

Declares interface for operations that create abstract product objects

## Concrete Factory :

- Implements operations to create concrete product objects

## Abstract Product :

- Declares an interface for a type of product object.

## Concrete Product:

- Defines a product object to be created by concrete factory
- Implements the abstract product interface

## Client:

- Uses only interfaces declared by Abstract Factory and AbstractProduct classes.

## **Collaborators :**

- Usually only one ConcreteFactory instance is used for an activation, matched to a specific application context. It builds a specific product family for client use -- the client doesn't care which family is used -- it simply needs the services appropriate for the current context.
- The client may use the AbstractFactory interface to initiate creation, or some other agent may use the AbstractFactory on the client's behalf.

## **Presentation Remark :**

- Here, we often use a sequence diagram (event-trace) to show the dynamic interactions between participants.
- For the Abstract Factory Pattern, the dynamic interaction is simple, and a sequence diagram would not add much new information.

## **Consequences :**

- The Abstract Factory Pattern has the following benefits:
  - It isolates concrete classes from the client.
    - You use the Abstract Factory to control the classes of objects the client creates.
    - Product names are isolated in the implementation of the ConcreteFactory, clients use the instances through their abstract interfaces.
  - Exchanging product families is easy.
    - None of the client code breaks because the abstract interfaces don't change.
    - Because the abstract factory creates a complete family of products, the whole product family changes when the concrete factory is changed.



- It promotes consistency among products.
  - It is the concrete factory's job to make sure that the right products are used together.

### **More benefits of the Abstract Factory Pattern**

- It supports the imposition of constraints on product families, e.g., always use A1 and B1 together, otherwise use A2 and B2 together.
- **The Abstract Factory pattern has the following liability:**
  - Adding new kinds of products to existing factory is difficult.
- Adding a new product requires extending the abstract interface which implies that all of its derived concrete classes also must change.
- Essentially everything must change to support and use the new product family
- abstract factory interface is extended
- derived concrete factories must implement the extensions
- a new abstract product class is added
- a new product implementation is added
- client has to be extended to use the new product

### **Implementation**

- Concrete factories are often implemented as singletons.
- Creating the products
  - Concrete factory usually use the factory method.
    - simple
    - new concrete factory is required for each product family
  - alternately concrete factory can be implemented using prototype.
    - only one is needed for all families of products

- product classes now have special requirements - they participate in the creation
- Defining extensible factories by using create function with an argument
  - only one virtual create function is needed for the AbstractFactory interface
  - all products created by a factory must have the same base class or be able to be safely coerced to a given type
  - it is difficult to implement subclass specific operations

### **Know Uses:-**

- **Interviews**
  - used to generate “look and feel” for specific user interface objects
  - uses the Kit suffix to denote AbstractFactory classes, e.g., WidgetKit and DialogKit.
  - also includes a layoutKit that generates different composite objects depending on the needs of the current context

### **ET++**

- another windowing library that uses the AbstractFactory to achieve portability across different window systems (X Windows and SunView).

### **Related Patterns:-**

- Factory Method -- a “virtual” constructor
- Prototype -- asks products to clone themselves
- Singleton -- allows creation of only a single instance

## **Code Examples:-**

- **Skeleton Example**
  - Abstract Factory Structure
  - Skeleton Code
- **Neural Net Example**
  - Neural Net Physical Structure
  - Neural Net Logical Structure
  - Simulated Neural Net Example

## **BUILDER :-**

- **Intent:**

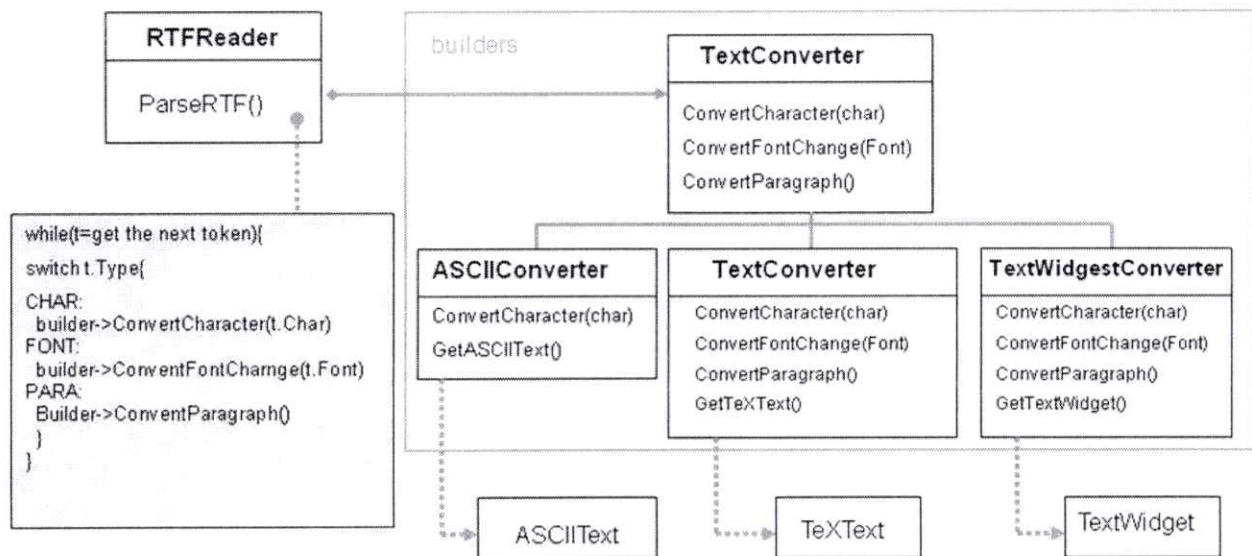
Separate the construction of a complex object from its representation so that the same construction process can create different representations

- Motivation:
  - RTF reader should be able to convert RTF to many text format
  - Adding new conversions without modifying the reader should be easy

- **Solution:**

- Configure RTFReader class with a Text Converter object
- Subclasses of Text Converter specialize in different conversions and formats
- TextWidgetConverter will produce a complex UI object and lets the user see and edit the text

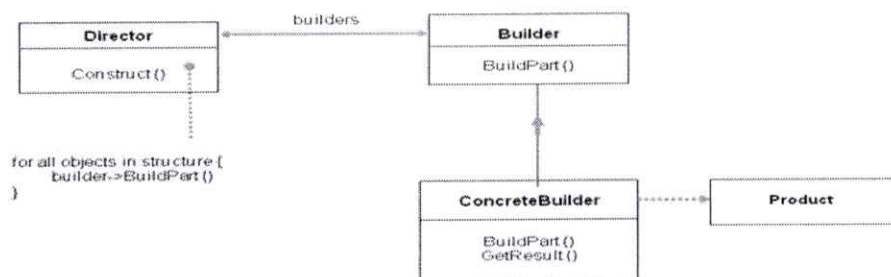
## BUILDER Motivation:-



## Applicability:-

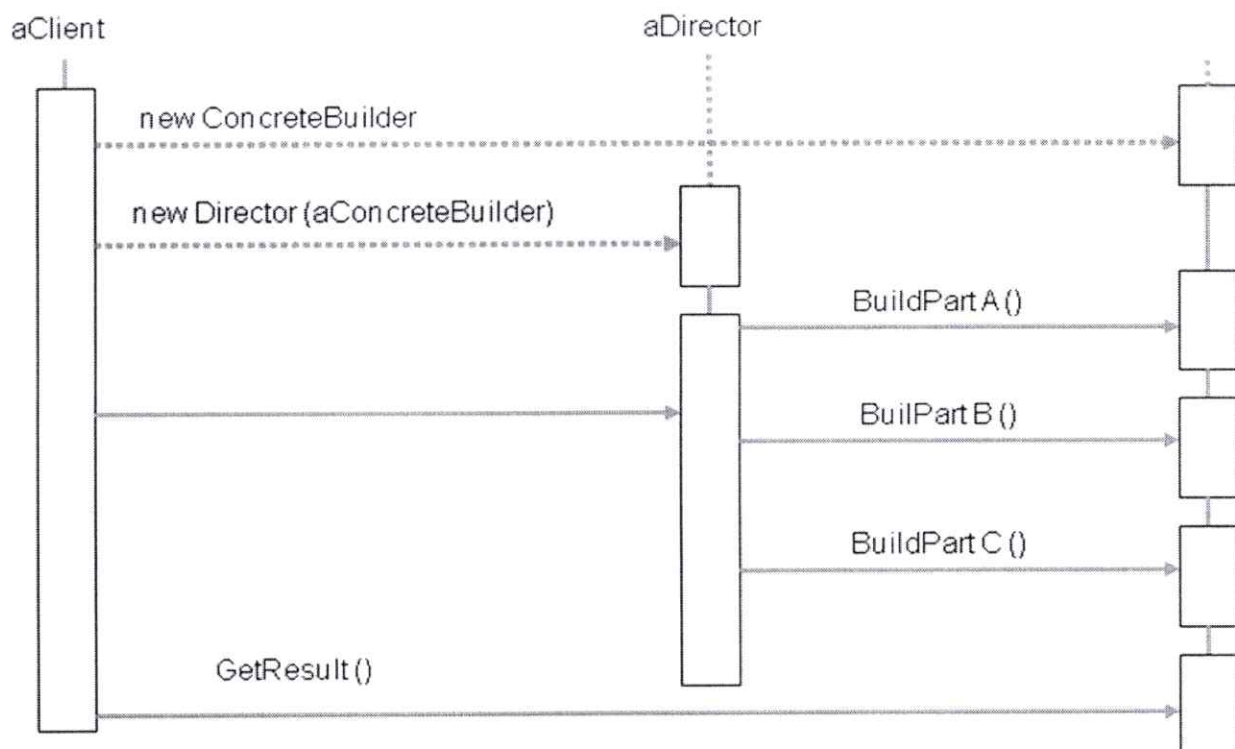
- Use the Builder pattern when
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled
  - The construction process must allow different representations for the object that is constructed

## BUILDER Structure:-



### Builder – Collaborations:-

- Client creates Director object and configures it with the desired Builder object
- Director notifies Builder whenever a part of the product should be built
- Builder handles requests from the Director and adds parts to the product
- Client retrieves the product from the Builder



**Why do we use Builder?**



- Common manner to Create an Instance

- **Constructor!**
- Each Parts determined by Parameter of the Constructor

```
public class Room {  
    private int area;  
    private int windows;  
    public String purpose;  
}
```

```
Room() {  
}  
  
Room(int newArea, int newWindows, String newPurpose){  
    area = newArea;  
    windows = newWindows;  
    purpose = newPurpose;  
}
```

There are Only 2 different ways to Create an Instance part-by-part.

- In the previous example,
  - We can either determine all the arguments or determine nothing and just construct.  
We can't determine arguments partially.
  - We can't control whole process to Create an instance.
  - Restriction of ways to Create an Object
  - Bad Abstraction & Flexibility

### Discussion:-

- Uses Of Builder
  - Parsing Program(RTF converter)
  - GUI

## FACTORY METHOD (Class Creational):-

- **Intent:**

- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Factory Method lets a class defer instantiation to subclasses.

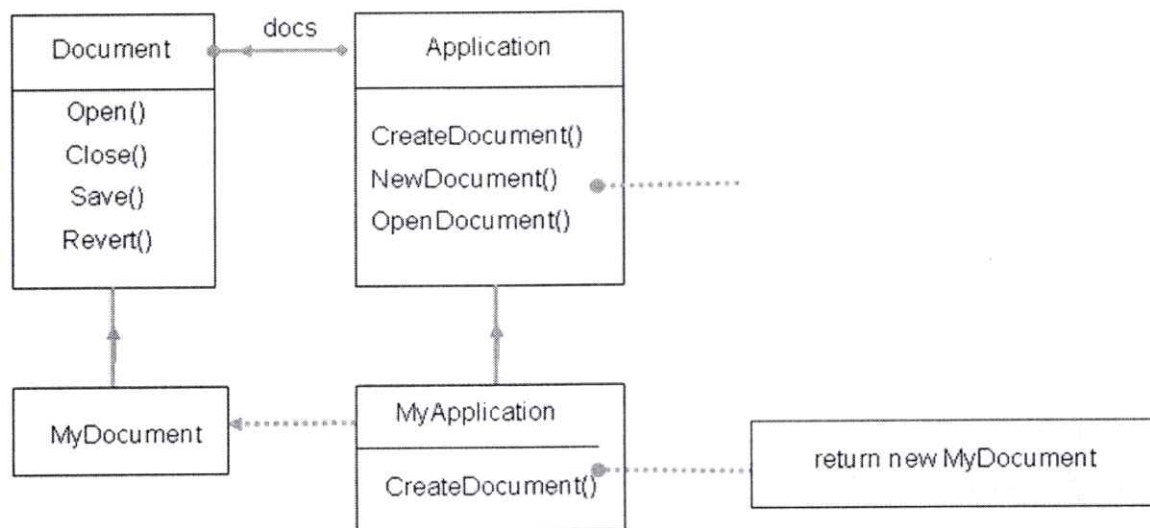
- **Motivation:**

- Framework use abstract classes to define and maintain relationships between objects
- Framework has to create objects as well - must instantiate classes but only knows about abstract classes - which it cannot instantiate

### Motivation:-

- Motivation: Factory method encapsulates knowledge of which subclass to create - moves this knowledge out of the framework
- Also Known As: Virtual Constructor

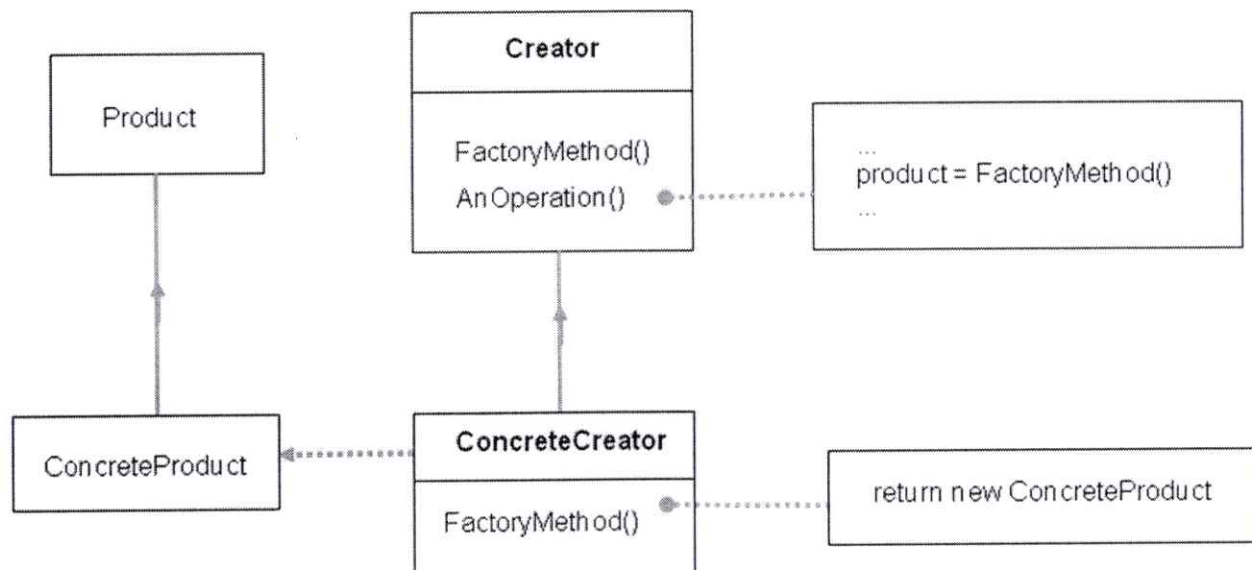
### FACTORY METHOD Motivation:-



## Applicability:-

- Use the Factory Method pattern when
  - a class can't anticipate the class of objects it must create.
  - a class wants its subclasses to specify the objects it creates.
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

## FACTORY METHOD Structure:-



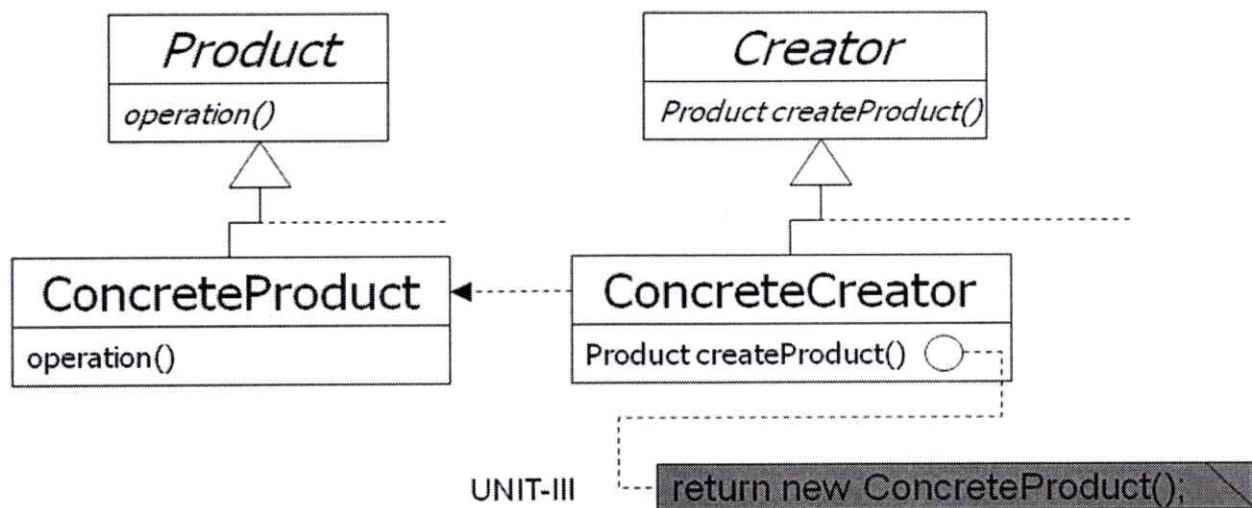
## Participants:-

- **Product**
  - Defines the interface of objects the factory method creates
- **ConcreteProduct**
  - Implements the product interface
- **Creator**
  - Declares the factory method which returns object of type product

- May contain a default implementation of the factory method
- Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate Concrete Product.
- ConcreteCreator
  - Overrides factory method to return instance of ConcreteProduct

### Factory Method:-

- Defer object instantiation to subclasses
- Eliminates binding of application-specific subclasses
- Connects parallel class hierarchies
- A related pattern is AbstractFactory



## PROTOTYPE (Object Creational):-

- **Intent:**

- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

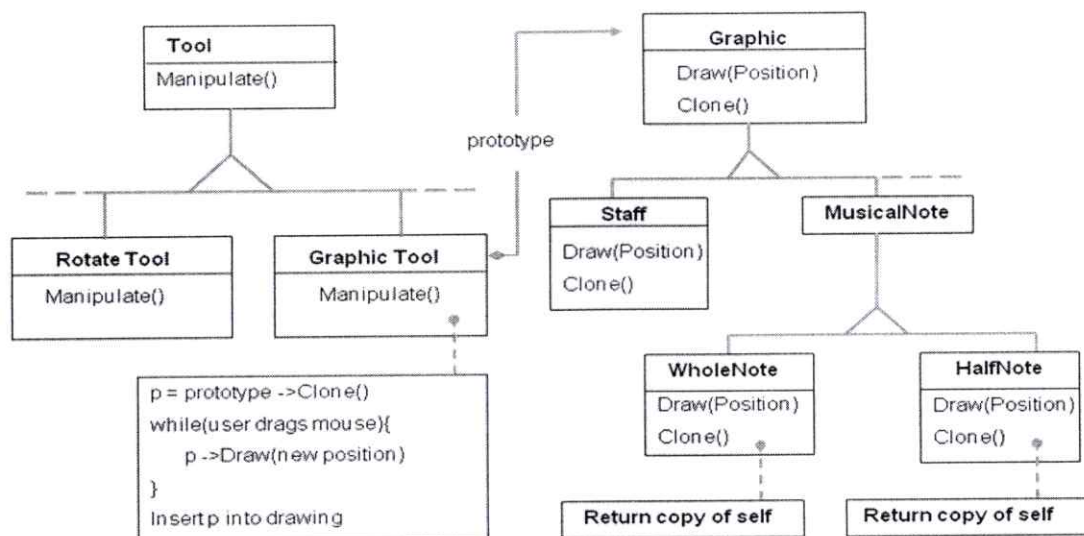
- **Motivation:**

- Framework implements Graphic class for graphical components and GraphicTool class for tools manipulating/creating those components

### Motivation:-

- Actual graphical components are application-specific
- How to parameterize instances of Graphic Tool class with type of objects to create?
- Solution: create new objects in Graphic Tool by cloning a **prototype** object instance

### PROTOTYPE Motivation:-

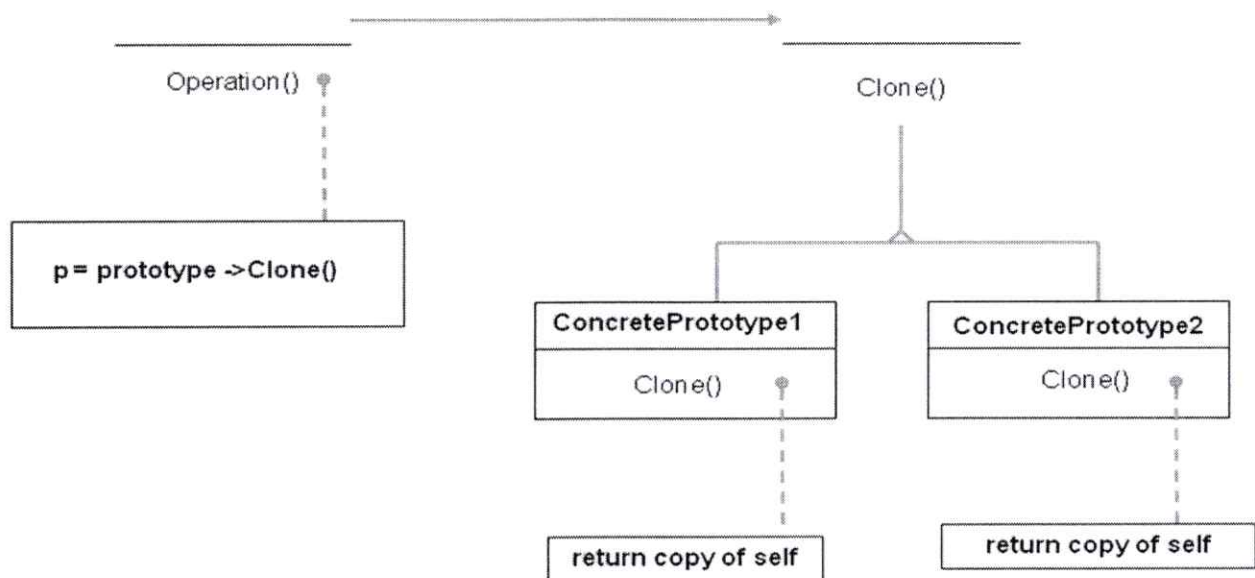




## Applicability:-

- Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented;
  - when the classes to instantiate are specified at run-time, for example, by dynamic loading; or
  - to avoid building a class hierarchy of factories that parallels the class hierarchy of products; or when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

## PROTOTYPE Structure:-



## **Participants:**

- Prototype (Graphic)
  - Declares an interface for cloning itself
- ConcretePrototype (Staff, WholeNote, HalfNote)
  - Implements an interface for cloning itself
- Client (GraphicTool)
  - Creates a new object by asking a prototype to clone itself

## **Collaborations:**

- A client asks a prototype to clone Itself.

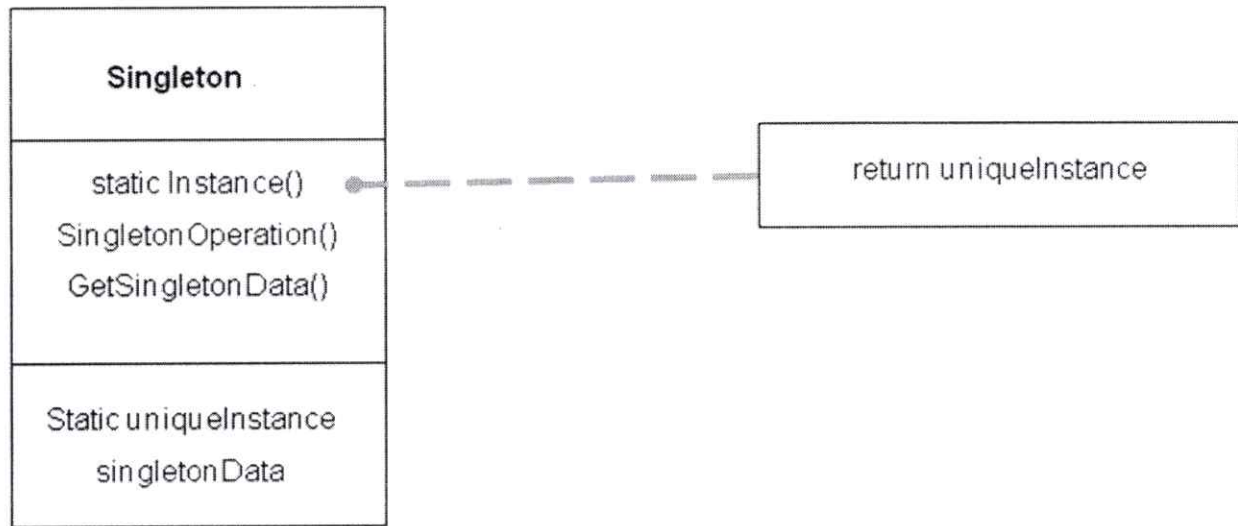
## **SINGELTON:-**

- Intent:
  - Ensure a class only has one instance, and provide a global point of access to it.
- Motivation:
  - Some classes should have exactly one instance  
(one print spooler, one file system, one window manager)
  - A global variable makes an object accessible but doesn't prohibit instantiation of multiple objects
  - Class should be responsible for keeping track of its sole interface

## **Applicability:-**

- Use the Singleton pattern when
  - there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
  - when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

## SINGLETON Structure:-



## Participants and Collaborations:-

- Singleton:
- Defines an instance operation that lets clients access its unique interface
- Instance is a class operation (static in Java)
- May be responsible for creating its own unique instance
- Collaborations:
- Clients access a Singleton instance solely through Singleton's Instance operation.

## Singleton:-

- Ensures a class has only one instance
- Provides a single point of reference

### **Singleton – Use When:-**

- There must be exactly one instance of a class.
- May provide synchronous access to avoid deadlocks.
- Very common in GUI toolkits, to specify the connection to the OS/Windowing system

### **Singleton – Benefits:-**

- Controls access to a scarce or unique resource
- Helps avoid a central application class with various global object references
- Subclasses can have different implementations as required. Static or global references don't allow this
- Multiple or single instances can be allowed

### **Singleton – Example 1:-**

- An Application class, where instantiating it makes a connection to the base operating system and sets up the rest of the toolkit's framework for the user interface.
- In the Qt toolkit:

```
QApplication* app = new QApplication(argc, argv)
```

### **Singleton – Example 2:-**

- A status bar is required for the application, and various application pieces need to be able to update the text to display information to the user. However, there is only one status bar, and the interface to it should be limited. It could be implemented as a Singleton object, allowing only one instance and a focal point for updates. This would allow updates to be queued, and prevent messages from being overwritten too quickly for the user to read them.

## Singleton Code [1]:-

```
class Singleton {
```

```
    public:
```

```
        static Singleton* Instance();
```

```
    protected:
```

```
        Singleton();
```

```
    private:
```

```
        Static Singleton* _instance
```

```
}
```



```
// Cannot access directly.
```



### Singleton Code [2]:-

```
Singleton* Singleton::_instance=0;

Singleton* Singleton::Instance(){
    if (_instance ==0) {
        _instance=new Singleton;
    }
    Return _instance;
}

if (_instance ==0) {
    _instance=new Singleton;
}

Return _instance;

// Clients access the singleton
// exclusively via the Instance member
// function.
}
```

### Implementation Points:-

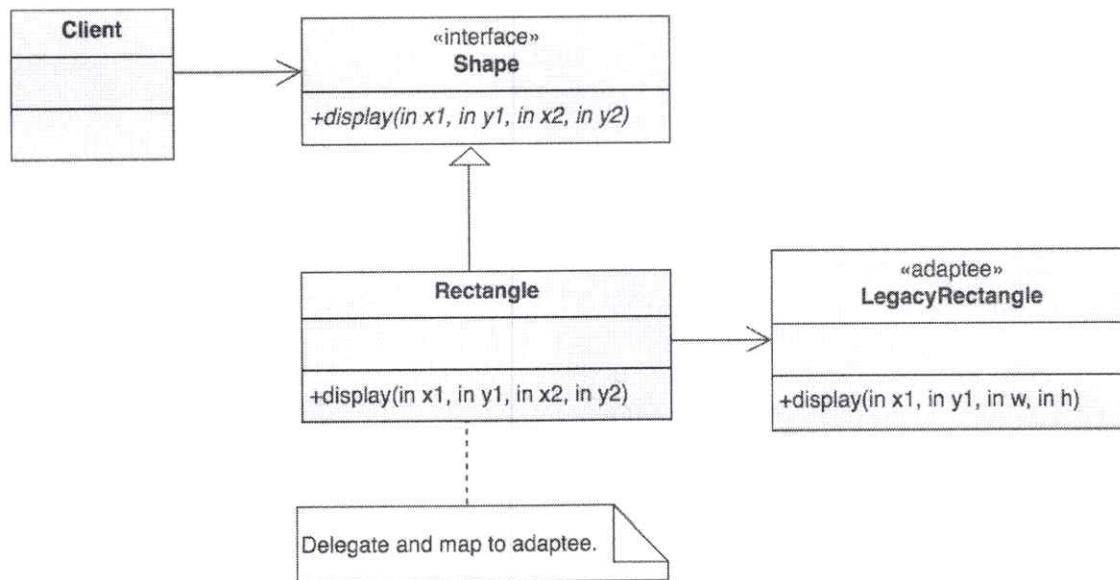
- Generally, a single instance is held by the object, and controlled by a single interface.
- Sub classing the Singleton may provide both default and overridden functionality.

## UNIT-III

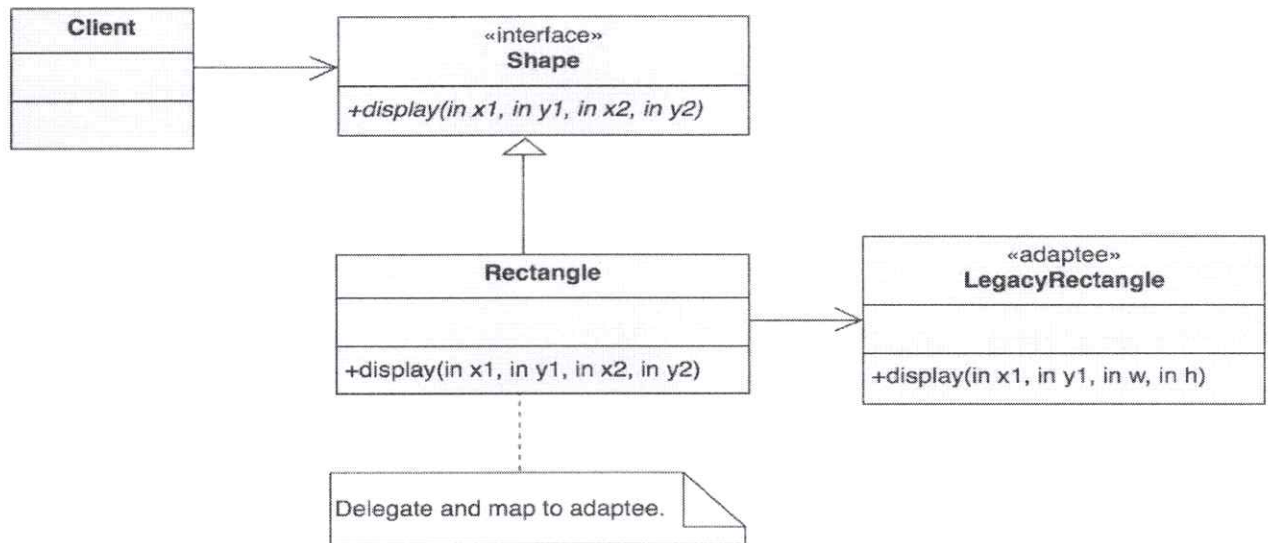
### Structural Pattern Part-I

#### Structural patterns

In Software Engineering, Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities.



- ☐ Adapter  
Match interfaces of different classes
- ☐ Bridge  
Separates an object's interface from its implementation
- ☐ Composite  
A tree structure of simple and composite objects
- ☐ Decorator  
Add responsibilities to objects dynamically
- ☐ Facade  
A single class that represents an entire subsystem
- ☐ Flyweight  
A fine-grained instance used for efficient sharing



Private Class Data  
Restricts accessor/mutator access

Proxy  
An object representing another object

## Rules of thumb

1. Adapter makes things work after they're designed; Bridge makes them work before they are.
2. Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together.
3. Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.
4. Adapter changes an object's interface, Decorator enhances an object's responsibilities. Decorator is thus more transparent to the client. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters.
5. Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.
6. Composite can be traversed with Iterator. Visitor can apply an operation over a Composite. Composite could use Chain of responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition. It could use Observer to tie one object structure to another and State to let a component change its behavior as its state changes.
7. Composite can let you compose a Mediator out of smaller pieces through recursive composition.
8. Decorator lets you change the skin of an object. Strategy lets you change the guts.

9. Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert.
10. Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.
11. Facade defines a new interface, whereas Adapter reuses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one.
12. Facade objects are often Singleton because only one Facade object is required.
13. Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communication between colleague objects, it routinely "adds value", and it is known/referenced by the colleague objects. In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes.
14. Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.
15. Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem.
16. Flyweight is often combined with Composite to implement shared leaf nodes.
17. Flyweight explains when and how State objects can be shared.

## **Adapter Design Patterns**

### **Intent**

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Wrap an existing class with a new interface.
- Impedance match an old component to a new system

### **Problem**

An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.



## Discussion

Reuse has always been painful and elusive. One reason has been the tribulation of designing something new, while reusing something old. There is always something not quite right between the old and the new. It may be physical dimensions or misalignment. It may be timing or synchronization. It may be unfortunate assumptions or competing standards.

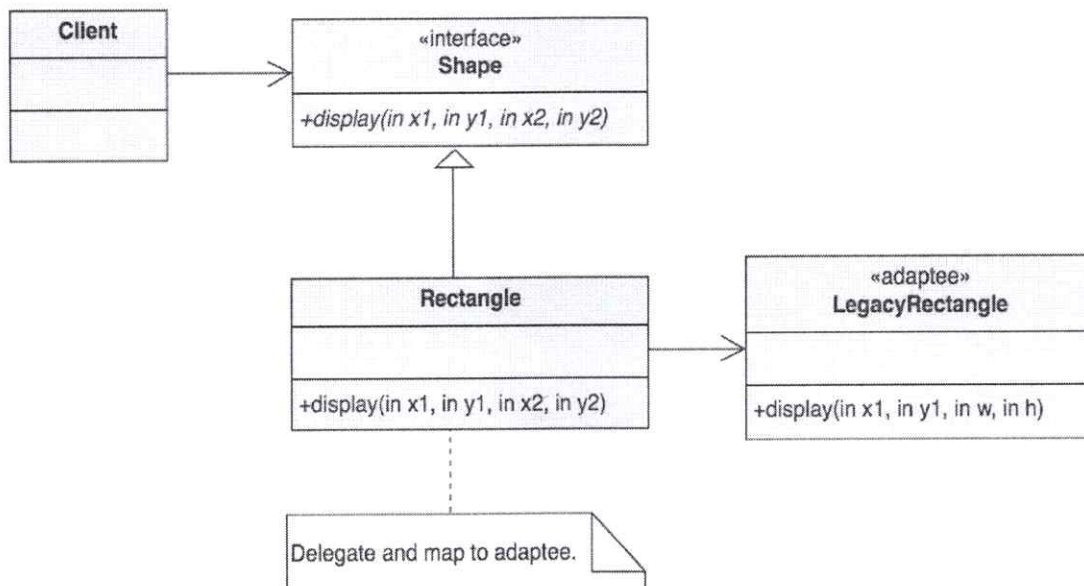
It is like the problem of inserting a new three-prong electrical plug in an old two-prong wall outlet – some kind of adapter or intermediary is necessary.

Adapter is about creating an intermediary abstraction that translates, or maps, the old component to the new system. Clients call methods on the Adapter object which redirects them into calls to the legacy component. This strategy can be implemented either with inheritance or with aggregation.

Adapter functions as a wrapper or modifier of an existing class. It provides a different or translated view of that class.

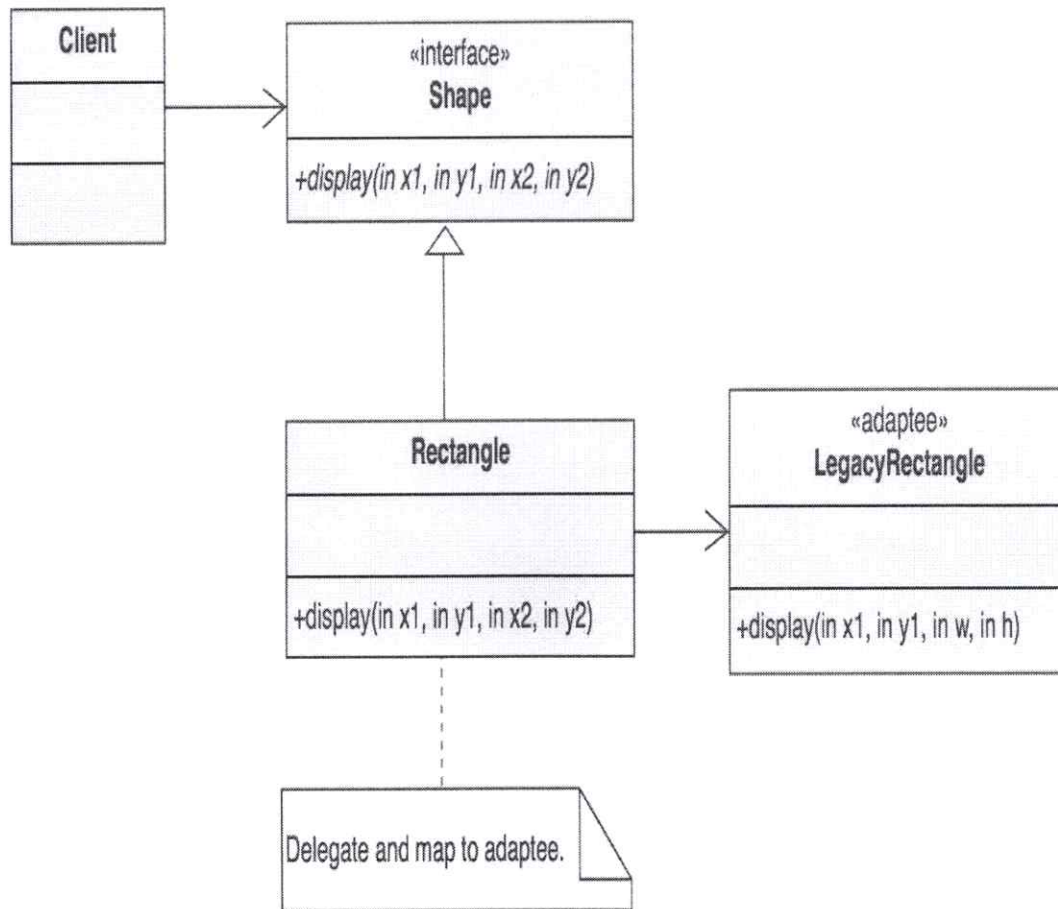
## Structure

Below, a legacy Rectangle component's display() method expects to receive "x, y, w, h" parameters. But the client wants to pass "upper left x and y" and "lower right x and y". This incongruity can be reconciled by adding an additional level of indirection – i.e. an Adapter object.



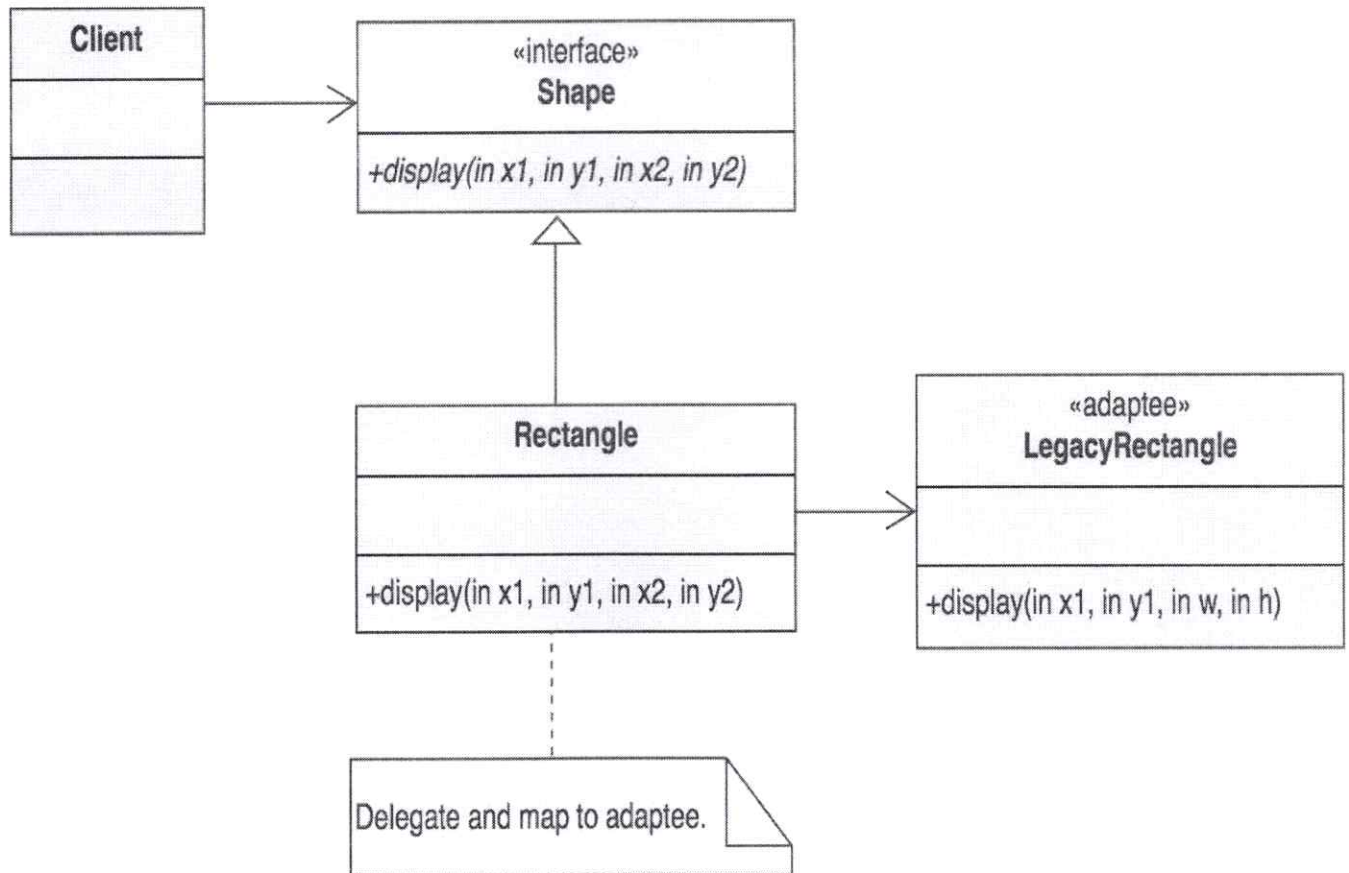
The Adapter could also be thought of as a "wrapper".





## Example

The Adapter pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients. Socket wrenches provide an example of the Adapter. A socket attaches to a ratchet, provided that the size of the drive is the same. Typical drive sizes in the United States are 1/2" and 1/4". Obviously, a 1/2" drive ratchet will not fit into a 1/4" drive socket unless an adapter is used. A 1/2" to 1/4" adapter has a 1/2" female connection to fit on the 1/2" drive ratchet, and a 1/4" male connection to fit in the 1/4" drive socket.



### Check list

1. Identify the players: the component(s) that want to be accommodated (i.e. the client), and the component that needs to adapt (i.e. the adaptee).
2. Identify the interface that the client requires.
3. Design a "wrapper" class that can "impedance match" the adaptee to the client.
4. The adapter/wrapper class "has a" instance of the adaptee class.
5. The adapter/wrapper class "maps" the client interface to the adaptee interface.
6. The client uses (is coupled to) the new interface

## Rules of thumb

- Adapter makes things work after they're designed; Bridge makes them work before they are.
- Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together.
- Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.
- Adapter is meant to change the interface of an existing object. Decorator enhances another object without changing its interface. Decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters.
- Facade defines a new interface, whereas Adapter reuses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one.

## Bridge Design Pattern

### Intent

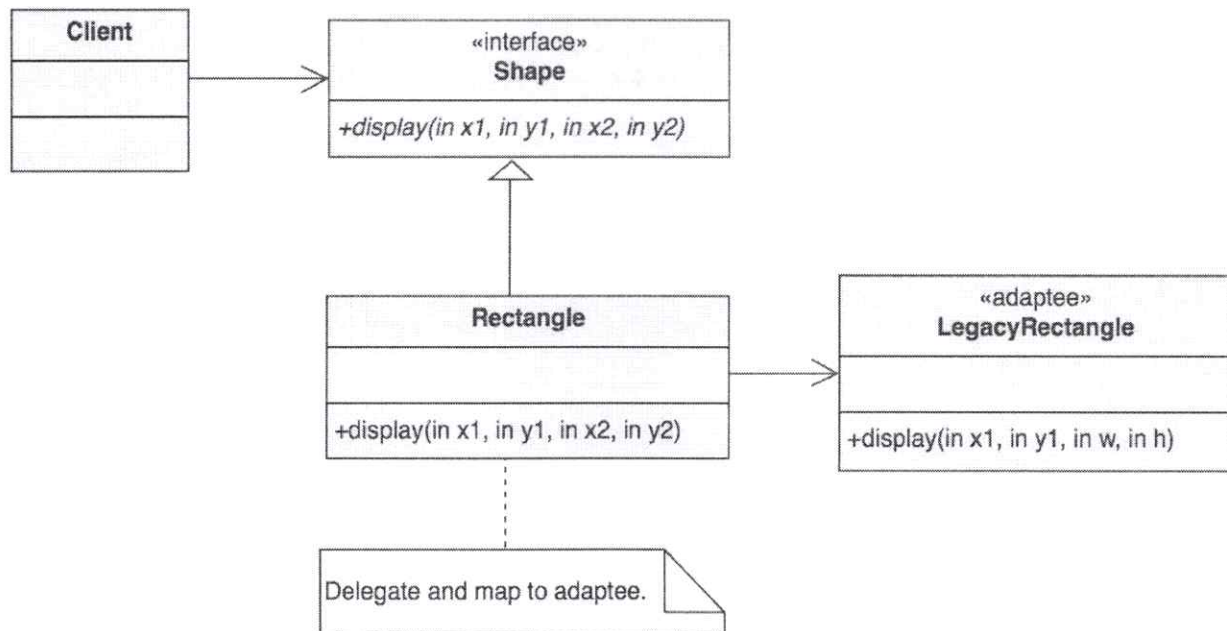
- Decouple an abstraction from its implementation so that the two can vary independently.
- Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.
- Beyond encapsulation, to insulation

### Problem

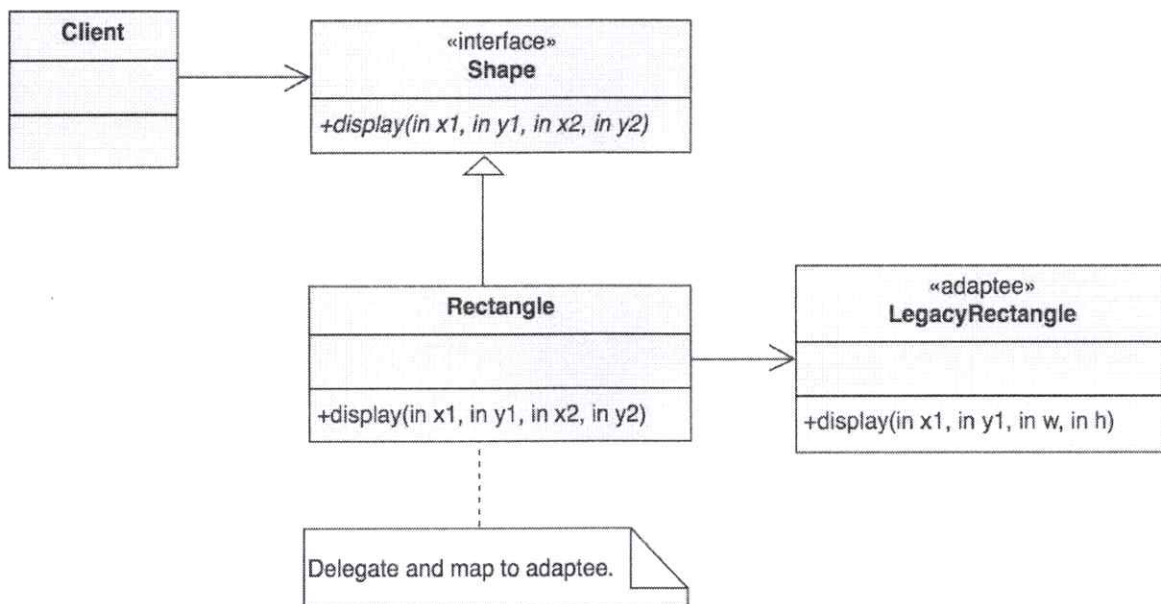
"Hardening of the software arteries" has occurred by using subclassing of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be independently extended or composed.

## Motivation

Consider the domain of "thread scheduling".



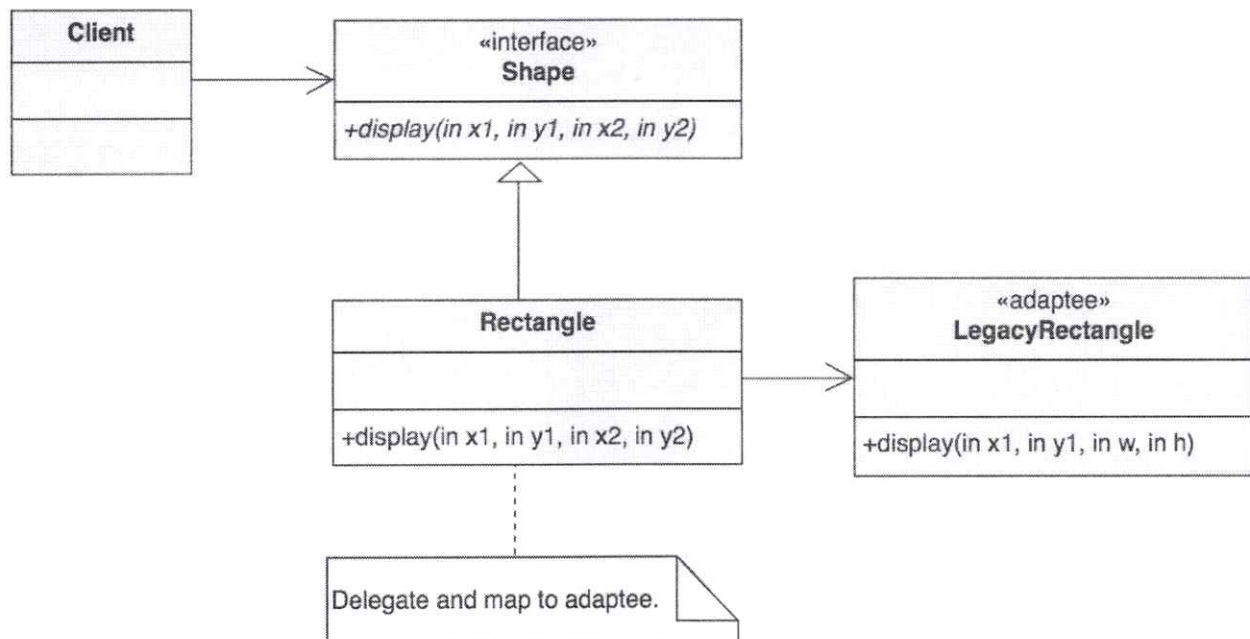
There are two types of thread schedulers, and two types of operating systems or "platforms". Given this approach to specialization, we have to define a class for each permutation of these two dimensions. If we add a new platform (say ... Java's Virtual Machine), what would our hierarchy look like?



What if we had three kinds of thread schedulers, and four kinds of platforms? What if we had five kinds of thread schedulers, and ten kinds of platforms? The number of classes we would have to define is the product of the number of scheduling schemes and the number of platforms.



The Bridge design pattern proposes refactoring this exponentially explosive inheritance hierarchy into two orthogonal hierarchies – one for platform-independent abstractions, and the other for platform-dependent implementations.



## Discussion

Decompose the component's interface and implementation into orthogonal class hierarchies. The interface class contains a pointer to the abstract implementation class. This pointer is initialized with an instance of a concrete implementation class, but all subsequent interaction from the interface class to the implementation class is limited to the abstraction maintained in the implementation base class. The client interacts with the interface class, and it in turn "delegates" all requests to the implementation class.

The interface object is the "handle" known and used by the client; while the implementation object, or "body", is safely encapsulated to ensure that it may continue to evolve, or be entirely replaced (or shared at run-time).

Use the Bridge pattern when:

- you want run-time binding of the implementation,
- you have a proliferation of classes resulting from a coupled interface and numerous implementations,
- you want to share an implementation among multiple objects,
- you need to map orthogonal class hierarchies.

Consequences include:

- decoupling the object's interface,
- improved extensibility (you can extend (i.e. subclass) the abstraction and implementation hierarchies independently),



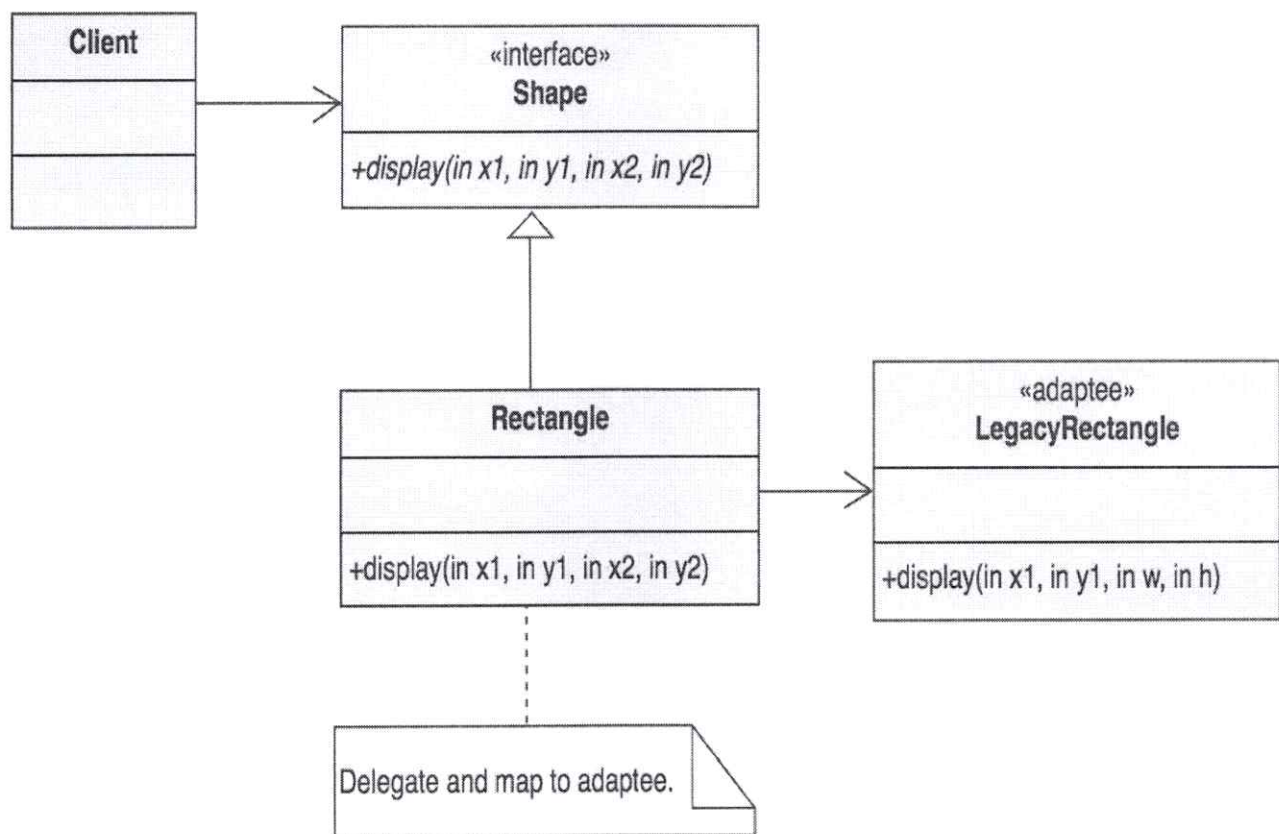
- hiding details from clients.

Bridge is a synonym for the "handle/body" idiom. This is a design mechanism that encapsulates an implementation class inside of an interface class. The former is the body, and the latter is the handle. The handle is viewed by the user as the actual class, but the work is done in the body. "The handle/body class idiom may be used to decompose a complex abstraction into smaller, more manageable classes. The idiom may reflect the sharing of a single resource by multiple classes that control access to it (e.g. reference counting)."

## Structure

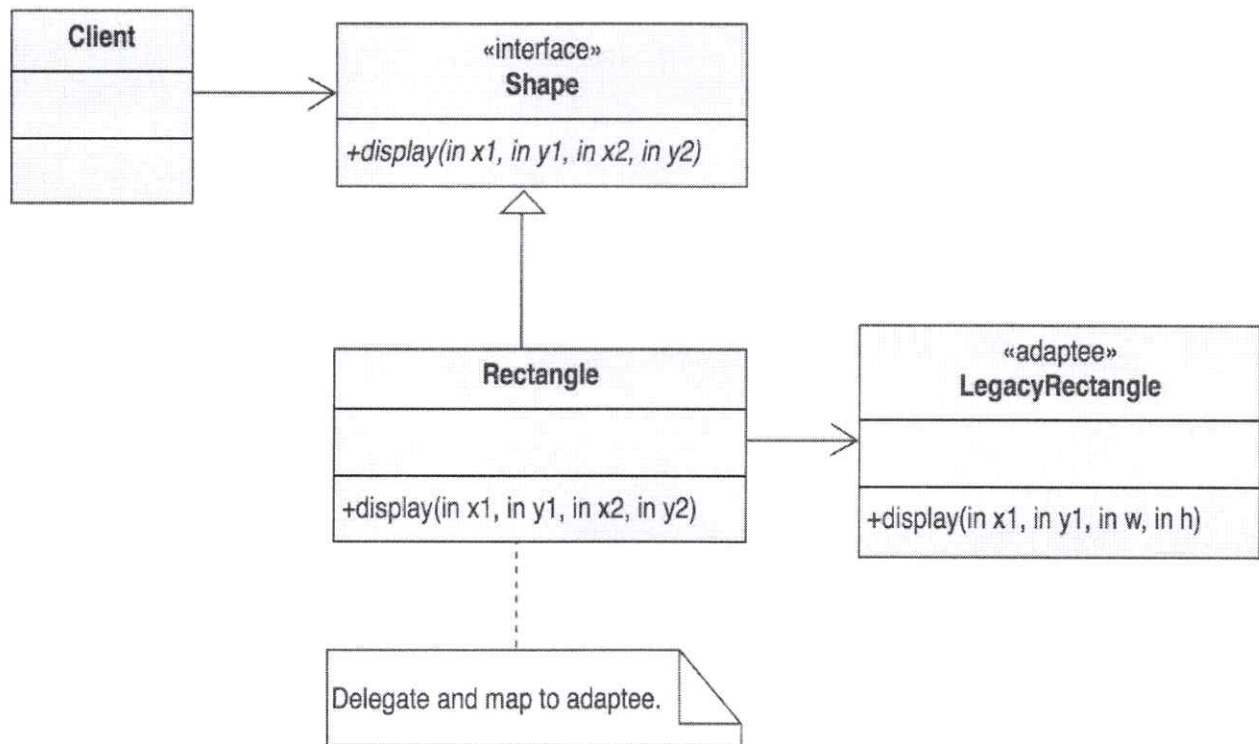
The Client doesn't want to deal with platform-dependent details. The Bridge pattern encapsulates this complexity behind an abstraction "wrapper".

Bridge emphasizes identifying and decoupling "interface" abstraction from "implementation" abstraction.



## Example

The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches.



## Check list

1. Decide if two orthogonal dimensions exist in the domain. These independent concepts could be: abstraction/platform, or domain/infrastructure, or front-end/back-end, or interface/implementation.
2. Design the separation of concerns: what does the client want, and what do the platforms provide.
3. Design a platform-oriented interface that is minimal, necessary, and sufficient. Its goal is to decouple the abstraction from the platform.
4. Define a derived class of that interface for each platform.
5. Create the abstraction base class that "has a" platform object and delegates the platform-oriented functionality to it.
6. Define specializations of the abstraction class if desired.

## Rules of thumb

- Adapter makes things work after they're designed; Bridge makes them work before they are.
- Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together.
- State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom. They differ in intent - that is, they solve different problems.

- The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one). The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently.
- If interface classes delegate the creation of their implementation classes (instead of creating/coupling themselves directly), then the design usually uses the Abstract Factory pattern to create the implementation objects.

## Composite Design Pattern

### Intent

- Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive composition
- "Directories contain entries, each of which could be a directory."
- 1-to-many "has a" up the "is a" hierarchy

### Problem

Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.

### Discussion

Define an abstract base class (Component) that specifies the behavior that needs to be exercised uniformly across all primitive and composite objects. Subclass the Primitive and Composite classes off of the Component class. Each Composite object "couples" itself only to the abstract type Component as it manages its "children".

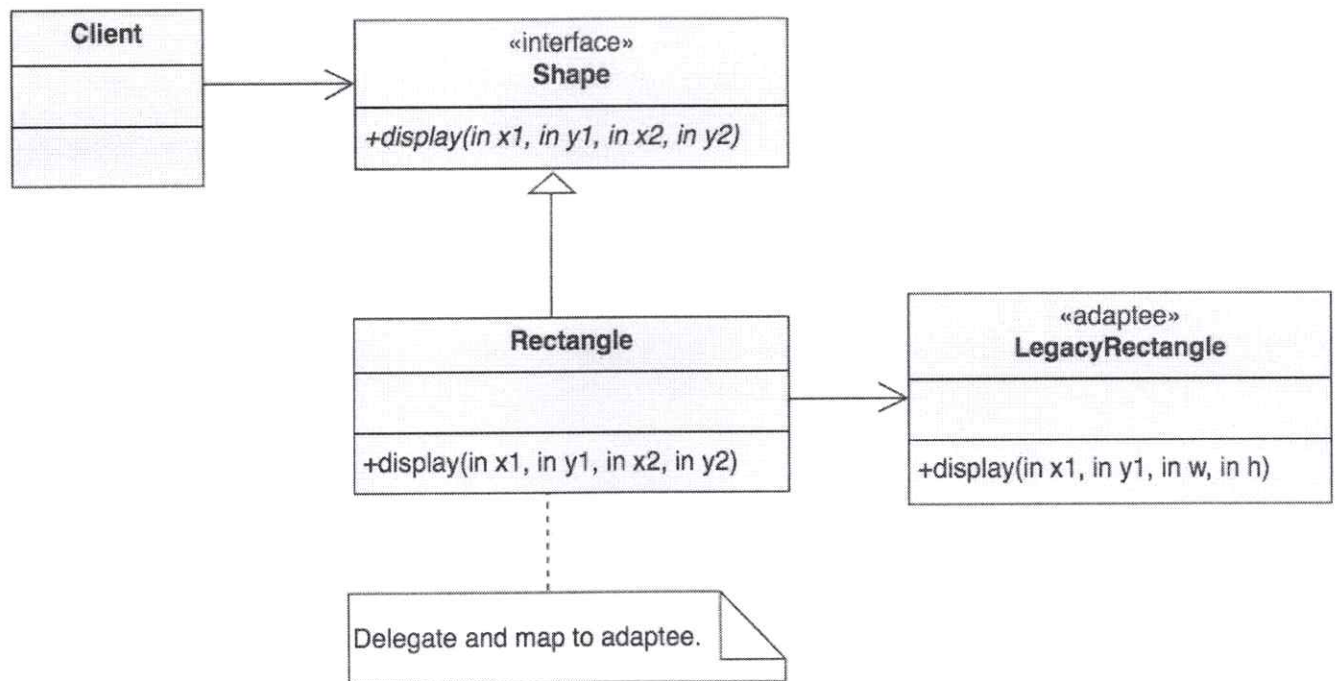
Use this pattern whenever you have "composites that contain components, each of which could be a composite".

Child management methods [e.g. addChild(), removeChild()] should normally be defined in the Composite class. Unfortunately, the desire to treat Primitives and Composites uniformly requires that these methods be moved to the abstract Component class. See the "Opinions" section below for a discussion of "safety" versus "transparency" issues.



## Structure

Composites that contain Components, each of which could be a Composite.



Menus that contain menu items, each of which could be a menu.

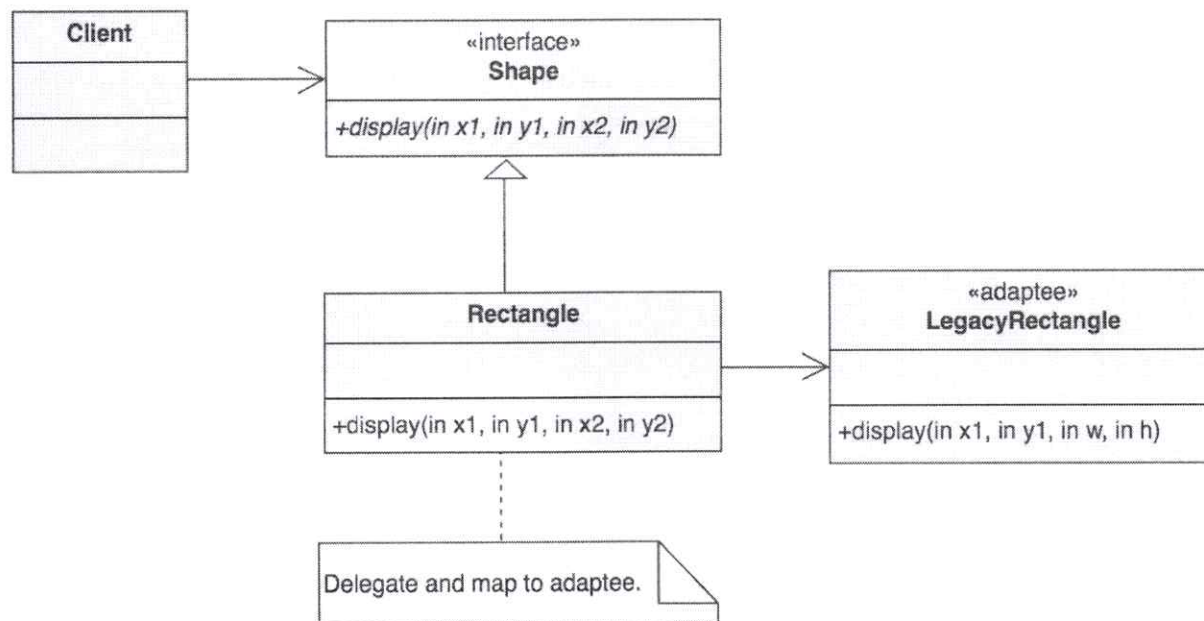
Row-column GUI layout managers that contain widgets, each of which could be a row-column GUI layout manager.

Directories that contain files, each of which could be a directory.

Containers that contain Elements, each of which could be a Container.

## Example

The Composite composes objects into tree structures and lets clients treat individual objects and compositions uniformly. Although the example is abstract, arithmetic expressions are Composites. An arithmetic expression consists of an operand, an operator (+ - \* /), and another operand. The operand can be a number, or another arithmetic expression. Thus,  $2 + 3$  and  $(2 + 3) + (4 * 6)$  are both valid expressions.



## Check list

1. Ensure that your problem is about representing "whole-part" hierarchical relationships.
2. Consider the heuristic, "Containers that contain containees, each of which could be a container." For example, "Assemblies that contain components, each of which could be an assembly." Divide your domain concepts into container classes, and containee classes.
3. Create a "lowest common denominator" interface that makes your containers and containees interchangeable. It should specify the behavior that needs to be exercised uniformly across all containee and container objects.
4. All container and containee classes declare an "is a" relationship to the interface.
5. All container classes declare a one-to-many "has a" relationship to the interface.
6. Container classes leverage polymorphism to delegate to their containee objects.
7. Child management methods [e.g. addChild(), removeChild()] should normally be defined in the Composite class. Unfortunately, the desire to treat Leaf and Composite objects uniformly may require that these methods be promoted to the abstract Component class. See the Gang of Four for a discussion of these "safety" versus "transparency" trade-offs.

## Rules of thumb

- Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.
- Composite can be traversed with Iterator. Visitor can apply an operation over a Composite. Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these



properties on parts of the composition. It could use Observer to tie one object structure to another and State to let a component change its behavior as its state changes.

- Composite can let you compose a Mediator out of smaller pieces through recursive composition.
- Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert.
- Flyweight is often combined with Composite to implement shared leaf nodes.

## Opinions

The whole point of the Composite pattern is that the Composite can be treated atomically, just like a leaf. If you want to provide an Iterator protocol, fine, but I think that is outside the pattern itself. At the heart of this pattern is the ability for a client to perform operations on an object without needing to know that there are many objects inside.

Being able to treat a heterogeneous collection of objects atomically (or transparently) requires that the "child management" interface be defined at the root of the Composite class hierarchy (the abstract Component class). However, this choice costs you safety, because clients may try to do meaningless things like add and remove objects from leaf objects. On the other hand, if you "design for safety", the child management interface is declared in the Composite class, and you lose transparency because leaves and Composites now have different interfaces.

Smalltalk implementations of the Composite pattern usually do not have the interface for managing the components in the Component interface, but in the Composite interface. C++ implementations tend to put it in the Component interface. This is an extremely interesting fact, and one that I often ponder. I can offer theories to explain it, but nobody knows for sure why it is true.

My Component classes do not know that Composites exist. They provide no help for navigating Composites, nor any help for altering the contents of a Composite. This is because I would like the base class (and all its derivatives) to be reusable in contexts that do not require Composites. When given a base class pointer, if I absolutely need to know whether or not it is a Composite, I will use `dynamic_cast` to figure this out. In those cases where `dynamic_cast` is too expensive, I will use a Visitor.

Common complaint: "if I push the Composite interface down into the Composite class, how am I going to enumerate (i.e. traverse) a complex structure?" My answer is that when I have behaviors which apply to hierarchies like the one presented in the Composite pattern, I typically use Visitor, so enumeration isn't a problem - the Visitor knows in each case, exactly what kind of object it's dealing with. The Visitor doesn't need every object to provide an enumeration interface.

Composite doesn't force you to treat all Components as Composites. It merely tells you to put all operations that you want to treat "uniformly" in the Component class. If add, remove, and similar operations cannot, or must not, be treated uniformly, then do not put them in the Component base class. Remember, by the way, that each pattern's structure diagram doesn't define the pattern; it merely depicts what in our experience is a common realization thereof.

## Structural Pattern Part-II

## Decorator Design Pattern

## Intent

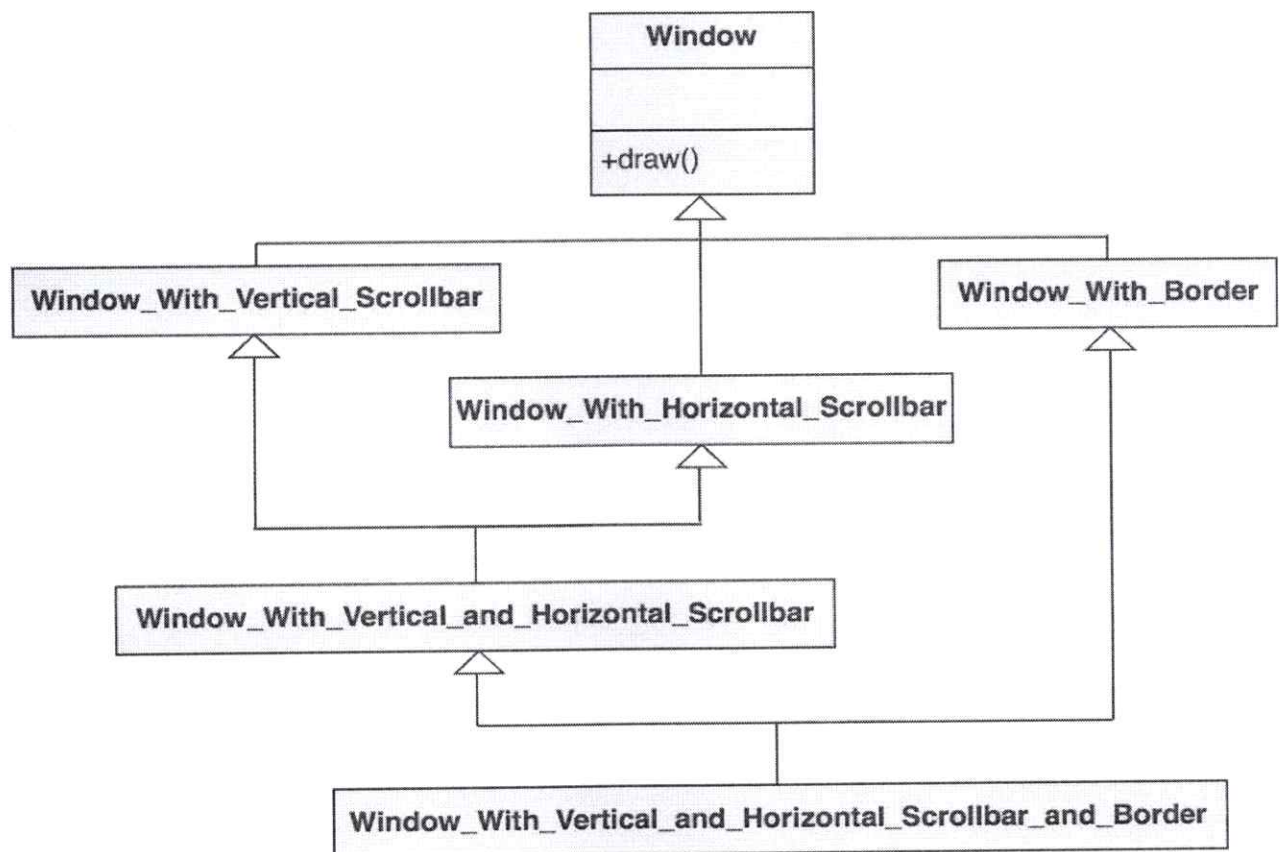
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Client-specified embellishment of a core object by recursively wrapping it.
- Wrapping a gift, putting it in a box, and wrapping the box.

## Problem

You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

## Discussion

Suppose you are working on a user interface toolkit and you wish to support adding borders and scroll bars to windows. You could define an inheritance hierarchy like ...



But the Decorator pattern suggests giving the client the ability to specify whatever combination of "features" is desired.

```
Widget* aWidget = new BorderDecorator(
    new HorizontalScrollBarDecorator(
```

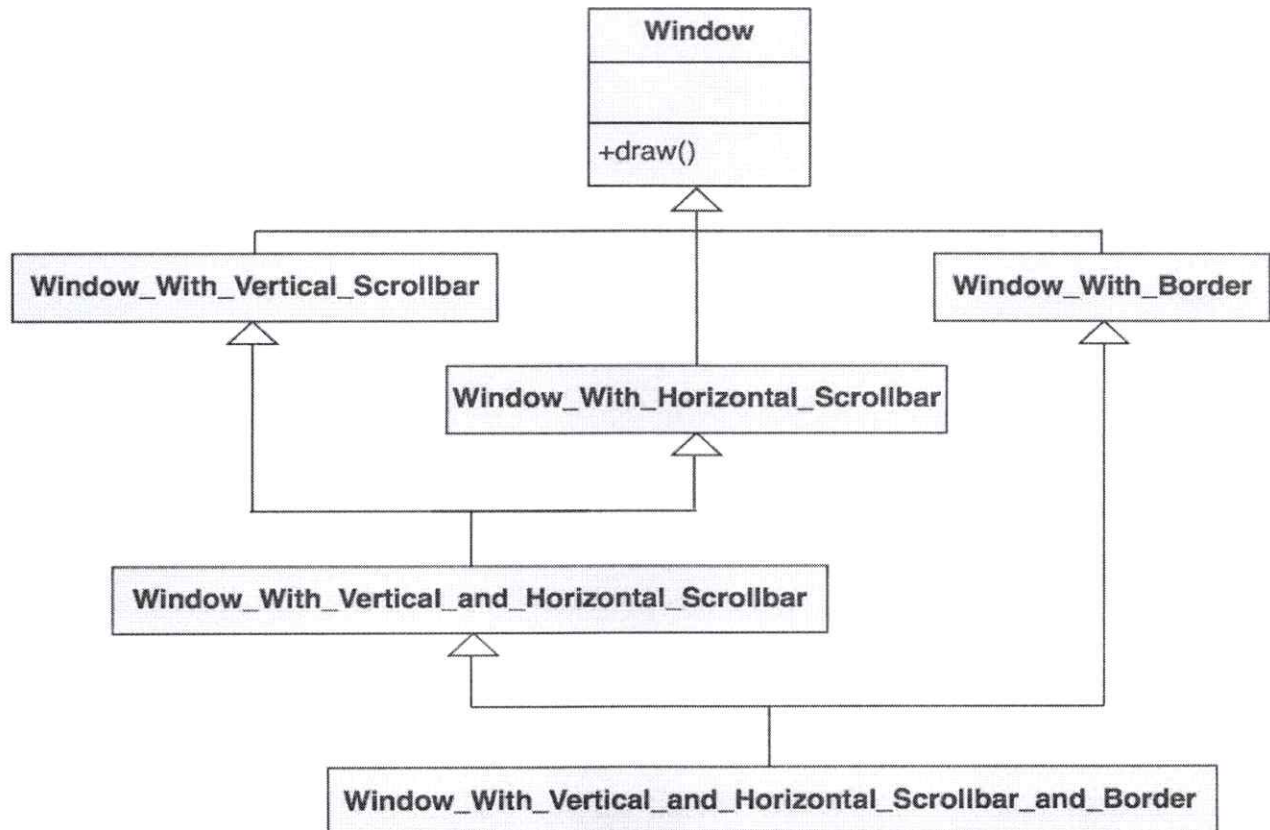


```

new VerticalScrollBarDecorator(
    new Window( 80, 24 )));
aWidget->draw();

```

This flexibility can be achieved with the following design



Another example of cascading (or chaining) features together to produce a custom object might look like ...

```

Stream* aStream = new CompressingStream(
    new ASCII7Stream(
        new FileStream("fileName.dat")));
aStream->putString( "Hello world" );

```

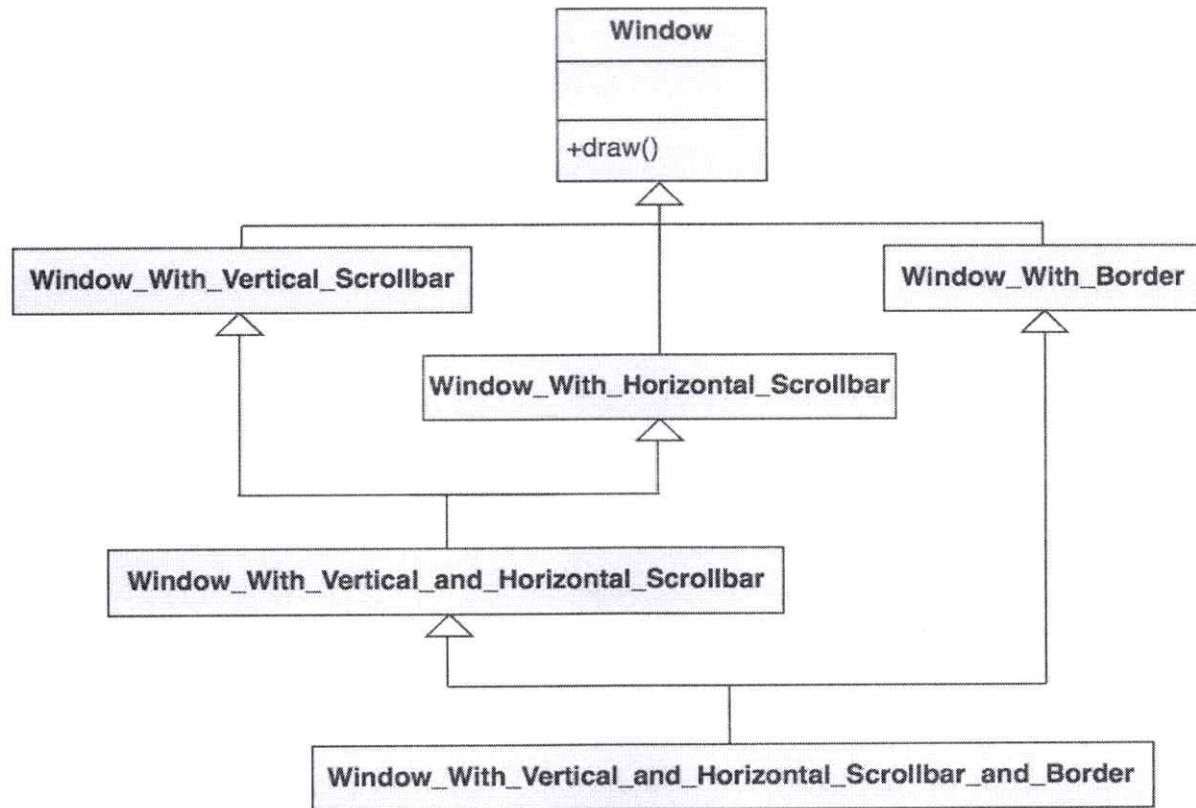
The solution to this class of problems involves encapsulating the original object inside an abstract wrapper interface. Both the decorator objects and the core object inherit from this abstract interface. The interface uses recursive composition to allow an unlimited number of decorator "layers" to be added to each core object.

Note that this pattern allows responsibilities to be added to an object, not methods to an object's interface. The interface presented to the client must remain constant as successive layers are specified.

Also note that the core object's identity has now been "hidden" inside of a decorator object. Trying to access the core object directly is now a problem.

## Structure

The client is always interested in `CoreFunctionality.doThis()`. The client may, or may not, be interested in `OptionalOne.doThis()` and `OptionalTwo.doThis()`. Each of these classes always delegate to the Decorator base class, and that class always delegates to the contained "wrappee" object.

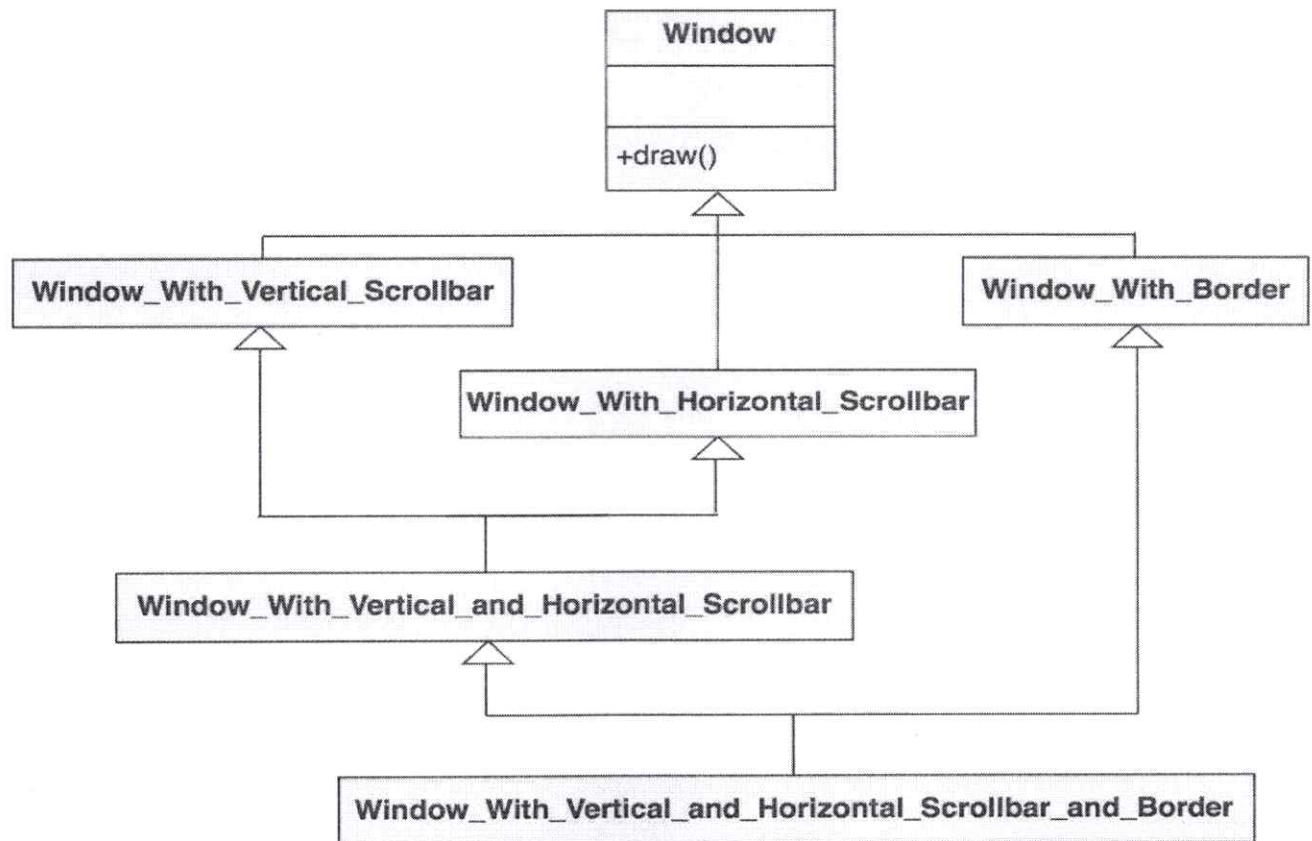


## Example

The Decorator attaches additional responsibilities to an object dynamically. The ornaments that are added to pine or fir trees are examples of Decorators. Lights, garland, candy canes, glass ornaments, etc., can be added to a tree to give it a festive look. The ornaments do not change the tree itself which is recognizable as a Christmas tree regardless of particular ornaments used. As an example of additional functionality, the addition of lights allows one to "light up" a Christmas tree.

Another example: assault gun is a deadly weapon on it's own. But you can apply certain "decorations" to make it more accurate, silent and devastating.





### Check list

1. Ensure the context is: a single core (or non-optional) component, several optional embellishments or wrappers, and an interface that is common to all.
2. Create a "Lowest Common Denominator" interface that makes all classes interchangeable.
3. Create a second level base class (Decorator) to support the optional wrapper classes.
4. The Core class and Decorator class inherit from the LCD interface.
5. The Decorator class declares a composition relationship to the LCD interface, and this data member is initialized in its constructor.
6. The Decorator class delegates to the LCD object.
7. Define a Decorator derived class for each optional embellishment.
8. Decorator derived classes implement their wrapper functionality - and - delegate to the Decorator base class.
9. The client configures the type and ordering of Core and Decorator objects.

### Rules of thumb

- Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.

- Adapter changes an object's interface, Decorator enhances an object's responsibilities. Decorator is thus more transparent to the client. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters.
- Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.
- A Decorator can be viewed as a degenerate Composite with only one component. However, a Decorator adds additional responsibilities - it isn't intended for object aggregation.
- Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert.
- Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition.
- Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.
- Decorator lets you change the skin of an object. Strategy lets you change the guts.

## **Facade Design Pattern:-**

### **Intent**

- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Wrap a complicated subsystem with a simpler interface.

### **Problem**

A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

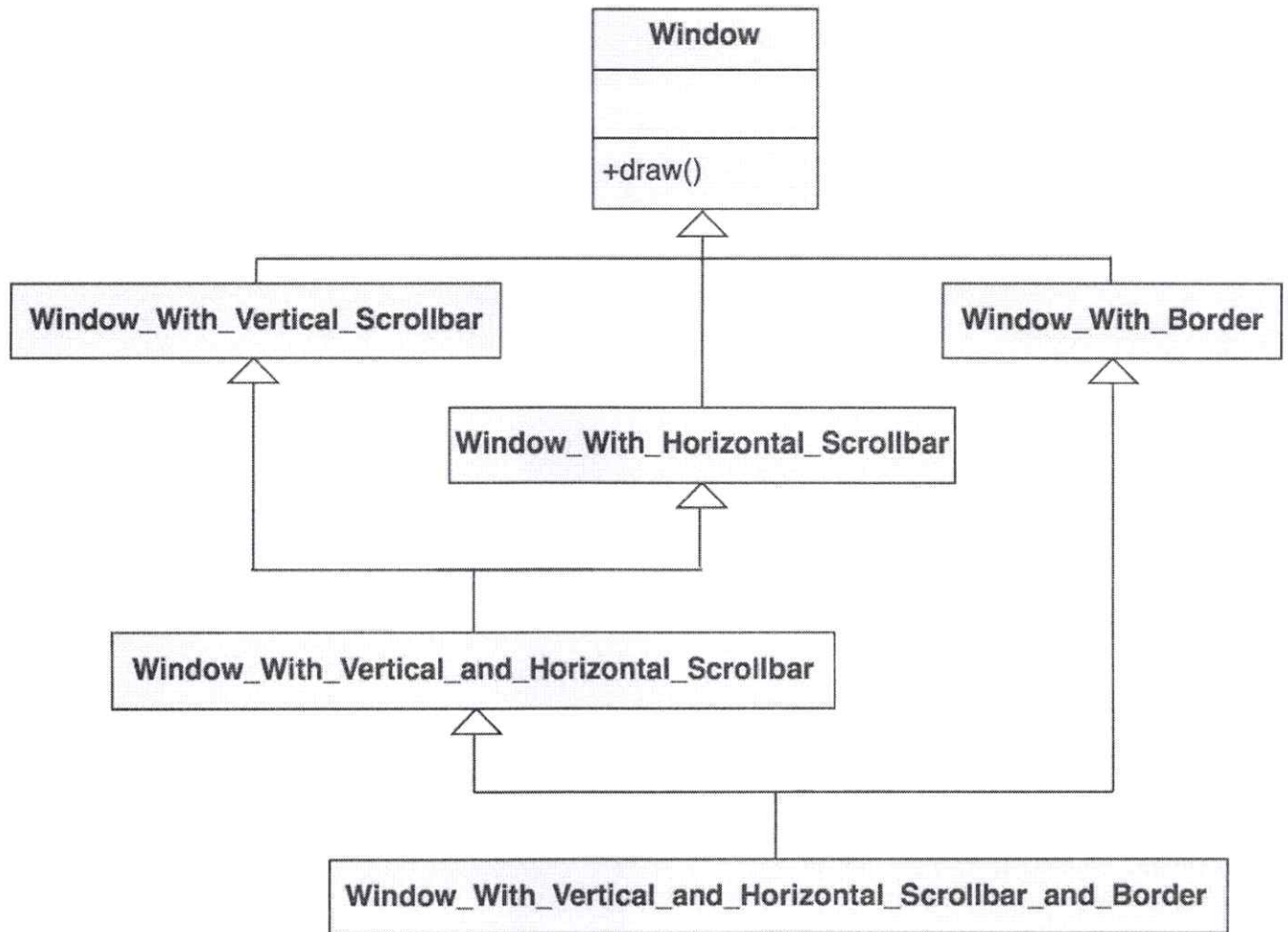
### **Discussion**

Facade discusses encapsulating a complex subsystem within a single interface object. This reduces the learning curve necessary to successfully leverage the subsystem. It also promotes decoupling the subsystem from its potentially many clients. On the other hand, if the Facade is the only access point for the subsystem, it will limit the features and flexibility that "power users" may need.

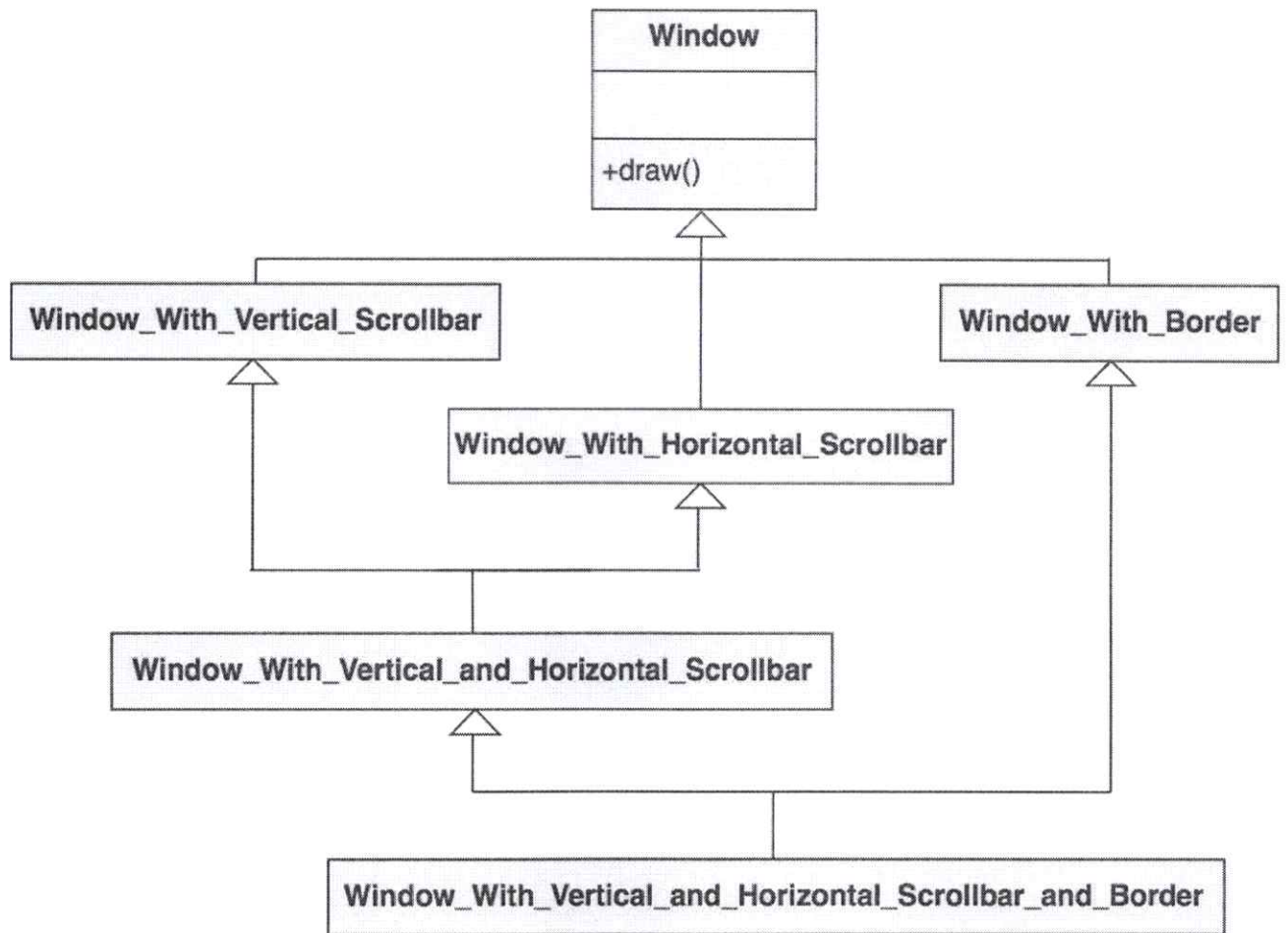
The Facade object should be a fairly simple advocate or facilitator. It should not become an all-knowing oracle or "god" object.

## Structure

Facade takes a "riddle wrapped in an enigma shrouded in mystery", and interjects a wrapper that tames the amorphous and inscrutable mass of software.



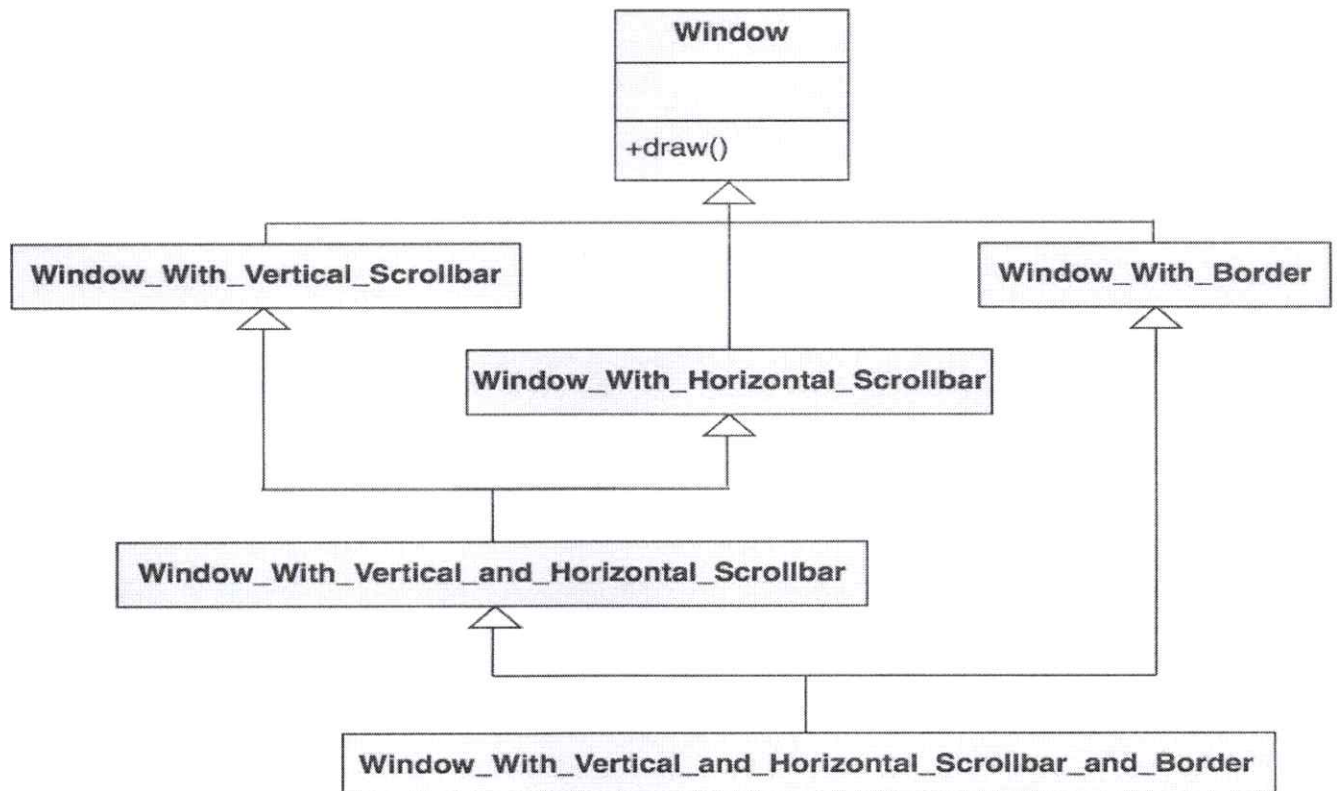
SubsystemOne and SubsystemThree do not interact with the internal components of SubsystemTwo. They use the SubsystemTwoWrapper "facade" (i.e. the higher level abstraction).



### Example

The Facade defines a unified, higher level interface to a subsystem that makes it easier to use. Consumers encounter a Facade when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a Facade, providing an interface to the order fulfillment department, the billing department, and the shipping department.





### Check list

1. Identify a simpler, unified interface for the subsystem or component.
2. Design a 'wrapper' class that encapsulates the subsystem.
3. The facade/wrapper captures the complexity and collaborations of the component, and delegates to the appropriate methods.
4. The client uses (is coupled to) the Facade only.
5. Consider whether additional Facades would add value.

### Rules of thumb

- Facade defines a new interface, whereas Adapter uses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one.
- Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem.
- Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communications between colleague objects. It routinely "adds value", and it is known/referenced by the colleague objects. In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes.
- Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.

- Facade objects are often Singletons because only one Facade object is required.
- Adapter and Facade are both wrappers; but they are different kinds of wrappers. The intent of Facade is to produce a simpler interface, and the intent of Adapter is to design to an existing interface. While Facade routinely wraps multiple objects and Adapter wraps a single object; Facade could front-end a single complex object and Adapter could wrap several legacy objects.

Question: So the way to tell the difference between the Adapter pattern and the Facade pattern is that the Adapter wraps one class and the Facade may represent many classes?

Answer: No! Remember, the Adapter pattern changes the interface of one or more classes into one interface that a client is expecting. While most textbook examples show the adapter adapting one class, you may need to adapt many classes to provide the interface a client is coded to. Likewise, a Facade may provide a simplified interface to a single class with a very complex interface. The difference between the two is not in terms of how many classes they "wrap", it is in their intent.

## **Flyweight Design Pattern:-**

### **Intent**

- Use sharing to support large numbers of fine-grained objects efficiently.
- The Motif GUI strategy of replacing heavy-weight widgets with light-weight gadgets.

### **Problem**

Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage.

### **Discussion**

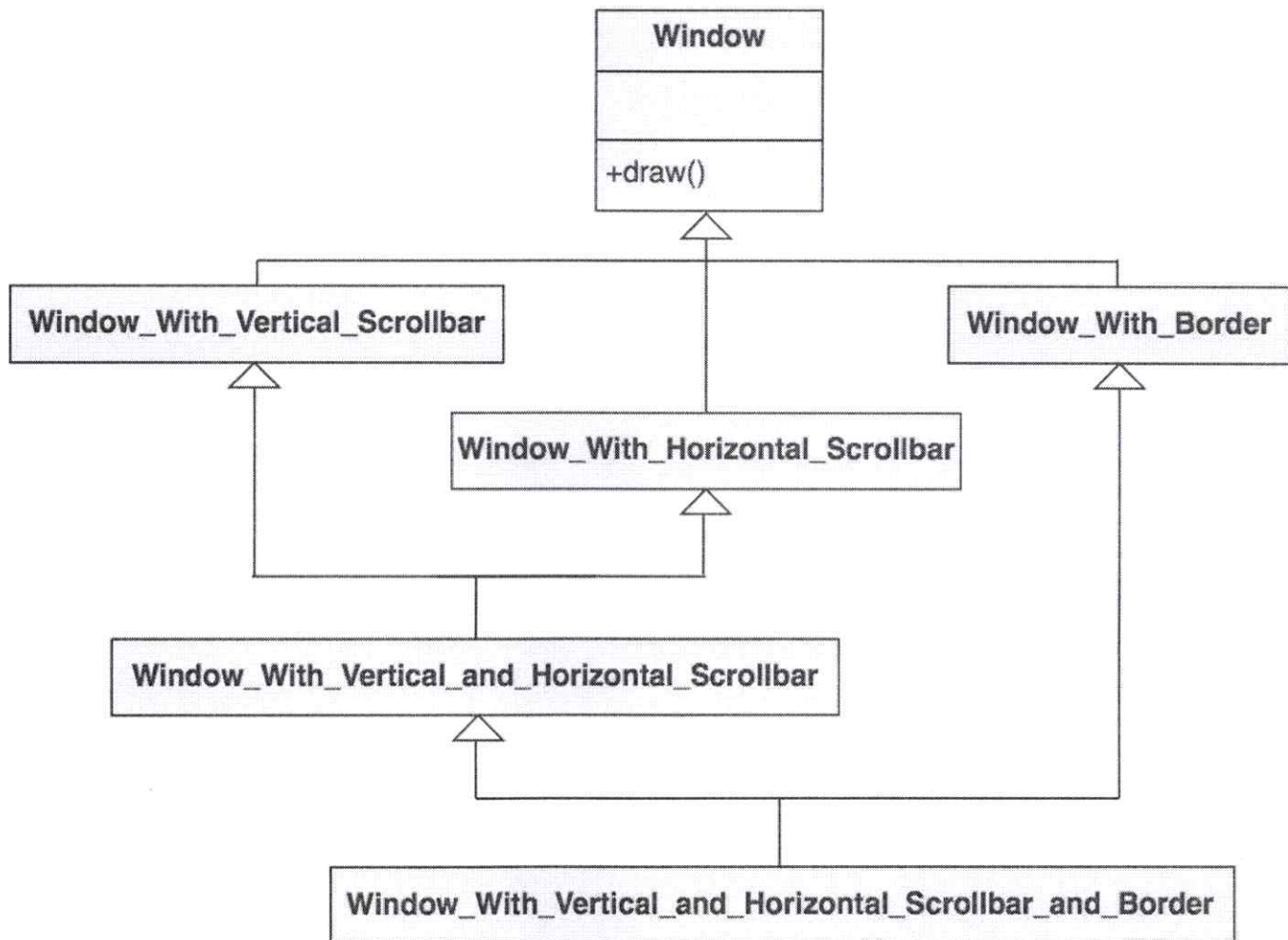
The Flyweight pattern describes how to share objects to allow their use at fine granularities without prohibitive cost. Each "flyweight" object is divided into two pieces: the state-dependent (extrinsic) part, and the state-independent (intrinsic) part. Intrinsic state is stored (shared) in the Flyweight object. Extrinsic state is stored or computed by client objects, and passed to the Flyweight when its operations are invoked.

An illustration of this approach would be Motif widgets that have been re-engineered as light-weight gadgets. Whereas widgets are "intelligent" enough to stand on their own; gadgets exist in a dependent relationship with their parent layout manager widget. Each layout manager provides context-dependent event handling, real estate management, and resource services to its flyweight gadgets, and each gadget is only responsible for context-independent state and behavior.

### **Structure**

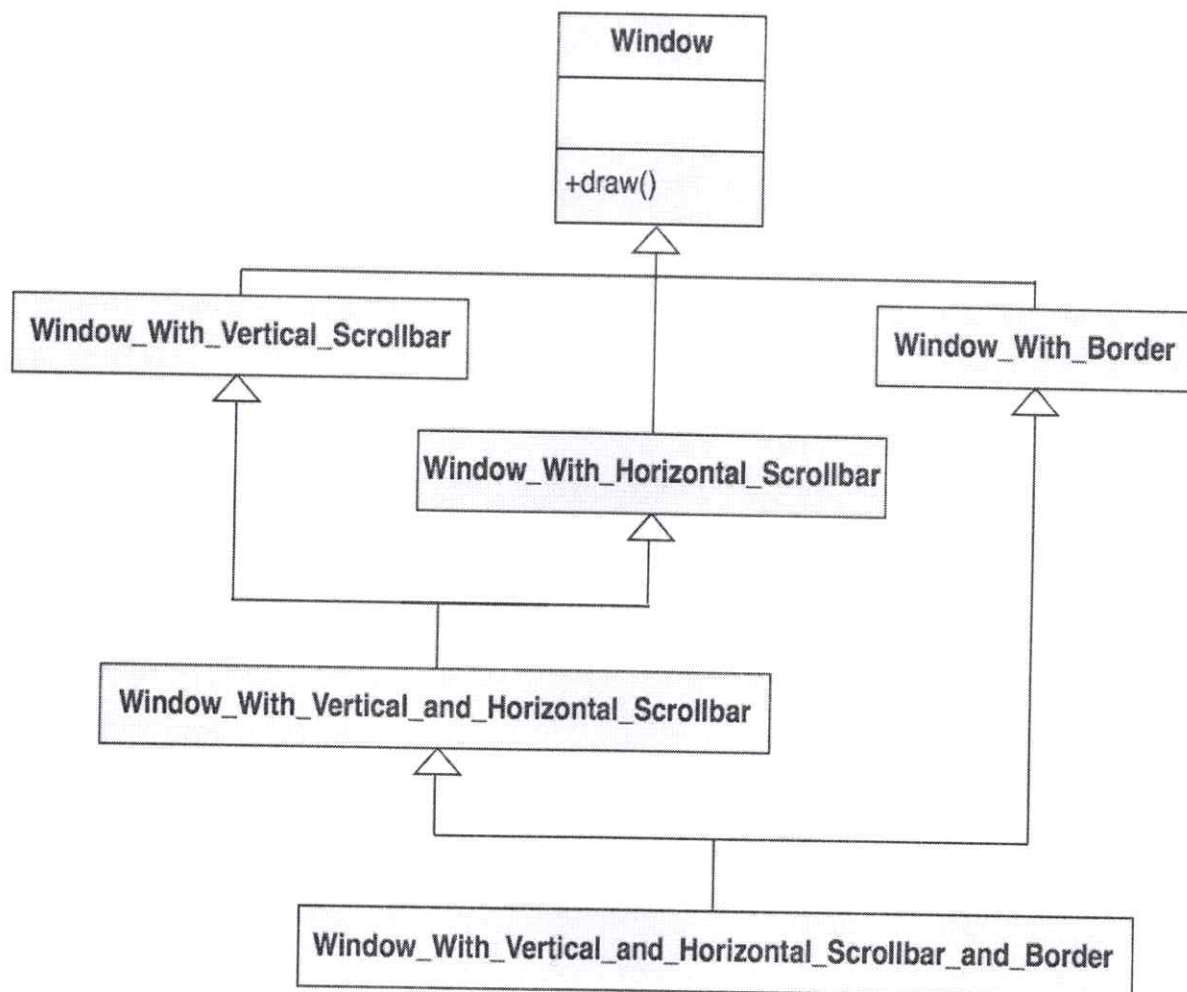
Flyweights are stored in a Factory's repository. The client restrains herself from creating Flyweights directly, and requests them from the Factory. Each Flyweight cannot stand on its

own. Any attributes that would make sharing impossible must be supplied by the client whenever a request is made of the Flyweight. If the context lends itself to "economy of scale" (i.e. the client can easily compute or look-up the necessary attributes), then the Flyweight pattern offers appropriate leverage.



The Ant, Locust, and Cockroach classes can be "light-weight" because their instance-specific state has been de-encapsulated, or externalized, and must be supplied by the client.

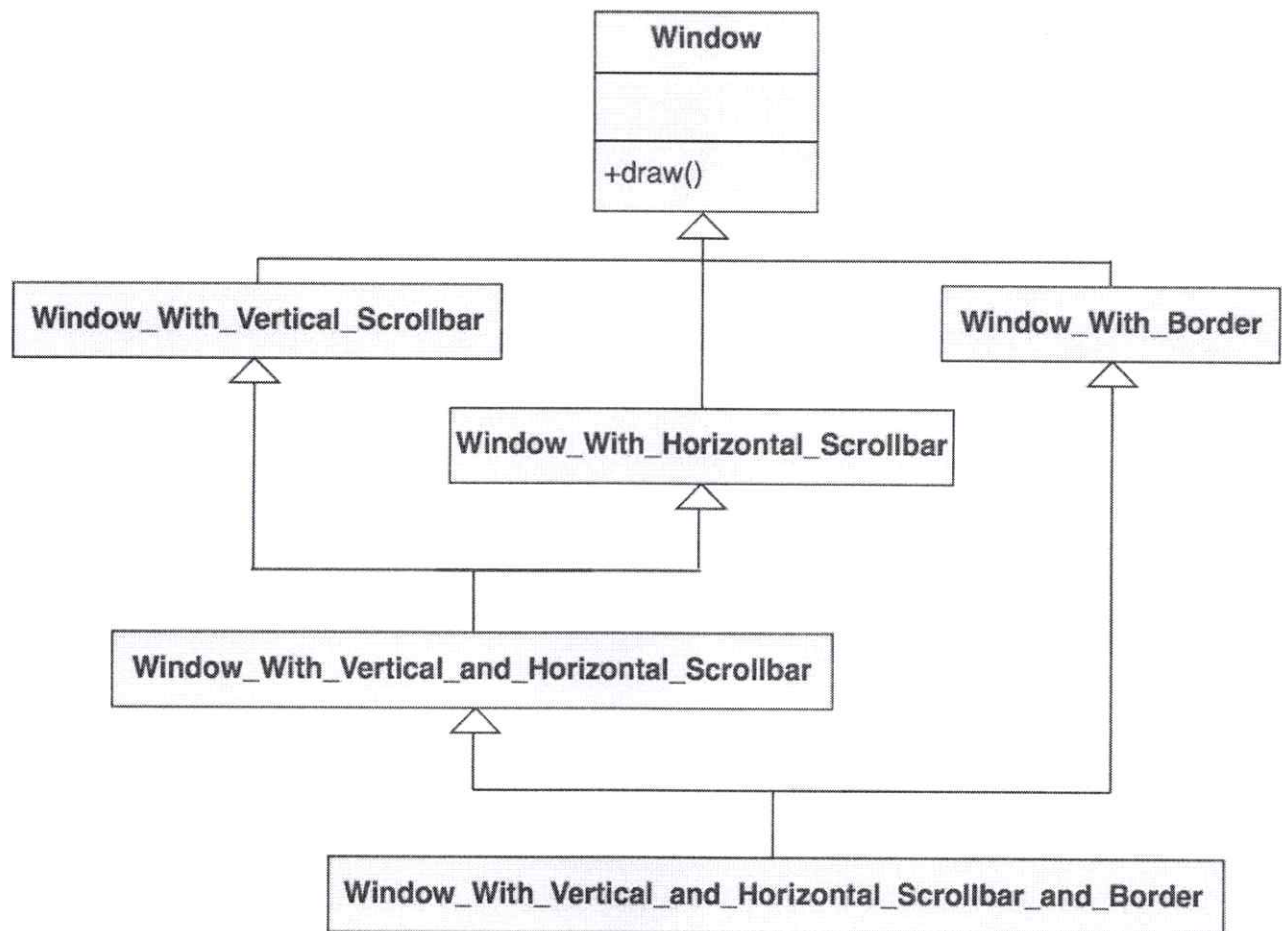




### Example

The Flyweight uses sharing to support large numbers of objects efficiently. Modern web browsers use this technique to prevent loading same images twice. When browser loads a web page, it traverse through all images on that page. Browser loads all new images from Internet and places them the internal cache. For already loaded images, a flyweight object is created, which has some unique data like position within the page, but everything else is referenced to the cached one.





### Check list

1. Ensure that object overhead is an issue needing attention, and, the client of the class is able and willing to absorb responsibility realignment.
2. Divide the target class's state into: shareable (intrinsic) state, and non-shareable (extrinsic) state.
3. Remove the non-shareable state from the class attributes, and add it the calling argument list of affected methods.
4. Create a Factory that can cache and reuse existing class instances.
5. The client must use the Factory instead of the new operator to request objects.
6. The client (or a third party) must look-up or compute the non-shareable state, and supply that state to class methods.

### Rules of thumb

- Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem.
- Flyweight is often combined with Composite to implement shared leaf nodes.
- Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight.

- Flyweight explains when and how State objects can be shared.

## **Proxy Design Pattern:-**

### **Intent**

- Provide a surrogate or placeholder for another object to control access to it.
- Use an extra level of indirection to support distributed, controlled, or intelligent access.
- Add a wrapper and delegation to protect the real component from undue complexity.

### **Problem**

You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.

### **Discussion**

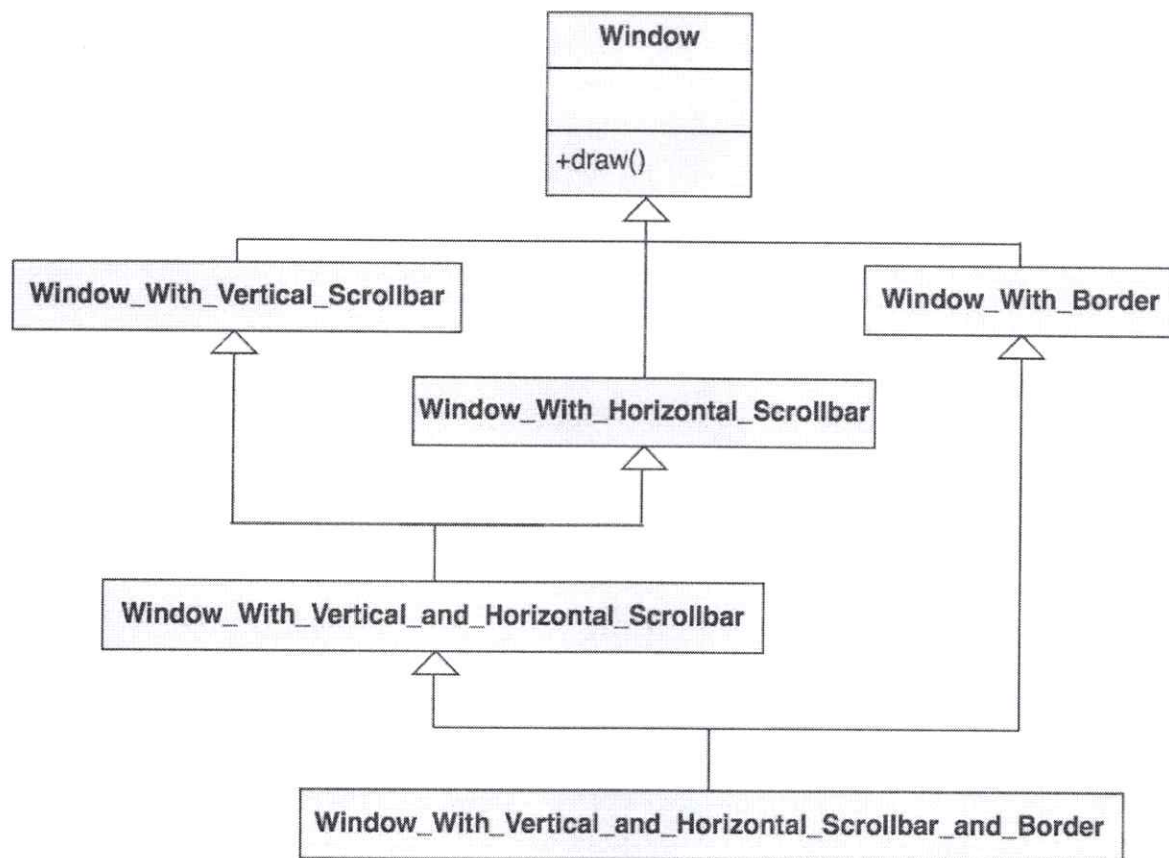
Design a surrogate, or proxy, object that: instantiates the real object the first time the client makes a request of the proxy, remembers the identity of this real object, and forwards the instigating request to this real object. Then all subsequent requests are simply forwarded directly to the encapsulated real object.

There are four common situations in which the Proxy pattern is applicable.

1. A virtual proxy is a placeholder for "expensive to create" objects. The real object is only created when a client first requests/accesses the object.
2. A remote proxy provides a local representative for an object that resides in a different address space. This is what the "stub" code in RPC and CORBA provides.
3. A protective proxy controls access to a sensitive master object. The "surrogate" object checks that the caller has the access permissions required prior to forwarding the request.
4. A smart proxy interposes additional actions when an object is accessed. Typical uses include:
  - Counting the number of references to the real object so that it can be freed automatically when there are no more references (aka smart pointer),
  - Loading a persistent object into memory when it's first referenced,
  - Checking that the real object is locked before it is accessed to ensure that no other object can change it.

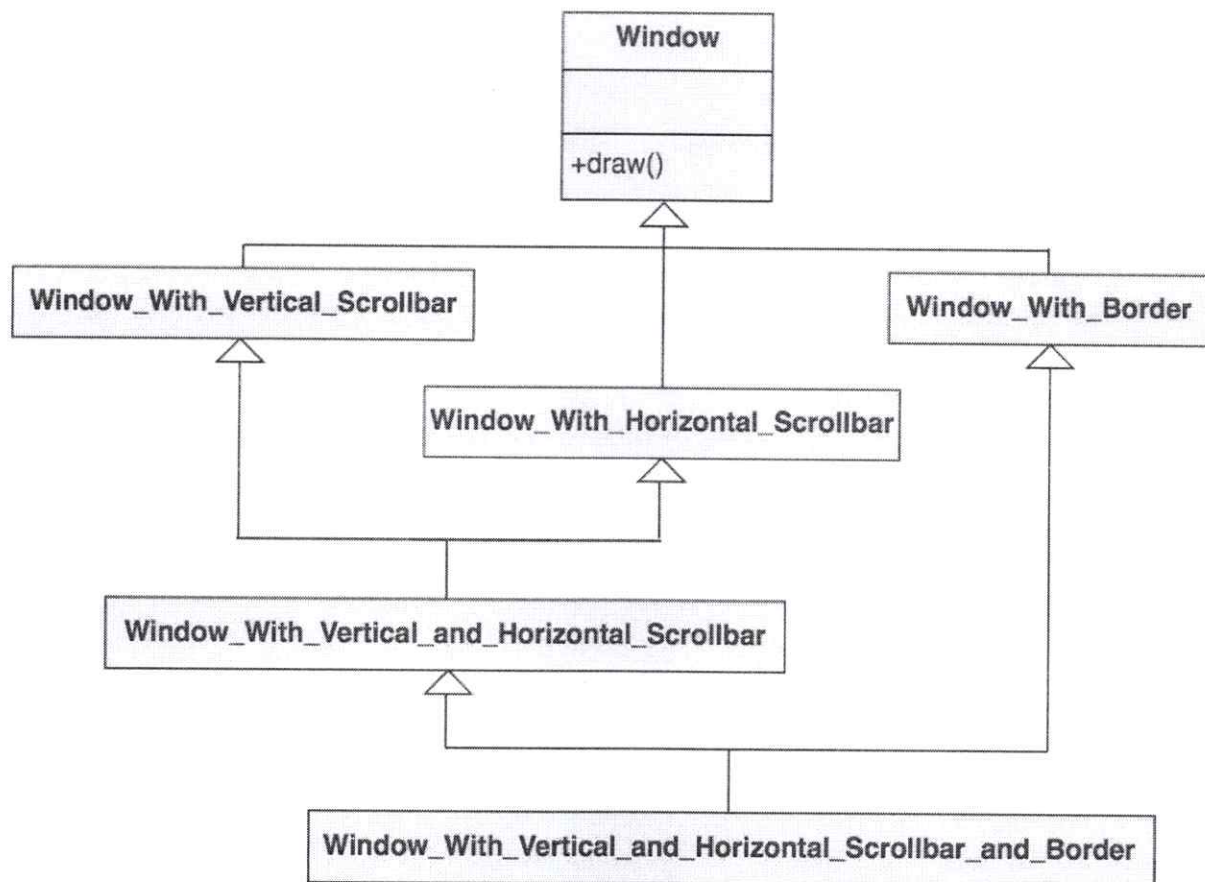
### **Structure**

By defining a Subject interface, the presence of the Proxy object standing in place of the RealSubject is transparent to the client.



### Example

The Proxy provides a surrogate or place holder to provide access to an object. A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account.



### Check list

1. Identify the leverage or "aspect" that is best implemented as a wrapper or surrogate.
2. Define an interface that will make the proxy and the original component interchangeable.
3. Consider defining a Factory that can encapsulate the decision of whether a proxy or original object is desirable.
4. The wrapper class holds a pointer to the real class and implements the interface.
5. The pointer may be initialized at construction, or on first use.
6. Each wrapper method contributes its leverage, and delegates to the wrappee object.

### Rules of thumb

- Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.
- Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.



## UNIT-IV

### Behavioral Patterns

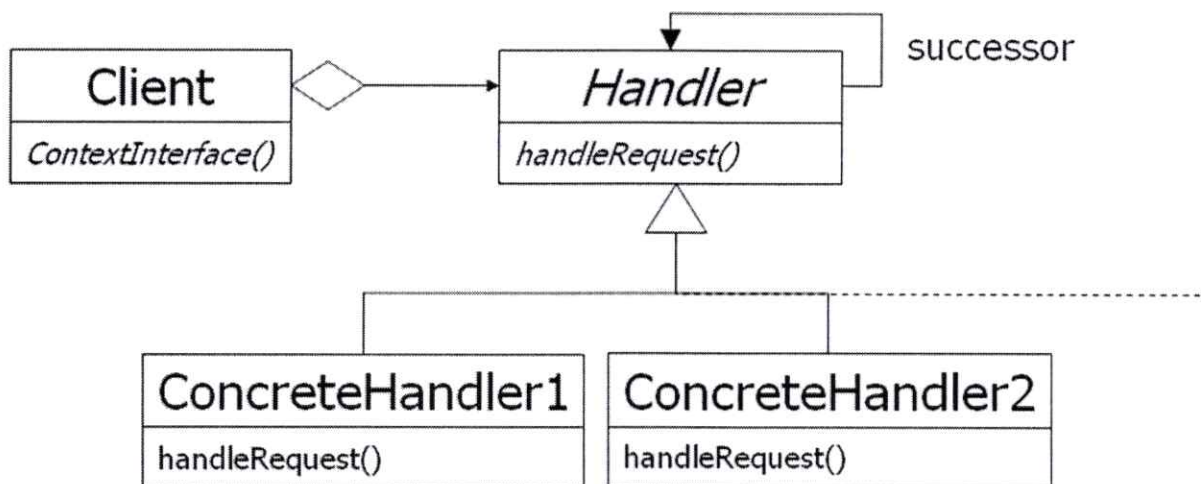
Behavioural Patterns Part-I : Chain of Responsibility, Command, Interpreter, Iterator.

#### Behavioral Patterns (1)

- Deal with the way objects interact and distribute responsibility.
- Chain of Responsibility: Avoid coupling the sender of a request to its 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 object handles it.
- Command: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Interpreter: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

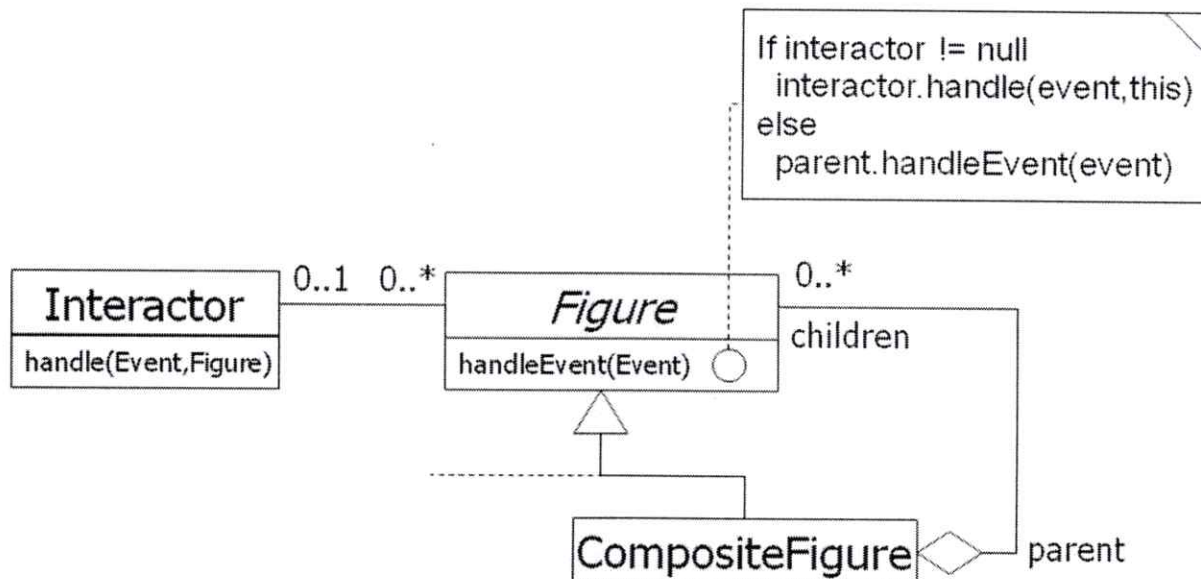
#### Chain of Responsibility:

- Decouple sender of a request from receiver.
- Give more than one object a chance to handle.
- Flexibility in assigning responsibility.
- Often applied with Composite.



## Chain of Responsibility (2)

- Example: handling events in a graphical hierarchy



## Command: Encapsulating Control Flow :

Name: Command design pattern

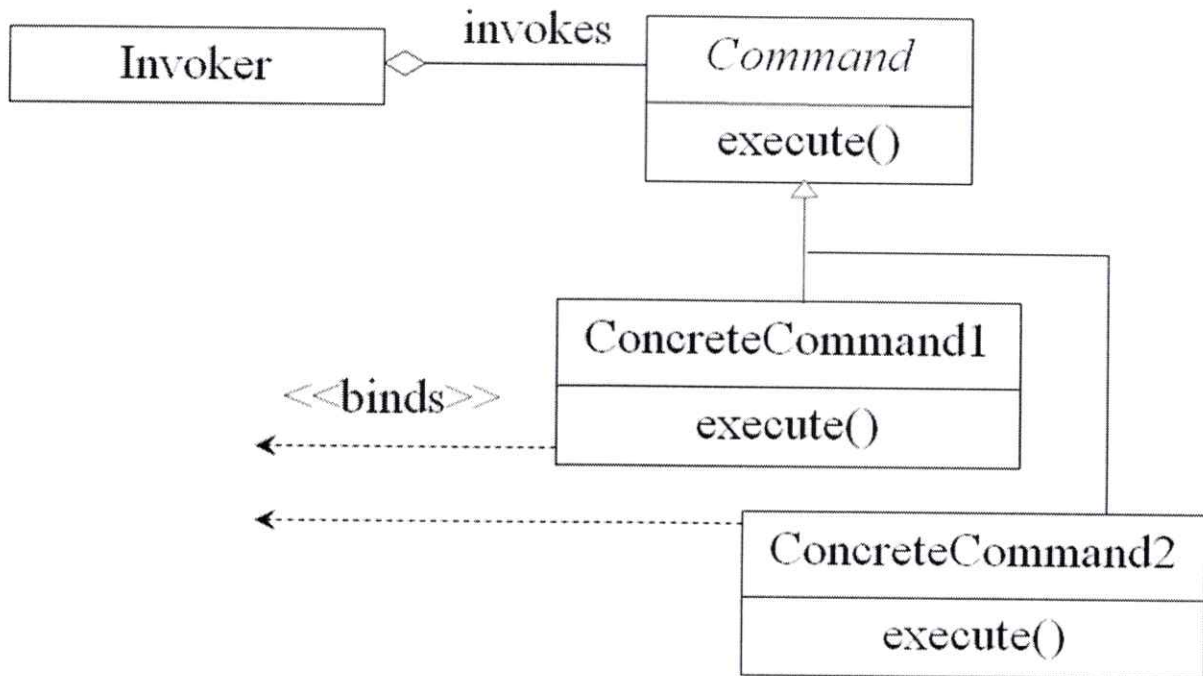
Problem description:

Encapsulates requests so that they can be executed, undone, or queued independently of the request.

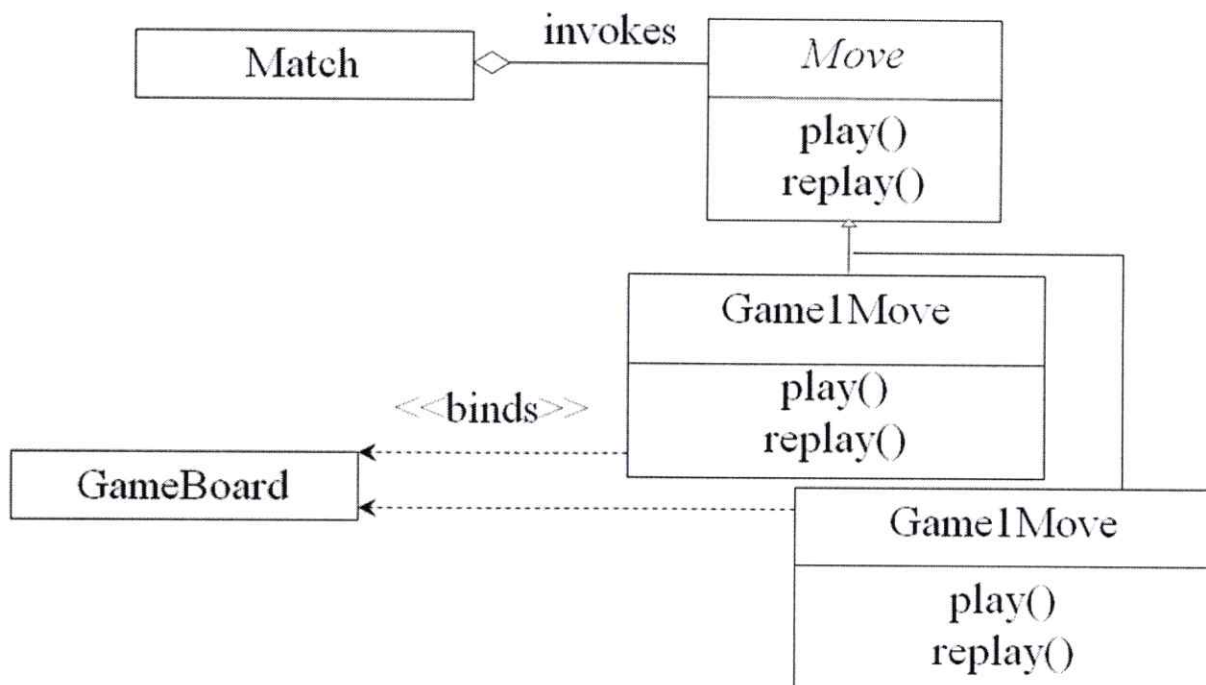
Solution:

A Command abstract class declares the interface supported by all ConcreteCommands. ConcreteCommands encapsulate a service to be applied to a Receiver. The Client creates ConcreteCommands and binds them to specific Receivers. The Invoker actually executes a command.

### Command: Class Diagram



### Command: Class Diagram for Match



## **Command: Consequences**

Consequences:

The object of the command (Receiver) and the algorithm of the command. (ConcreteCommand) are decoupled.

Invoker is shielded from specific commands.

ConcreteCommands are objects. They can be created and stored.

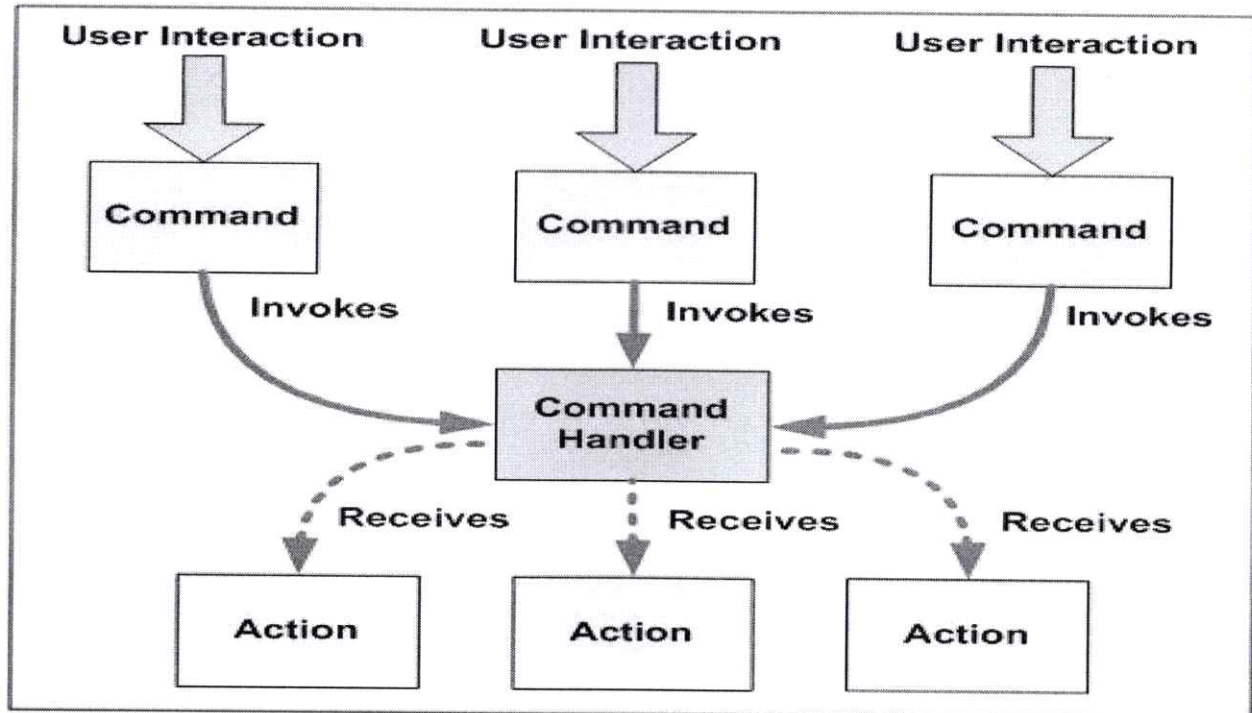
New ConcreteCommands can be added without changing existing code.

## **Command:**

- You have commands that need to be
  - executed,
  - undone, or
  - queued
- Command design pattern separates
  - Receiver from Invoker from Commands
- All commands derive from Command and implement do(), undo(), and redo().



## Command Design Pattern :



- Separates command invoker and receiver.

## Pattern: Interpreter:

- Intent: Given a language, interpret sentences.
- Participants: Expressions, Context, Client.
- Implementation: A class for each expression type  
An Interpret method on each class  
A class and object to store the global state (context)
- No support for the parsing process  
(Assumes strings have been parsed into exp trees)

## Pattern: Interpreter with Macros:

- Example: Definite Clause Grammars.
- A language for writing parsers/interpreters.
- Macros make it look like (almost) standard BNF.  
Command(move(D)) -> "go", Direction(D).
- Built-in to Prolog; easy to implement in Dylan, Lisp.
- Does parsing as well as interpretation.

- Builds tree structure only as needed.  
(Or, can automatically build complete trees)
- May or may not use expression classes.

### Method Combination:

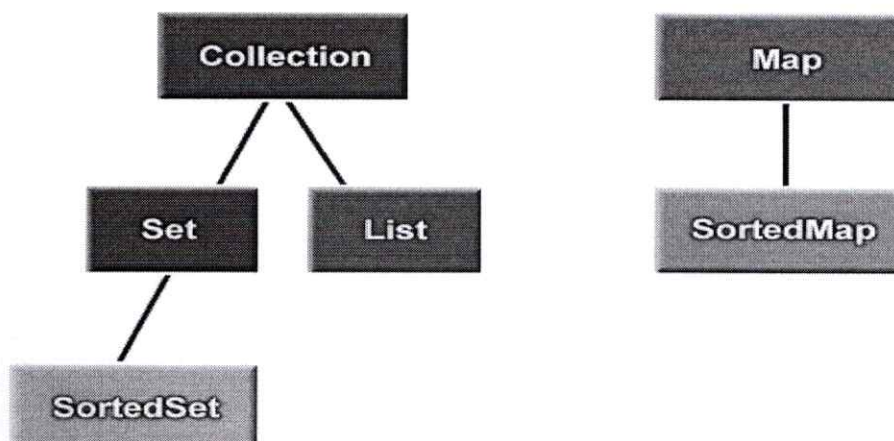
- Build a method from components in different classes
- Primary methods: the “normal” methods; choose the most specific one
- Before/After methods: guaranteed to run;  
No possibility of forgetting to call super  
Can be used to implement *Active Value* pattern
- Around methods: wrap around everything;  
Used to add tracing information, etc.
- Is added complexity worth it?  
Common Lisp: Yes; Most languages: No

### Iterator pattern :

- **iterator**: an object that provides a standard way to examine all elements of any collection.
- uniform interface for traversing many different data structures without exposing their implementations.
- supports concurrent iteration and element removal.
- removes need to know about internal structure of collection or different methods to access data from different collections.

### Pattern: Iterator

objects that traverse collections



## **Iterator interfaces in Java:**

```
public interface java.util.Iterator {  
    public boolean hasNext();  
    public Object next();  
    public void remove();  
}  
  
public interface java.util.Collection {  
    ... // List, Set extend Collection  
    public Iterator iterator();  
}  
  
public interface java.util.Map {  
    ...  
    public Set keySet(); // keys, values are Collections  
    public Collection values(); // (can call iterator() on them)  
}
```

## **Iterators in Java:**

- all Java collections have a method `iterator` that returns an iterator for the elements of the collection.
- can be used to look through the elements of any kind of collection (an alternative to for loop).

```
List list = new ArrayList();  
... add some elements ...  
for (Iterator itr = list.iterator(); itr.hasNext()) {  
    BankAccount ba = (BankAccount)itr.next();  
    System.out.println(ba);  
}
```

## Adding your own Iterators :

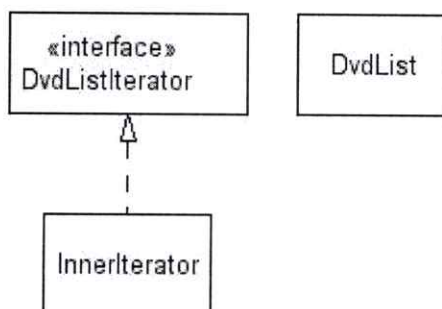
- when implementing your own collections, it can be very convenient to use Iterators

– **discouraged (has nonstandard interface):**

```
public class PlayerList {  
    public int getNumPlayers() { ... }  
    public boolean empty() { ... } public  
    Player getPlayer(int n) { ... }  
}
```

– **preferred:**

```
public class PlayerList {  
    public Iterator iterator() { ... }  
    public int size() { ... }  
    public boolean isEmpty() { ... }  
}
```



## Command: Encapsulating Control Flow:

**Name:** Command design pattern

### Problem description:

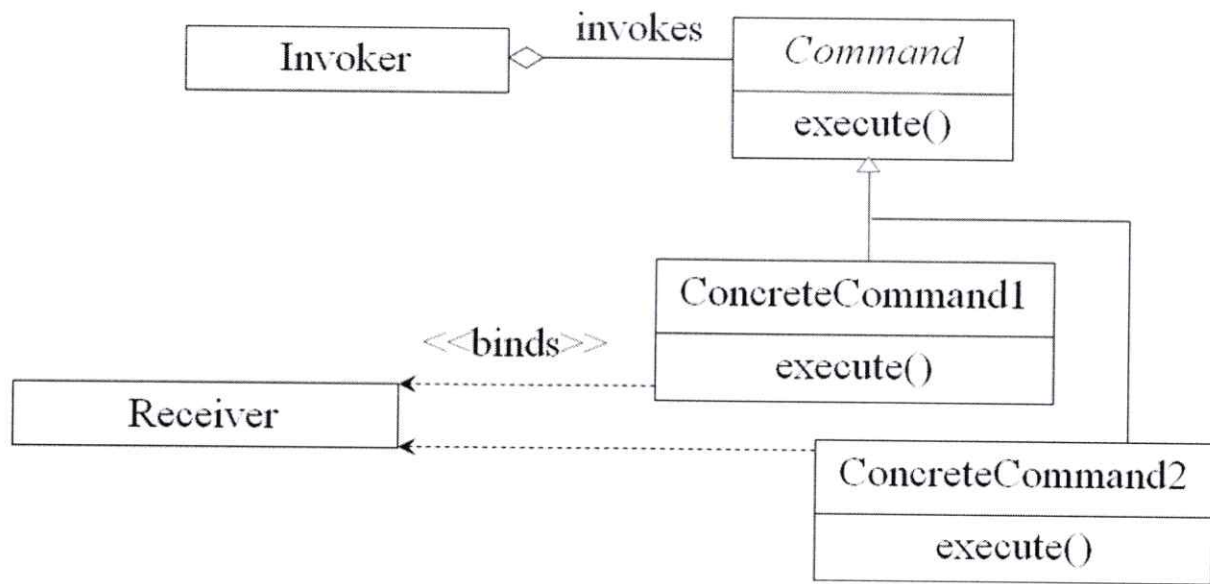
Encapsulates requests so that they can be executed, undone, or queued independently of the request.

### Solution:

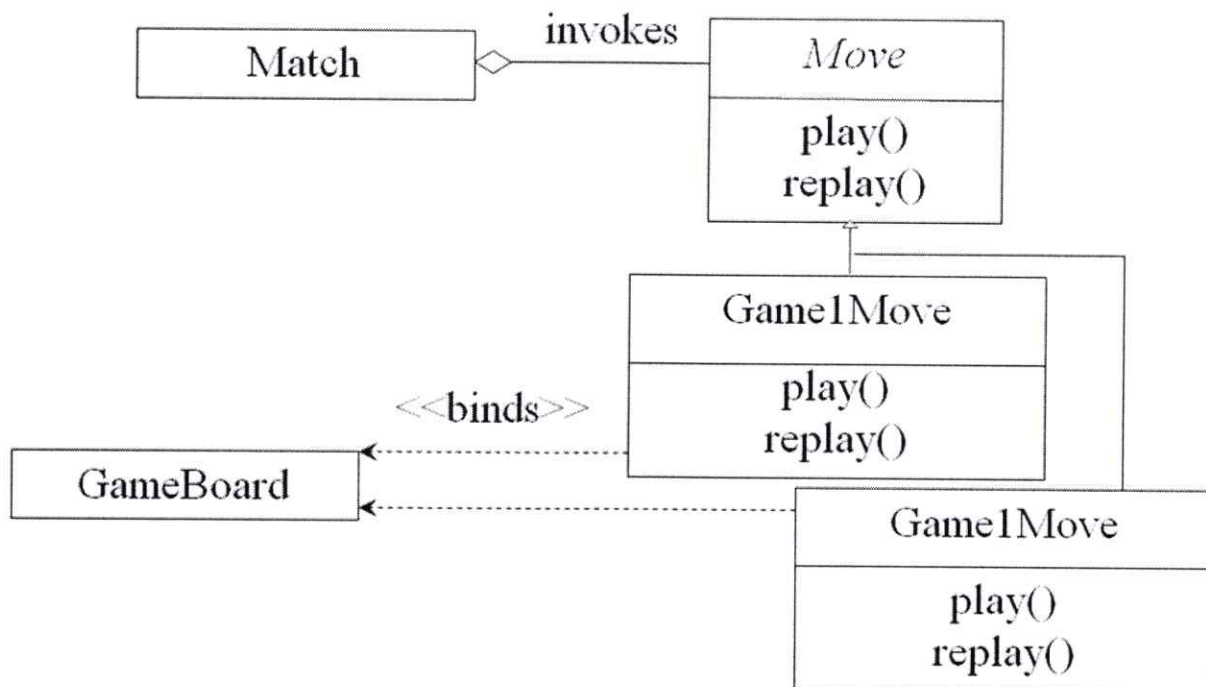
A Command abstract class declares the interface supported by all ConcreteCommands. ConcreteCommands encapsulate a service to be applied to a Receiver. The Client creates ConcreteCommands and binds them to specific Receivers. The Invoker actually executes a command.



### Command: Class Diagram



### Command: Class Diagram for Match



## Command: Consequences

### Consequences:

The object of the command (Receiver) and the algorithm of the command (ConcreteCommand) are decoupled.

Invoker is shielded from specific commands.

ConcreteCommands are objects. They can be created and stored.

New ConcreteCommands can be added without changing existing code.

- **Intent:** Given a language, interpret sentences
- **Participants:** Expressions, Context, Client
- **Implementation:** A class for each expression type  
An Interpret method on each class  
A class and object to store the global state (context)
- No support for the parsing process  
(Assumes strings have been parsed into exp trees)

### Pattern: Interpreter with Macros

- **Example:** Definite Clause Grammars
- A language for writing parsers/interpreters
- Macros make it look like (almost) standard BNF  
Command(move(D)) -> "go", Direction(D).
- Built-in to Prolog; easy to implement in Dylan, Lisp
- Does parsing as well as interpretation.
- Builds tree structure only as needed.  
(Or, can automatically build complete trees)
- May or may not use expression classes.

## Method Combination:

- Build a method from components in different classes
- Primary methods: the “normal” methods; choose the most specific one
- Before/After methods: guaranteed to run;  
No possibility of forgetting to call super  
Can be used to implement *Active Value* pattern
- Around methods: wrap around everything;  
Used to add tracing information, etc.
- Is added complexity worth it?  
Common Lisp: Yes; Most languages: No

## **Behavioural Patterns Part-II**

Part-II : Mediator, Memento, Observer

### **Behavioral Patterns (1):**

- Deal with the way objects interact and distribute responsibility
- Chain of Responsibility: Avoid coupling the sender of a request to its 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 object handles it.
- Command: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Interpreter: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

### **Behavioral Patterns (2):**

Iterator: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- Mediator: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and lets you vary their interaction independently.
- Memento: Without violating encapsulation, capture and externalize an object's internal state so that the 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 its dependents are notified and updated automatically.

### **Behavioral Patterns (3)**

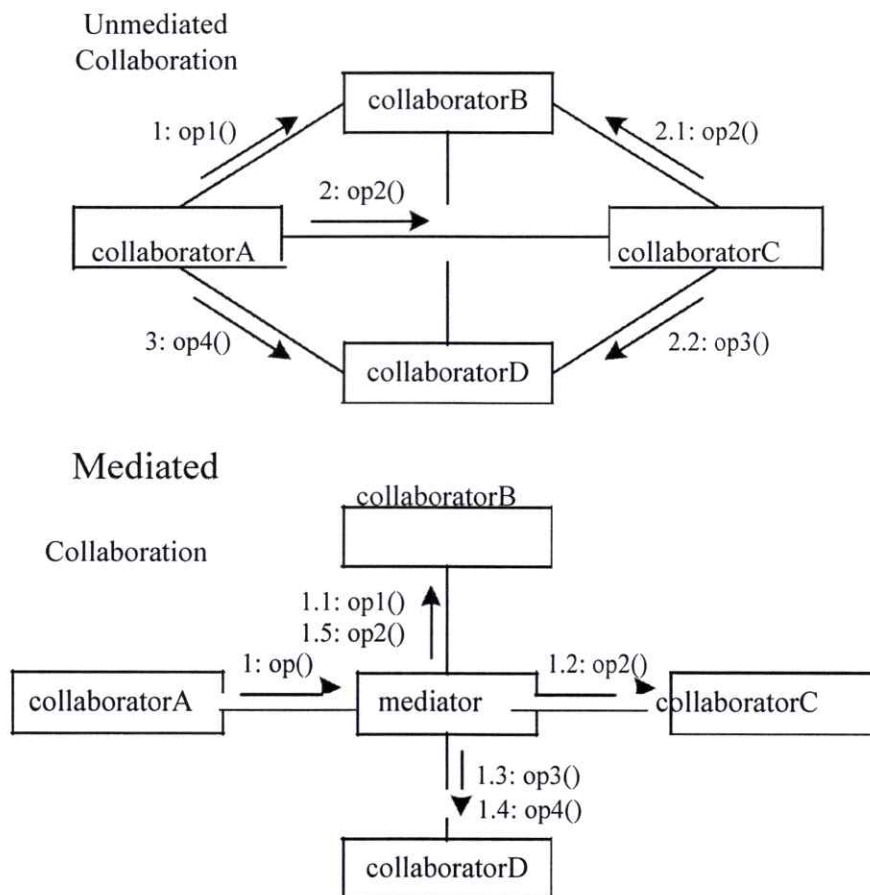
- State: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Strategy: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Template Method: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's 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.



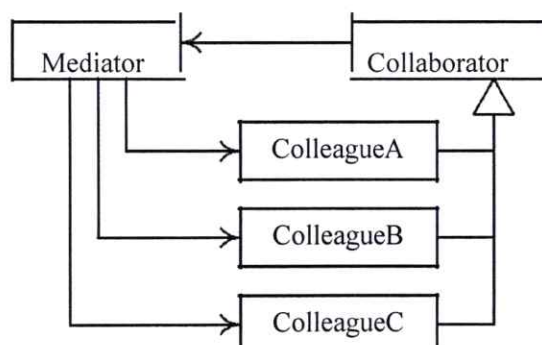
## The Mediator Pattern:

- The Mediator pattern reduces coupling and simplifies code when several objects must negotiate a complex interaction.
- Classes interact only with a mediator class rather than with each other.
- Classes are coupled only to the mediator where interaction control code resides.
- Mediator is like a multi-way Façade pattern.

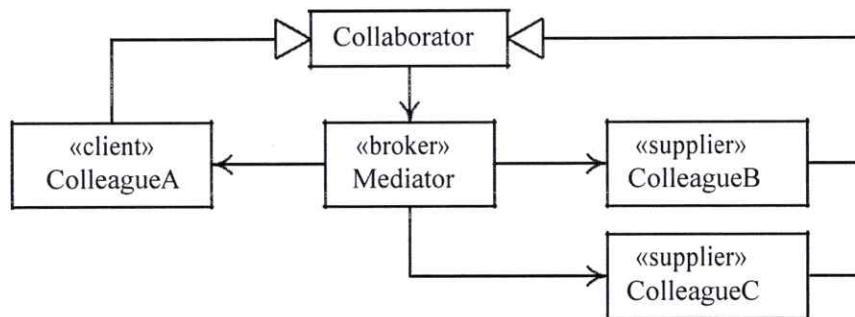
Analogy: a meeting scheduler.



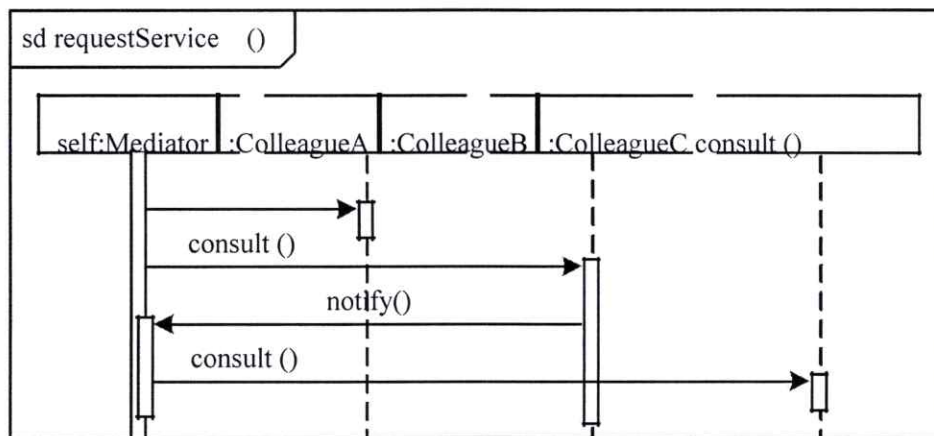
## Mediator Pattern Structure:



## Mediator as a Broker:



## Mediator Behavior:



## When to Use a Mediator:

- Use the Mediator pattern when a complex interaction between collaborators must be encapsulated to
  - Decouple collaborators,
  - Centralize control of an interaction, and
  - Simplify the collaborators.
- Using a mediator may compromise performance.

## **Mediators, Façades, and Control Styles:**

- The Façade and Mediator patterns provide means to make control more centralized.
- The Façade and Mediator patterns should be used to move from a dispersed to a delegated style, but not from a delegated to a centralized style.

### **Summary :**

- Broker patterns use a Broker class to facilitate the interaction between a Client and a Supplier.
- The Façade pattern uses a broker (the façade) to provide a simplified interface to a complex sub-system.
- The Mediator pattern uses a broker to encapsulate and control a complex interaction among several suppliers.

## **Memento Pattern**

### **Intent:**

- Capture and externalize an object's state without violating encapsulation.
- Restore the object's state at some later time.
  - Useful when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors.
  - Entrusts other objects with the information it needs to revert to a previous state without exposing its internal structure and representations.

### **Forces:**

- Application needs to capture states at certain times or at user discretion. May be used for:
  - Undue / redo
  - Log errors or events
  - Backtracking
- Need to preserve encapsulation
  - Don't share knowledge of state with other objects
- Object owning state may not know when to take state snapshot.

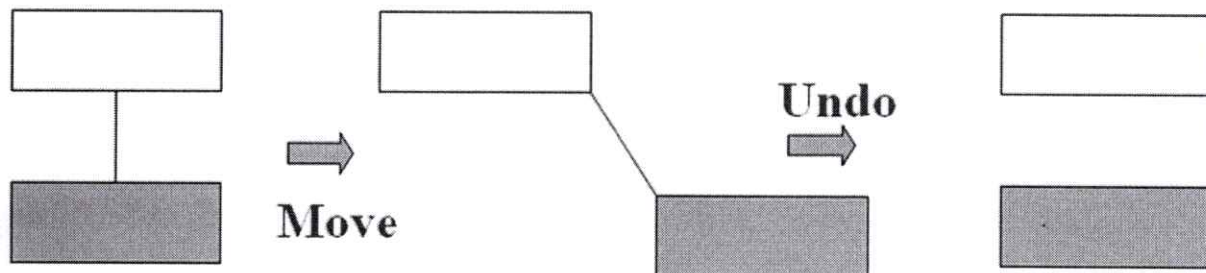
## Motivation:

- Many technical processes involve the exploration of some complex data structure.
- Often we need to backtrack when a particular path proves unproductive.

Examples are graph algorithms, searching knowledge bases, and text navigation.

Memento stores a snapshot of another object's internal state, exposure of which would violate encapsulation and compromise the application's reliability and extensibility.

A graphical editor may encapsulate the connectivity relationships between objects in a class, whose public interface might be insufficient to allow precise reversal of a move operation.



Memento pattern solves this problem as follows:

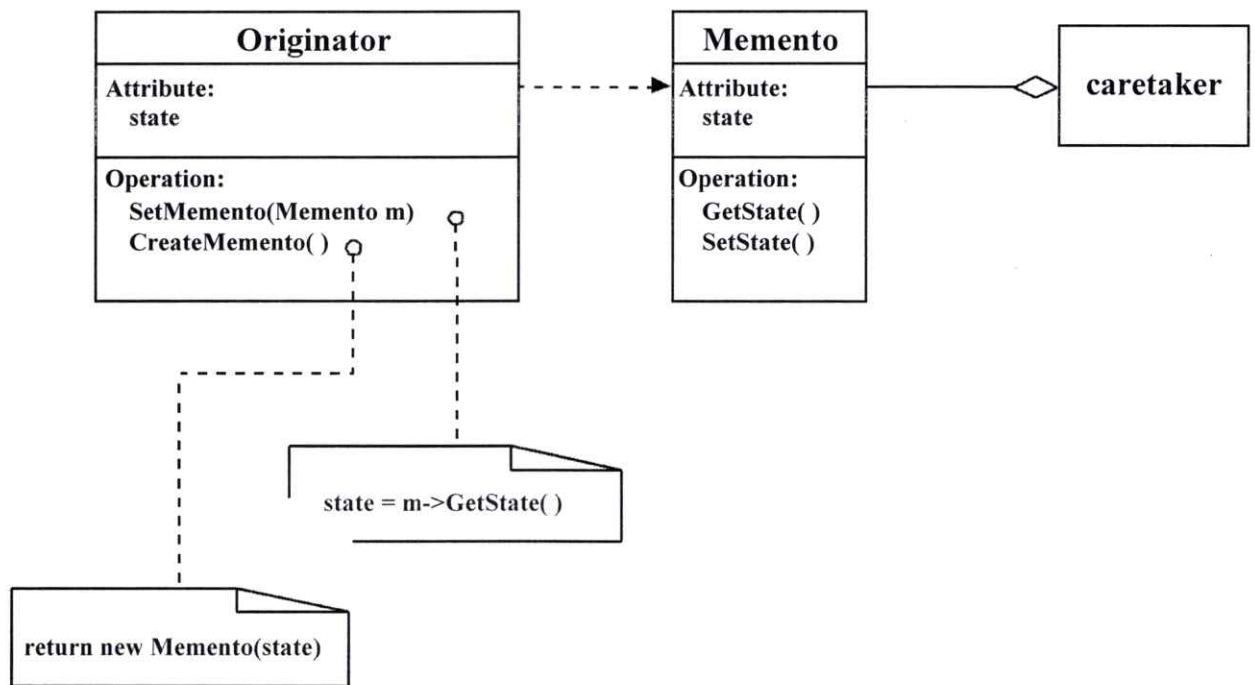
- The editor requests a memento from the object before executing *move* operation.
- Originator creates and returns a memento.
- During *undo* operation, the editor gives the memento back to the originator.
- Based on the information in the memento, the originator restores itself to its previous state.

## Applicability:

- Use the Memento pattern when:
  - A snapshot of an object's state must be saved so that it can be restored later, and direct access to the state would expose implementation details and break encapsulation.



## Structure:



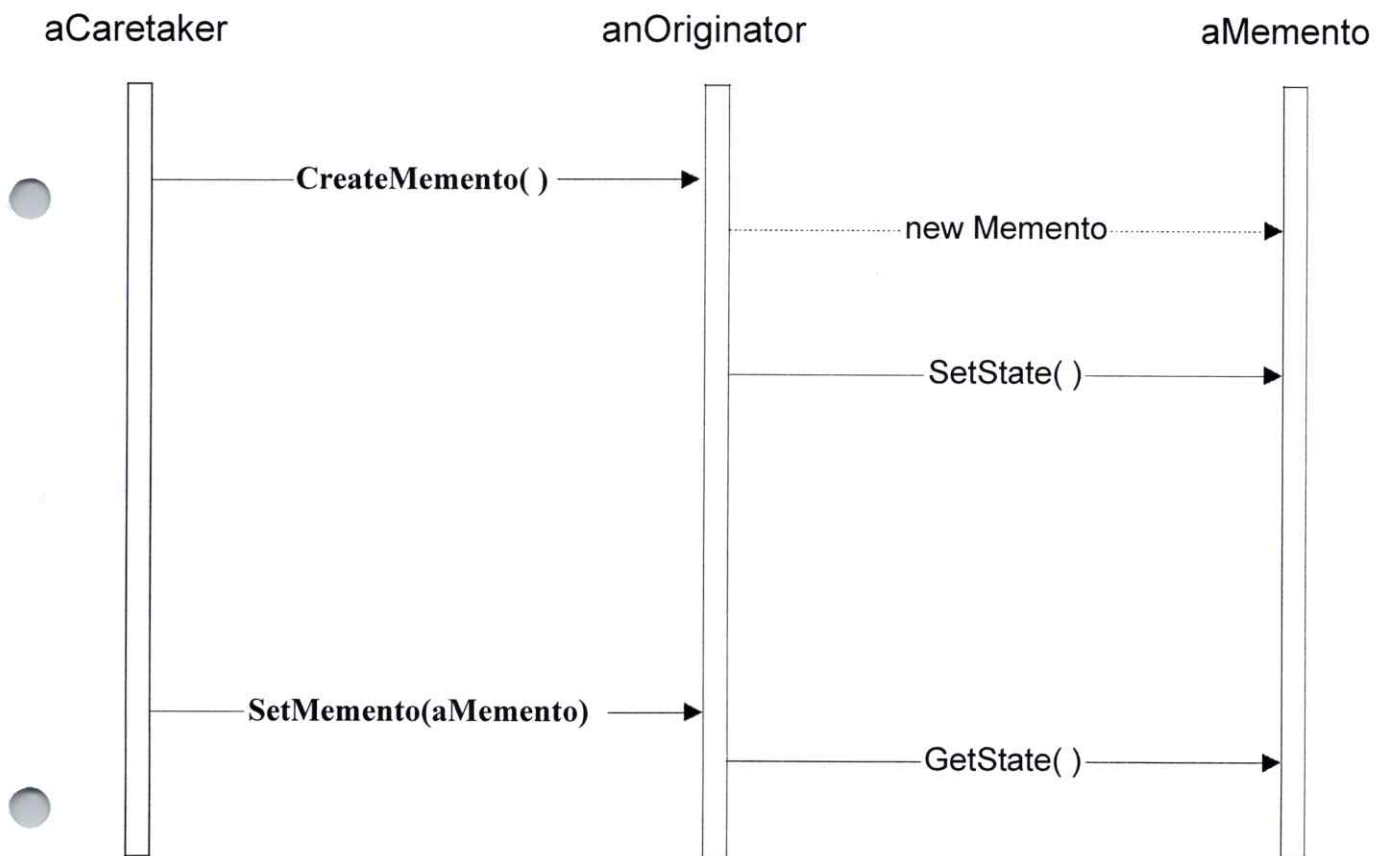
## Participants:

- **Memento**
  - Stores internal state of the Originator object. Originator decides how much.
  - Protects against access by objects other than the originator.
  - Mementos have two interfaces:
    - Caretaker sees a narrow interface.
    - Originator sees a wide interface.
- **Originator**
  - Creates a memento containing a snapshot of its current internal state.
  - Uses the memento to restore its internal state.

### Caretaker:

- Is responsible for the memento's safekeeping.
- Never operates on or examines the contents of a memento.

### Event Trace:



### Collaborations:

- A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator.
- Mementos are passive. Only the originator that created a memento will assign or retrieve its state.

## Consequences:

Memento has several consequences:

- Memento avoids exposing information that only an originator should manage, but for simplicity should be stored outside the originator.
  - Having clients manage the state they ask for simplifies the originator.
- Using mementos may be expensive, due to copying of large amounts of state or frequent creation of mementos.
- A caretaker is responsible for deleting the mementos it cares for.

A caretaker may incur large storage costs when it stores mementos.

## Implementation:

When mementos get created and passed back to their originator in a predictable sequence, then Memento can save just incremental changes to originator's state.

## Known Uses:

*Memento* is a 2000 film about Leonard Shelby and his quest to revenge the brutal murder of his wife. Though Leonard is hampered with short-term memory loss, he uses notes and tatoos to compile the information into a suspect.

## Known Use of Pattern

- Dylan language uses memento to provide iterators for its collection facility.
  - Dylan is a dynamic object oriented language using the functional style.
  - Development started by Apple, but subsequently moved to open source.

## Related Patterns

- Command

Commands can use mementos to maintain state for undo mechanisms.

- Iterator

Mementos can be used for iteration.

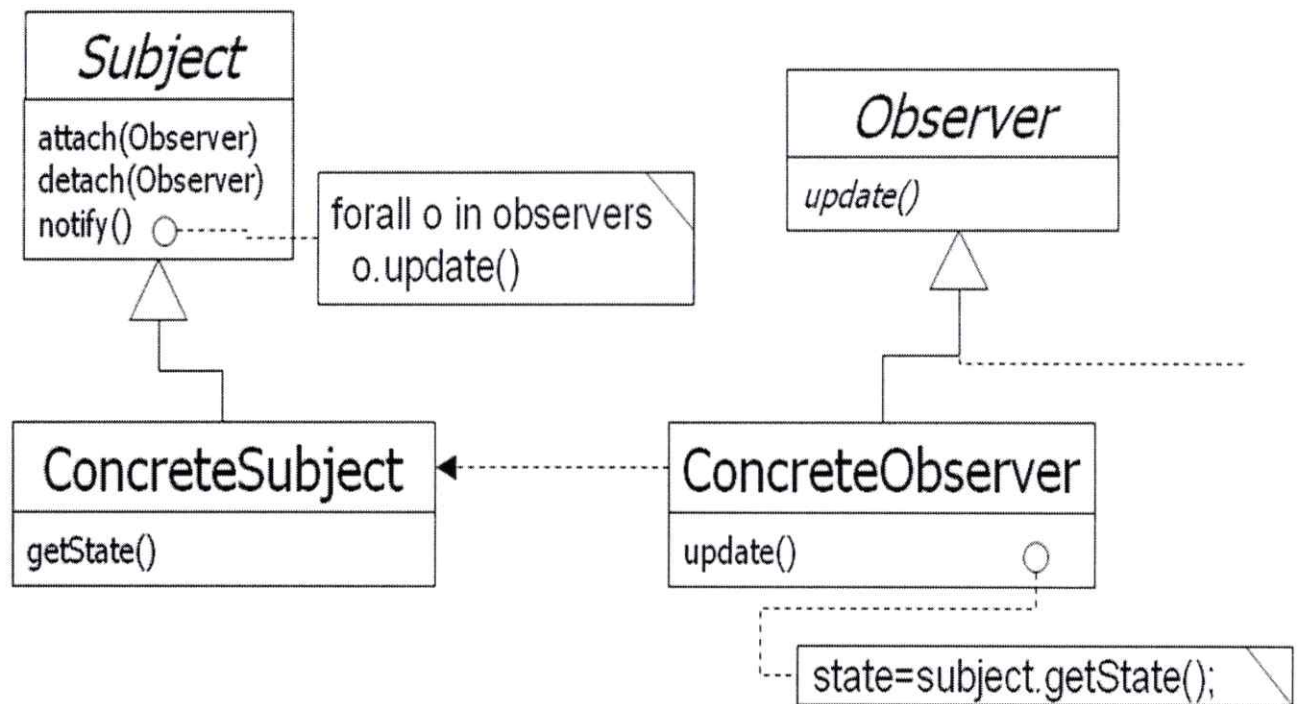
## Observer Pattern

- Define a one-to-many dependency, all the dependents are notified and updated automatically
- The interaction is known as **publish-subscribe** or **subscribe-notify**
- Avoiding observer-specific update protocol: **pull model** vs. **push model**
- Other consequences and open issues
- **Intent:**
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- **Key forces:**
  - There may be many observers
  - Each observer may react differently to the same notification
  - The subject should be as decoupled as possible from the observers to allow observers to change independently of the subject

## Observer

- Many-to-one dependency between objects
- Use when there are two or more views on the same “data”
- aka “Publish and subscribe” mechanism
- Choice of “push” or “pull” notification styles





## Observer: Encapsulating Control Flow

**Name:** Observer design pattern

### Problem description:

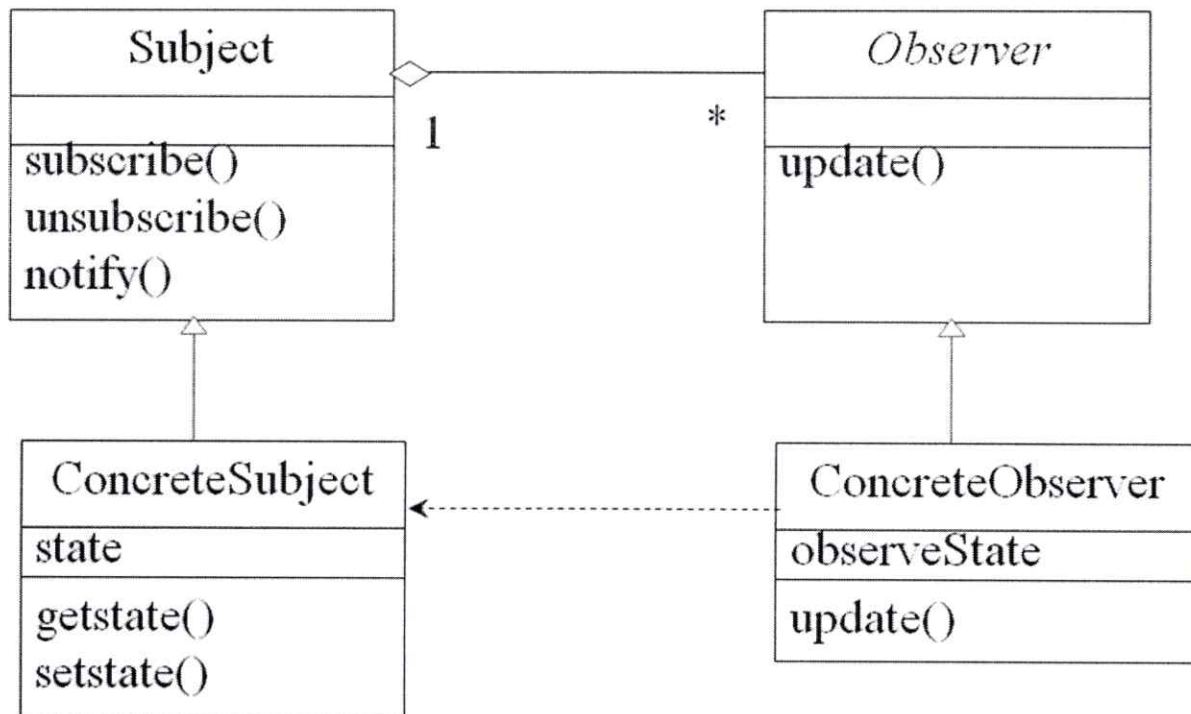
Maintains consistency across state of one Subject and many Observers.

### Solution:

A Subject has a primary function to maintain some state (e.g., a data structure). One or more Observers use this state, which introduces redundancy between the states of Subject and Observer.

Observer invokes the `subscribe()` method to synchronize the state. Whenever the state changes, Subject invokes its `notify()` method to iteratively invoke each `Observer.update()` method.

## Observer: Class Diagram



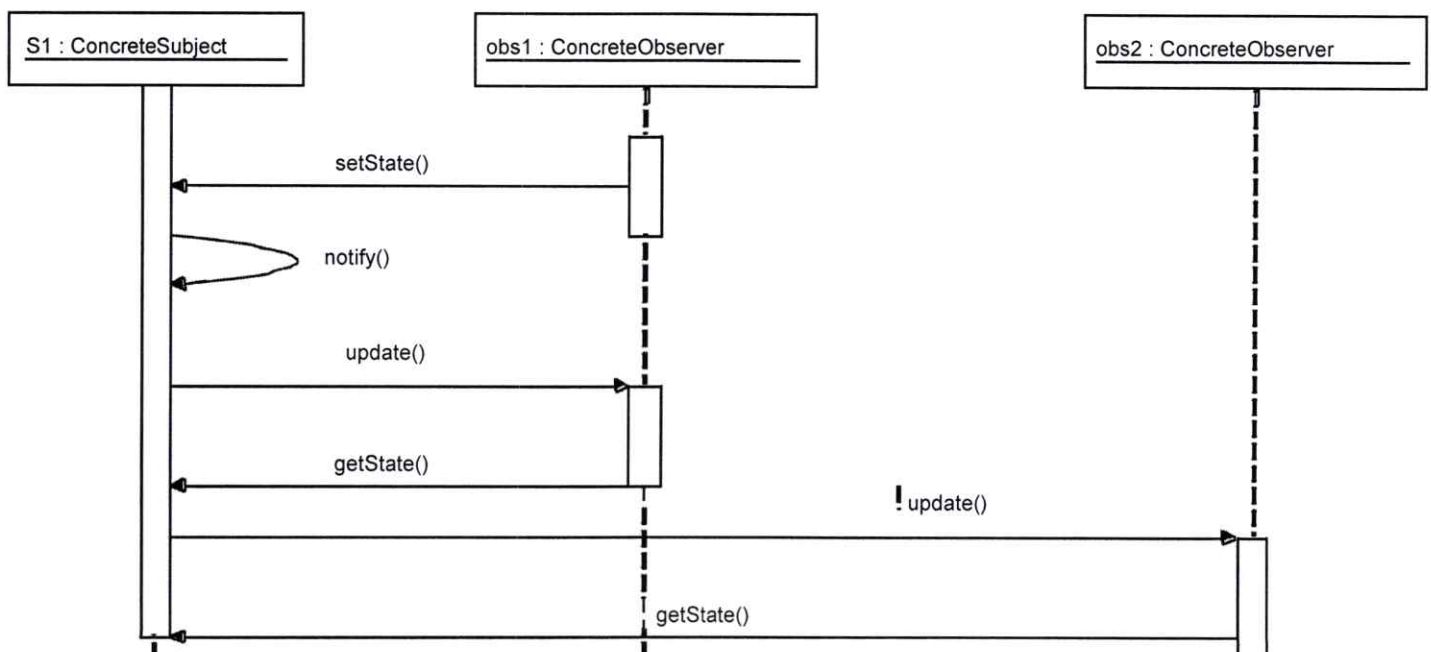
## Observer: Consequences

### Consequences:

Decouples Subject, which maintains state, from Observers, who make use of the state.

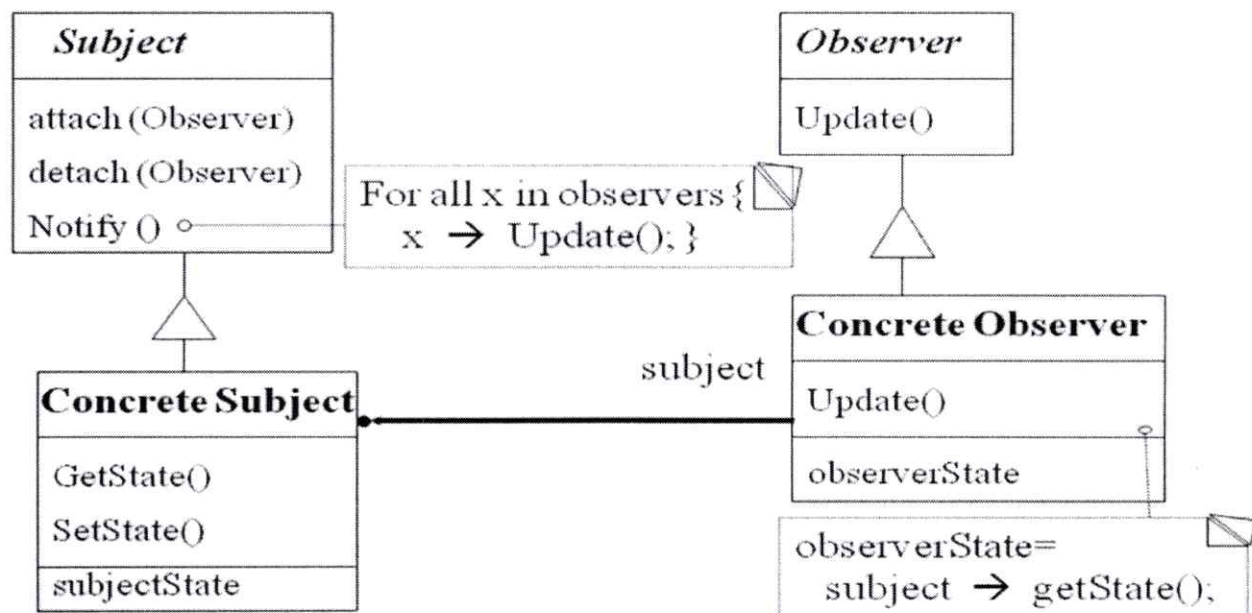
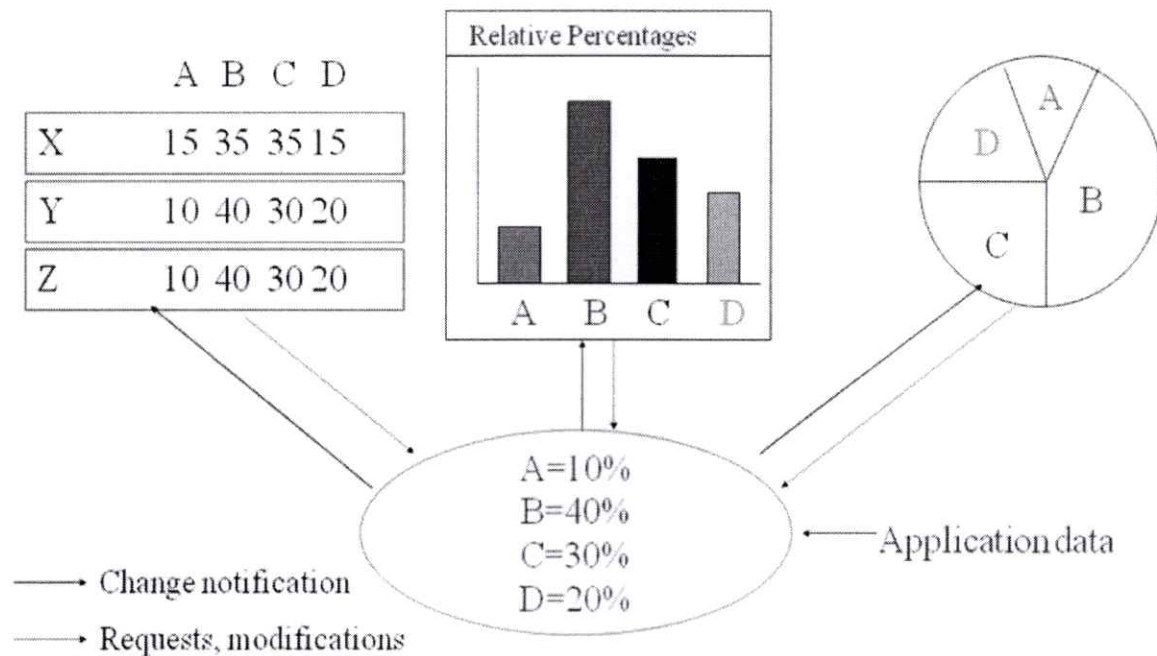
Can result in many spurious broadcasts when the state of Subject changes.

## Collaborations in Observer Pattern

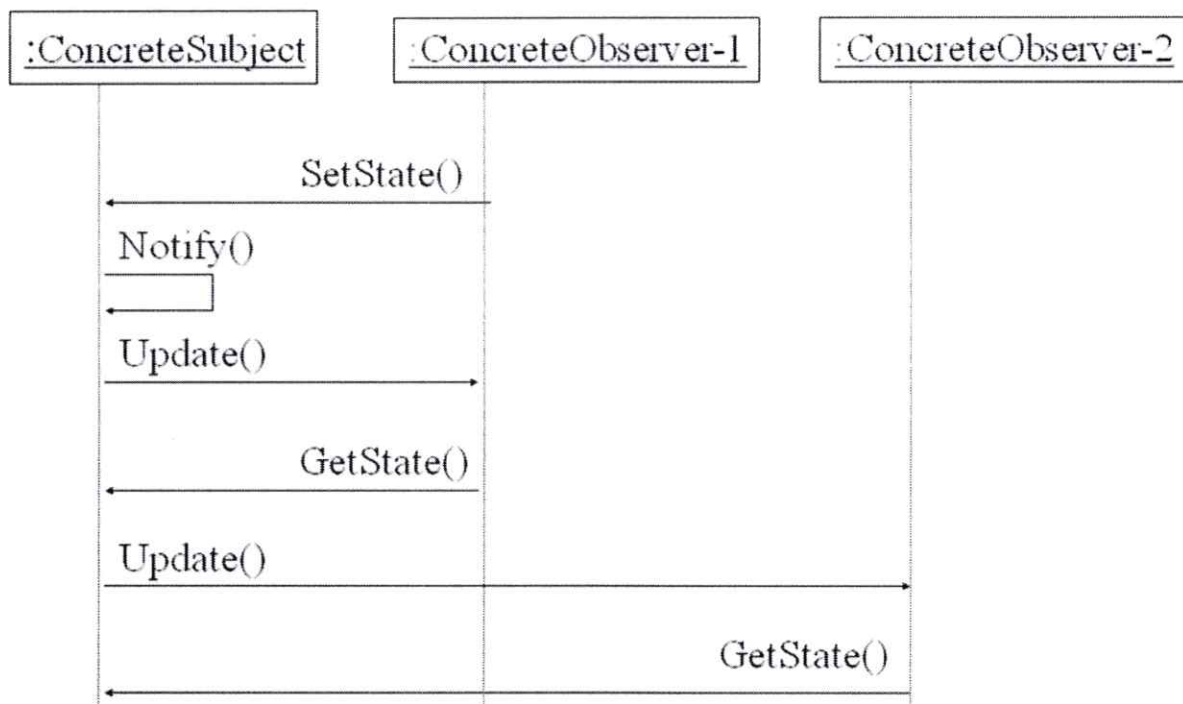


## Observer Pattern [1]

- Need to separate presentational aspects with the data, i.e. separate views and data.
- Classes defining application data and presentation can be reused.
- Change in one view automatically reflected in other views. Also, change in the application data is reflected in all views.
- Defines one-to-many dependency amongst objects so that when one object changes its state, all its dependents are notified.



## Class collaboration in Observer



### Observer Pattern: Observer code

```
class Subject;
```

```
class observer {
public:
```

```
    virtual ~observer;
```

```
    virtual void Update (Subject* theChangedSubject)=0;
```

```
protected:
```

```
    observer ();
```

```
};
```

Abstract class defining the Observer interface.

Note the support for multiple subjects.

### Observer Pattern: Subject Code



class Subject {

public:

virtual ~Subject;

virtual void Attach (observer\*);

virtual void Detach (observer\*) ;

virtual void Notify();

protected:

Subject ();

private:

List <Observer\*> \*\_observers;

};

void Subject :: Attach (Observer\* o){

    \_observers -> Append(o);

}

void Subject :: Detach (Observer\* o){

    \_observers -> Remove(o);

}

void Subject :: Notify (){

    iter.CurrentItem() -> Update(this);

}

}



Abstract class defining  
the Subject interface.

```
class ClockTimer : public Subject {  
public:
```

```
    ClockTimer();
```

```
    virtual int GetHour();
```

```
    virtual int GetMinutes();
```

```
    virtual int GetSecond();
```

```
    void Tick ();
```

```
}
```

```
ClockTimer :: Tick {
```

```
    // Update internal time keeping state.
```

```
    // gets called on regular intervals by an internal  
    timer.
```

```
    Notify();
```

```
}
```



```
class DigitalClock: public Widget, public
Observer {
public:
```

```
    DigitalClock(ClockTimer*);
```

```
    virtual ~DigitalClock();
```

```
    virtual void Update(Subject*);
```

Override Observer operation.

```
    virtual void Draw();
```

Override Widget operation.

```
private:
```

```
    ClockTimer* _subject;
```

```
}
```

```
DigitalClock::DigitalClock (ClockTimer* s) {
```

```
    _subject = s;
```

```
    _subject->Attach(this);
```

```
}
```

```
DigitalClock::~~DigitalClock() {
```

```
    _subject->Detach(this);
```

```
}
```



```

void DigitalClock ::Update (subject* theChangedSubject ) {
    If (theChangedSubject == _subject) {
        Draw();
    }
}

```

Check if this is the clock's subject.

```

void DigitalClock ::Draw () {
    int hour = _subject->GetHour();

    int minute = _subject->GeMinute(); // etc.

    // Code for drawing the digital clock.
}

```

### Observer Pattern: Main (skeleton)

```
ClockTimer* timer = new ClockTimer;
```

```
DigitalClock* digitalClock = new DigitalClock (timer);
```

### Observer Pattern: Consequences

- Abstract coupling between subject and observer. Subject has no knowledge of concrete observer classes. (What design principle is used?)
- Support for broadcast communication. A subject need not specify the receivers; all interested objects receive the notification.
- Unexpected updates: Observers need not be concerned about when then updates are to occur. They are not concerned about each other's presence. In some cases this may lead to unwanted updates.

### When to use the Observer Pattern?

- *When* an abstraction has two aspects: one dependent on the other. Encapsulating these aspects in separate objects allows one to vary and reuse them independently.
- *When* a change to one object requires changing others and the number of objects to be changed is not known.
- When an object should be able to notify others without knowing who they are. Avoid tight coupling between objects.

## UNIT-V

### Behavioral Patterns

Behavioural Patterns Part-II(cont'd) : State, Strategy, Template Method, Visitor, Discussion of Behavioural Patterns.

#### General Description

- A type of Behavioral pattern.
- Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Uses Polymorphism to define different behaviors for different states of an object.

#### **When to use STATE pattern ?**

- State pattern is useful when there is an object that can be in one of several states, with different behavior in each state.
- To simplify operations that have large conditional statements that depend on the object's state.

*if (myself = happy) then*

{

*eatIceCream();*

*....*

}

*else if (myself = sad) then*

{

*goToPub();*

*....*

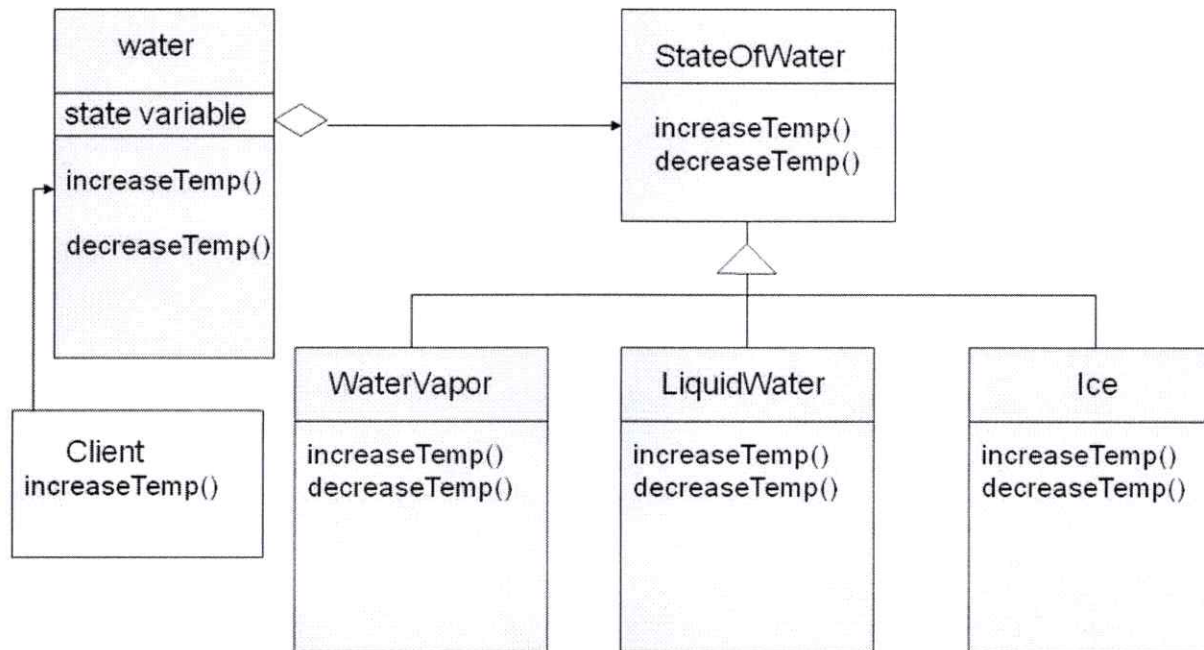
}

*else if (myself = ecstatic) then*

{

*....*

## Example I

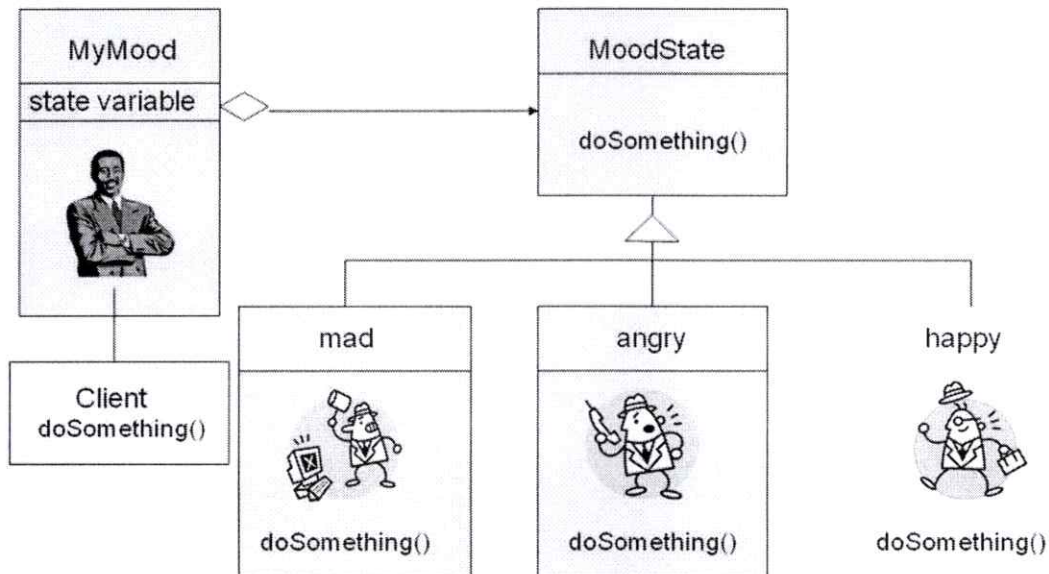


## How is STATE pattern implemented ?

- “Context” class:  
Represents the interface to the outside world.
- “State” abstract class:  
Base class which defines the different states of the “state machine”.
- “Derived” classes from the State class:  
Defines the true nature of the state that the state machine can be in.

Context class maintains a pointer to the current state. To change the state of the state machine, the pointer needs to be changed.

## Example II



## Benefits of using STATE pattern

- **Localizes all behavior associated with a particular state into one object.**
  - New state and transitions can be added easily by defining new subclasses.
  - Simplifies maintenance.
- **It makes state transitions explicit.**
  - Separate objects for separate states makes transition explicit rather than using internal data values to define transitions in one combined object.
  - **State objects can be shared.**
    - Context can share State objects if there are no instance variables.

## Food for thought...

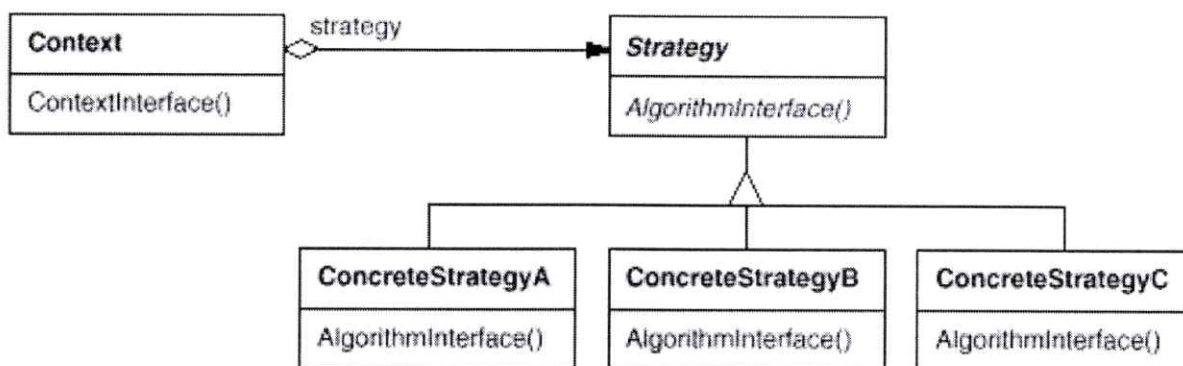
- **To have a monolithic single class or many subclasses ?**
  - Increases the number of classes and is less compact.
  - Avoids large conditional statements.
  - **Where to define the state transitions ?**



- If criteria is fixed, transition can be defined in the context.
- More flexible if transition is specified in the State subclass.
- Introduces dependencies between subclasses.
- **Whether to create State objects as and when required or to create-them-once-and-use-many-times ?**
- First is desirable if the context changes state infrequently.
- Later is desirable if the context changes state frequently.

## Pattern: Strategy

objects that hold alternate algorithms to solve a problem



## Strategy pattern

- pulling an algorithm out from the object that contains it, and encapsulating the algorithm (the "strategy") as an object
- each strategy implements one behavior, one implementation of how to solve the same problem
  - how is this different from **Command** pattern?
- separates algorithm for behavior from object that wants to act
- allows changing an object's behavior dynamically without extending / changing the object itself
- **examples:**
  - file saving/compression
  - layout managers on GUI containers
  - AI algorithms for computer game players

### Strategy example: Card player

```
// Strategy hierarchy parent
// (an interface or abstract class)
public interface Strategy {
    public Card getMove();
}

// setting a strategy player1.setStrategy(new
SmartStrategy());

// using a strategy
Card p1move = player1.move(); // uses strategy
```

### Strategy: Encapsulating Algorithms

**Name:** Strategy design pattern

**Problem description:**

Decouple a policy-deciding class from a set of mechanisms, so that different mechanisms can be changed transparently.

**Example:**

A mobile computer can be used with a wireless network, or connected to an Ethernet, with dynamic switching between networks based on location and network costs.

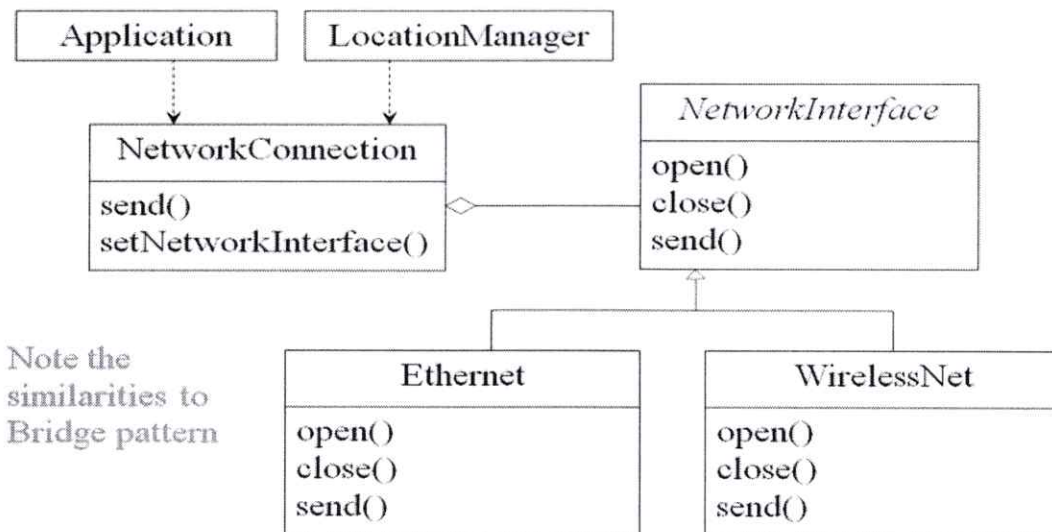
**Solution:**

A Client accesses services provided by a Context.

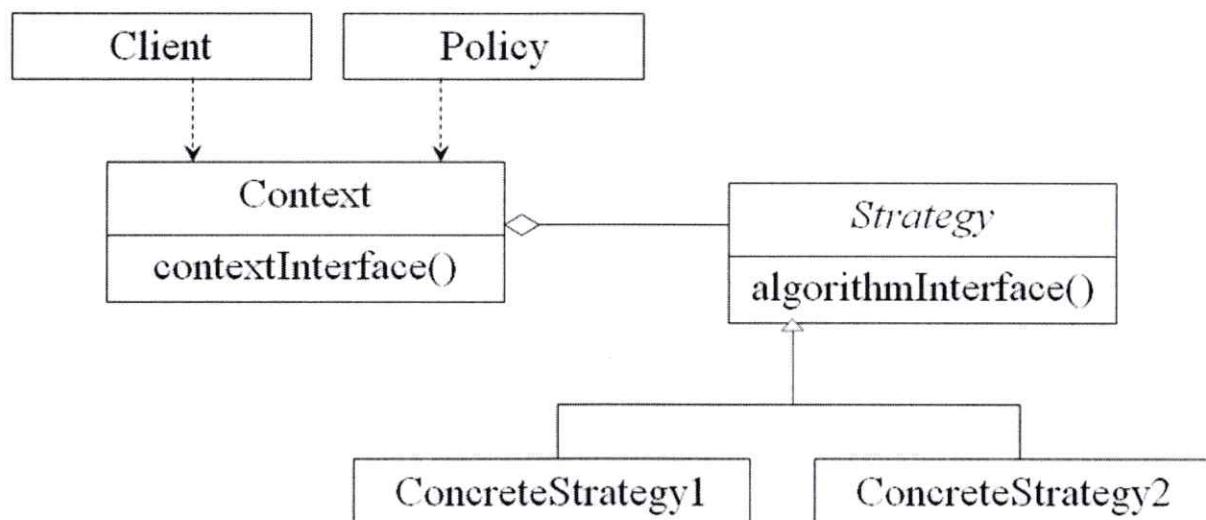
The Context services are realized using one of several mechanisms, as decided by a Policy object.

The abstract class Strategy describes the interface that is common to all mechanisms that Context can use. Policy class creates a ConcreteStrategy object and configures Context to use it.

## Strategy Example: Class Diagram for Mobile Computer



## Strategy: Class Diagram



## Strategy: Consequences

### Consequences:

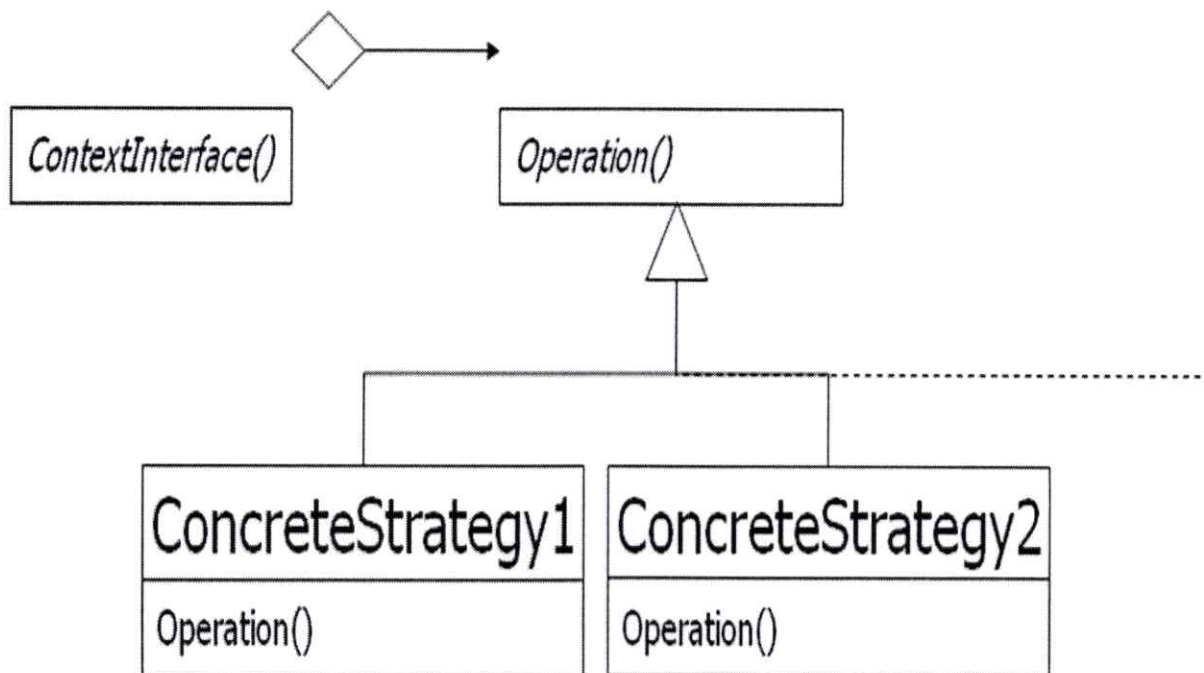
ConcreteStrategies can be substituted transparently from Context.

Policy decides which Strategy is best, given the current circumstances.

New policy algorithms can be added without modifying Context or Client.

## Strategy

- **You want to**
  - use different algorithms depending upon the context
  - avoid having to change the context or client
- **Strategy**
  - decouples interface from implementation
  - shields client from implementations
  - Context is not aware which strategy is being used; Client configures the Context
  - strategies can be substituted at runtime
  - example: interface to wired and wireless networks
- Make algorithms interchangeable---"changing the guts"
- Alternative to subclassing
- Choice of implementation at run-time
- Increases run-time complexity





## Template Method

Conducted By Raghavendar Japala

### Topics – Template Method

- Introduction to Template Method      Design Pattern
- Structure of Template Method
- Generic Class and Concrete Class
- Plotter class and Plotter Function Class

### Introduction

The DBAnimationApplet illustrates the use of an **abstract class** that serves as a template for classes with shared functionality.

An abstract class contains behavior that is common to all its subclasses. This behavior is encapsulated in nonabstract methods, which may even be declared *final* to prevent any modification. This action ensures that all subclasses will inherit the same common behavior and its implementation.

The abstract methods in such templates ensure the interface of the subclasses and require that context specific behavior be implemented for each concrete subclass.

### Hook Method and Template Method

The abstract method `paintFrame()` acts as a placeholder for the behavior that is implemented differently for each specific context.

We call such methods, *hook* methods, upon which context specific behavior may be hung, or implemented.

The `paintFrame()` hook is placed within the method `update()`, which is common to all concrete animation applets. Methods containing hooks are called *template* methods.

### Hook Method and Template Method (Con't)

The abstract method `paintFrame()` represents the behavior that is changeable, and its implementation is deferred to the concrete animation applets.

We call `paintFrame()` a hook method. Using the hook method, we are able to define the `update()` method, which represents a behavior common to all the concrete animation applets.

## Frozen Spots and Hot Spots

A template method uses hook methods to define a common behavior.

Template method describes the fixed behaviors of a generic class, which are sometimes called **frozen spots**.

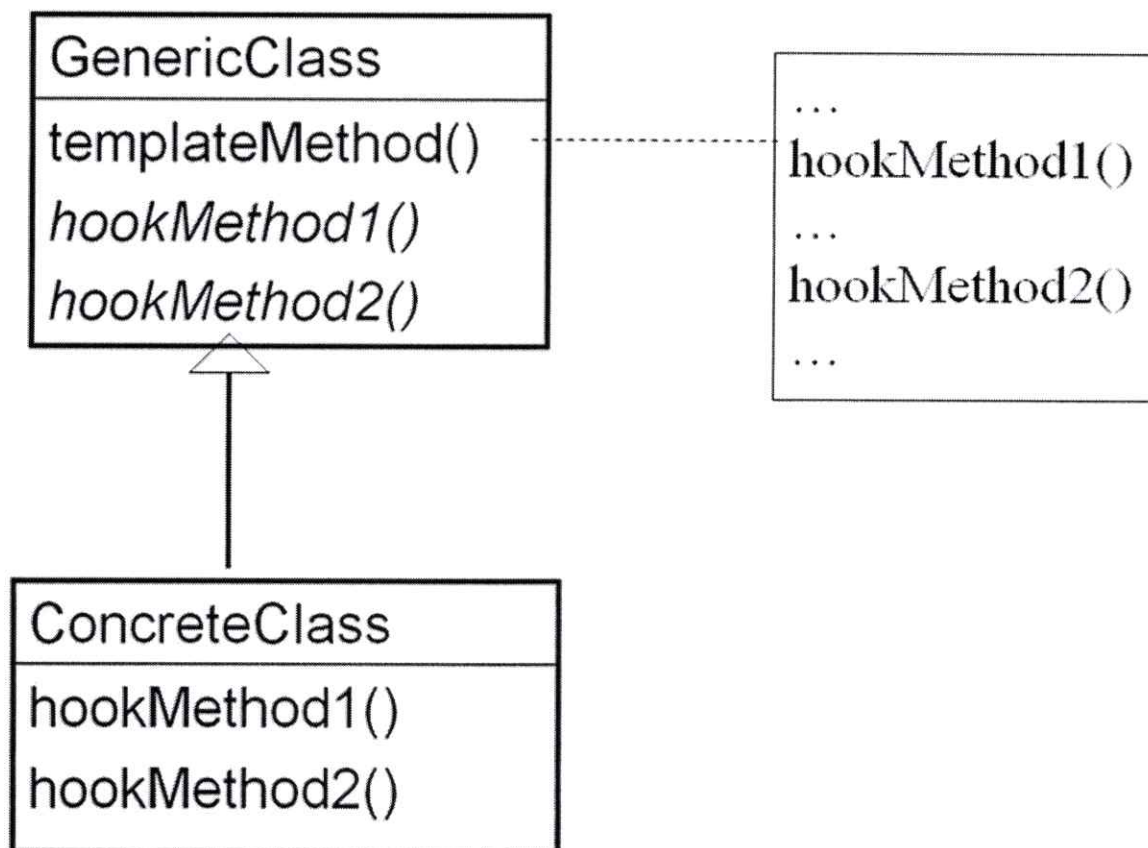
Hook methods indicate the changeable behaviors of a generic class, which are sometimes called **hot spots**.

## Hook Method and Template Method (Con't)

The abstract method `paintFrame()` represents the behavior that is changeable, and its implementation is deferred to the concrete animation applets.

We call `paintFrame()` a hook method. Using the hook method, we are able to define the `update()` method, which represents a behavior common to all the concrete animation applets.

## Structure of the Template Method Design Pattern



## Structure of the Template Method Design Pattern (Con't)

**GenericClass** (e.g., `DBAnimationApplet`), which defines abstract hook methods (e.g., `paintFrame()`) that concrete subclasses (e.g., `Bouncing-Ball2`) override to implement steps of an algorithm and implements a template method (e.g., `update()`) that defines the skeleton of an algorithm by calling the hook methods;

**ConcreteClass** (e.g., Bouncing-Ball2) which implements the hook methods (e.g., `paintFrame()`) to carry out subclass specific steps of the algorithm defined in the template method.

### Structure of the Template Method Design Pattern (Con't)

In the Template Method design pattern, *hook methods* **do not** have to be abstract.

The generic class may provide default implementations for the hook methods.

Thus the subclasses have the option of overriding the hook methods or using the default implementation.

The `initAnimator()` method in `DBAnimationApplet` is a nonabstract hook method with a default implementation.

The `init()` method is another template method.

### A Generic Function Plotter

The generic plotter should factorize all the behavior related to drawing and leave only the definition of the function to be plotted to its subclasses.

A concrete plotter `PlotSine` will be implemented to plot the function

$$y = \sin x$$

**What to Expect from Design Patterns, A Brief History, The Pattern Community An Invitation, A Parting Thought.**

**What to Expect from Design Patterns?**

- A Common Design Vocabulary.
- A Documentation and Learning Aid.
- An Adjunct to Existing Methods.
- A Target for Refactoring.

**A common design vocabulary**

1. Studies of expert programmers for conventional languages have shown that knowledge and experience isn't organized simply around syntax but in larger conceptual structures such as algorithms, data structures and idioms [AS85, Cop92, Cur89, SS86], and plans for fulfilling a particular goal [SE84].
2. Designers probably don't think about the notation they are using for recording the designing as much as they try to match the current design situation against plans, data structures, and idioms they have learned in the past.
3. Computer scientists name and catalog algorithms and data structures, but we don't often name other kinds of patterns. Design patterns provide a common vocabulary for designers to use to communicate, document, and explore design alternatives.

**A document and learning aid:**

1. Knowing the design patterns makes it easier to understand existing systems.
2. Most large object-oriented systems use this design patterns people learning object-oriented programming often complain that the systems they are working with use inheritance in convoluted ways and that it is difficult to follow the flow of control.
3. In large part this is because they do not understand the design patterns in the system learning these design patterns will help you understand existing object-oriented system.



### **An adjacent to existing methods:**

1. Object-oriented design methods are supposed to promote good design, to teach new designers how to design well, and standardize the way designs are developed.
2. A design method typically defines a set of notations (usually graphical) for modeling various aspects of design along with a set of rules that govern how and when to use each notation.
3. Design methods usually describe problems that occur in a design, how to resolve them and how to evaluate design. But then have not been able to capture the experience of expert designers.
4. A full fledged design method requires more kinds of patterns than just design patterns there can also be analysis patterns, user interface design patterns, or performance tuning patterns but the design patterns are an essential part, one that's been missing until now.

### **A target for refactoring:**

1. One of the problems in developing reusable software is that it often has to be recognized or refactored [OJ90].
2. Design patterns help you determine how to recognize a design and they can reduce a amount of refactoring need to later.

The life cycle of object-oriented software has several faces. Brian Foote identifies these phases as the prototyping expansionary, and consolidating phases [Foo92].

### **Design Patterns Applied:**

Example: An Hierarchical File System

Tree Structure → Composite

Patterns Overview

Symbolic Links → Proxy

Extending Functionality → Visitor

Single User Protection → Template Method

Multi User Protection → Singleton

User and Groups → Mediator

## A Brief History of Design Patterns

- 1979--Christopher Alexander pens The Timeless Way of Building
  - Building Towns for Dummies
  - Had nothing to do with software
- 1994--Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the Gang of Four, or GoF) publish Design patterns: Elements of Reusable Object-Oriented Software
  - Capitalized on the work of Alexander
  - The seminal publication on software design patterns.

## What's In a Design Pattern—1994

- The GOF book describes a pattern using the following four attributes:
  - The name to describes the pattern, its solutions and consequences in a word or two
  - The problem describes when to apply the pattern
  - The solution describes the elements that make up the design, their relationships, responsibilities, and collaborations
  - The consequences are the results and trade-offs in applying the pattern
- All examples in C++ and Smalltalk.

## What's In a Design Pattern – 2002

- Grand's book is the latest offering in the field and is very Java centric. He develops the GOF attributes to a greater granularity and adds the Java specifics
  - Pattern name—same as GOF attribute
  - Synopsis—conveys the essence of the solution
  - Context—problem the pattern addresses
  - Forces—reasons to, or not to use a solution
  - Solution—general purpose solution to the problem
  - Implementation—important considerations when using a solution
  - Consequences—implications, good or bad, of using a solution
  - Java API usage—examples from the core Java API
  - Code example—self explanatory
  - Related patterns—self explanatory

## Grand's Classifications of Design Pattern:

- Fundamental patterns
- Creational patterns
- Partitioning patterns
- Structural patterns
- Behavioral patterns
- Concurrency patterns

## **The Pattern Community An Invention**

☐ Christopher Alexander is the architect who first studied Patterns in buildings and communities and developed

A PATTERN LANGUAGE for generating them.

☐ His work has inspired time and again. So it's fitting worth while To compare our work to his.

☐ Then we'll look at other's work in software-related patterns.

## **Alexander's Pattern Languages**

There are many ways in which our work is like Alexander's

Both are based on observing existing systems and looking for patterns in them.

Both have templates for describing patterns although our templates are quite different)..

But there are just as many ways in which our work different.

➤ People have been making buildings for thousands of years, and there are many classic examples to draw upon. We have been making Software systems for a

Relatively short time, and few are considered classics.

➤ Alexander gives an order in which his patterns should be used; we have not.

➤ Alexander's patterns emphasize the problems they address ,

➤ where as design patterns describes the solutions in more detail.

➤ Alexander claims his patterns will generate complete buildings.

We do not claim that our patterns will generate complete programs.

When Alexander claims you can design a house simply applying his patterns one after Another ,he has goals similar to those of object-oriented design methodologies who Gives step-by-step rules for design,

In fact ,we think it's unlikely that there will ever be a compete pattern language for soft -ware.

But certainly possible to make one that is more complete.

A Parting Thought.

The best designs will use many design patterns that dovetail And intertwine to produce a greater whole.

As Alexander says:

It is possible to make buildings by stringing together pattern's,

In a rather loose way,

A building made like this , is an assembly of patterns. it is not Dense.

It is not profound. but it is also possible to put pattern's together

In such a way that many patterns overlap in the same physical

Space: the building is very dense; it has many meaning captured

In a small space; and through this density, it becomes profound.



# **Power Point Presentation**

## Design Patterns

IV B.Tech.I Semester

- Describe a recurring design structure
  - Defines a common vocabulary
  - Abstracts from concrete designs
  - Identifies classes, collaborations, and responsibilities
  - Describes applicability, trade-offs, and consequences
- Formats of pattern writers vary, but a pattern description usually has at least these four things.
  - pattern name
  - problem
  - solution
  - Consequences

## What Is a Design Pattern?

- A design pattern has 4 basic parts:
  - 1. Name
  - 2. Problem
  - 3. Solution
  - 4. Consequences and trade-offs of application
- Language- and implementation-independent
- A “micro-architecture”
- Adjunct to existing methodologies (Unified, OMT, etc.)
- No mechanical application
  - The solution needs to be translated into concrete terms in the application context by the developer

## Pattern Name

- A handle used to describe:
  - a design problem
  - its solutions
  - its consequences
- Increases design vocabulary
- Makes it possible to design at a higher level of abstraction
- Enhances communication
- *“The Hardest part of programming is coming up with good variable [function, and type] names”*

## Description of a Design Pattern

- Description of communicating objects and classes that are customized to solve a general design problem in a particular context.
- Language- & implementation-independent
- A “micro-architecture”
- Adjunct to existing methodologies (RUP, Fusion, SCRUM, etc.)

## Problem

- Describes when to apply the pattern
- Explains the problem and its context
- May describe specific design problems and/or object structures
- May contain a list of preconditions that must be met before it makes sense to apply the pattern

### Solution

- Describes the elements that make up the
  - design
  - relationships
  - responsibilities
  - collaborations
- Does not describe specific concrete implementation
- Abstract description of design problems and how the pattern solves it

### Design Pattern Template

- **Collaboration:**
  - A description of how classes and objects used in the pattern interact with each other.
- **Consequences:**
  - A description of the results, side effects, and trade offs caused by using the pattern.
- **Implementation:**
  - A description of an implementation of the pattern; the solution part of the pattern.
- **Sample Code:**
  - An illustration of how the pattern can be used in a programming language.
- **Known Uses:**
  - Examples of real usages of the pattern.
- **Related Patterns:**
  - Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns

### Consequences

- Results and trade-offs of applying the pattern
- Critical for:
  - evaluating design alternatives
  - understanding costs
  - understanding benefits of applying the pattern
- Includes the impacts of a pattern on a system's:
  - flexibility
  - extensibility
  - portability

### Goals of Design Patterns

- Codify good design
  - Distill and disseminate experience
  - Aid to novices and experts alike
  - Abstract how to think about design
- Give design structures explicit names
  - Common vocabulary
  - Reduced complexity
  - Greater expressiveness
- Capture and preserve design information
  - Articulate design decisions succinctly
  - Improve documentation
- Facilitate restructuring/refactoring
  - Patterns are interrelated
  - Additional flexibility

### Design Pattern Template

- **Pattern Name and Classification:**
  - A descriptive and unique name that helps in identifying and referring to the pattern.
- **Intent:**
  - A description of the goal behind the pattern and the reason for using it.
- **Also Known As:**
  - Other names for the pattern.
- **Motivation (Forces):**
  - A scenario consisting of a problem and a context in which this pattern can be used.
- **Applicability:**
  - Situations in which this pattern is usable; the context for the pattern.
- **Structure:**
  - A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.
- **Participants:**
  - A listing of the classes and objects used in the pattern and their roles in the design.

### Goals of Design Patterns

- Design patterns can speed up the development process by providing tested, proven development paradigms.
- Effective software design requires considering issues that may not become visible until later in the implementation.
- Reusing design patterns helps to prevent subtle issues that can cause major problems, and it also improves code readability for coders and architects who are familiar with the patterns.

### Why Design Patterns?

- Designing OO software is hard
- Designing *reusable* OO software – harder
- Experienced OO designers make good design
- New designers tend to fall back on non-OO techniques used before
- Experienced designers know something – what is it?

### OO Design

- OOD methods emphasize design notations
  - Fine for specification, documentation
- But OOD is more than just drawing diagrams
  - Good draftsmen ✍ good designers
- Good OO designers rely on lots of experience
  - At least as important as syntax
- Most powerful reuse is *design* reuse
  - Match problem to design experience

### Why Design Patterns?

- Expert designers know *not* to solve every problem from first principles.
- They *reuse* solutions.
- These patterns make OO designs more flexible, elegant, and ultimately reusable.

### Recurring OO Design Structures

- OO systems exhibit recurring structures that promote
  - abstraction
  - flexibility
  - modularity
  - elegance
- find pertinent objects and factor them into classes at the right granularity.
- define class interfaces and inheritance hierarchies
- Establish key relationship among classes.

### Two Major Principles of Object-Oriented Design

- Program to an interface, not an implementation.
- Favor object compositions over class inheritance.

### OO Design is hard

- The design should be specific to the problem at hand, but also general enough to address future problems and requirements.
- Therein lies valuable design knowledge, the problem is
  - Capturing,
  - Communicating,
  - applying this knowledge



### Expert designers use patterns

- When they find a good solution, they use it again and again.
- The design patterns solve specific design problems and make OO designs more flexible, elegant, and ultimately reusable.
- A designer who is familiar with patterns can apply them immediately to design problems

### Types of Patterns

- **Creational Patterns**
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- **Structural Patterns**
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy
- **Behavioral Patterns**
  - Chain of Responsibility
  - Command
  - Interpreter
  - **Iterator**
  - Mediator
  - Memento
  - Observer
  - State
  - **Strategy**
  - Template Method
  - Visitor

### Why Study Design Patterns?

- Can reuse solutions.
  - Gives us a head start
  - Avoids the unanticipated things later
  - No need to reinvent the wheel
- Establish common terminology
  - Design patterns provide a common point of reference
  - Easier to say, "We need some Strategies here."
- Provide a higher level prospective.
  - Frees us from dealing with the details too early.

### Design Space for Patterns

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

### Types of Patterns

- **Creational patterns:**
  - Deal with initializing and configuring classes and objects
- **Structural patterns:**
  - Deal with decoupling interface and implementation of classes and objects
  - Composition of classes or objects
- **Behavioral patterns:**
  - Deal with dynamic interactions among societies of classes and objects
  - How they distribute responsibility

# **Semester End Question Papers**





# Vidya Jyothi Institute of Technology (Autonomous)

(Accredited by NAAC & NBA, Approved By A.I.C.T.E., New Delhi, Permanently Affiliated to JNTU, Hyderabad)  
(Aziz Nagar, C.B.Post, Hyderabad -500075)

R15

Subject Code:A18543

B.Tech. IV Year II Semester Examinations - MAY 2019

SUBJECT :DESIGN PATTERNS

BRANCH : CSE &IT

Time: 3 Hours

Max. Marks:75

Note: This question paper contains two *Parts A and B*.

*Part A* is compulsory which carries 25 Marks. Answer all the questions.

*Part B* consists of 5 questions. Answer all the questions.

Bloom'S Level:

Remember	L1	Analyze	L4
Understand	L2	Evaluate	L5
Apply	L3	Create	L6

## PART - A

ANSWER ALL THE QUESTIONS		B.L	25M
1	What is a design Pattern?	L2	2M
2	Explain Design Patterns in Small Talk MVC.	L2	3M
3	List different types of Creational Patterns.	L3	2M
4	What is the intent of a Builder Pattern?	L1	3M
5	Write the motivation of Decorator Pattern?	L4	2M
6	Sketch the structure of Adaptor Pattern.	L3	3M
7	What are the known uses of Iterator Pattern?	L1	2M
8	Write benefits and drawbacks of Mediator Pattern.	L2	3M
9	Write a short note on the history of Design Patterns.	L2	2M
10	Describe the patterns in a software.	L1	3M

## PART - B

ANSWER ALL THE QUESTIONS		B.L	50M
11.i.a)	Write the general template used for describing the design patterns.	L2	5M
b)	How to select a design pattern? Explain in detail.	L4	5M
[OR]			
ii.a)	Describe the formatting in Lexi's Design.	L2	5M
b)	Explain the Spelling Checking and Hyphenation Design problem in detail.	L1	5M
12.i.a)	What are the consequences of Builder pattern? Explain Builder Pattern implementation	L3	5M
b)	Why Singleton Pattern uses a special method to instantiate objects? Discuss.	L2	5M
[OR]			
ii.a)	Draw the structure of Abstract Factory pattern and list out its participants.	L3	5M
b)	Discuss about the implementation issues of Prototype Pattern.	L3	5M
13.i.a)	Explain the sample code of Flyweight Pattern.	L5	5M
b)	Discuss about the participants of Adaptor Pattern and explain the functions of each.	L2	5M
[OR]			
ii.a)	Write the motivation and Consequences of Facade Pattern.	L3	5M
b)	Explain about the intent,uses and related patterns of Decorator and Proxy.	L2	5M
14.i.a)	Explain About motivation of Mediator Pattern.	L3	5M
b)	What are the issues to be considered while applying Observer Pattern.	L4	5M
[OR]			
ii.a)	What are the consequences of chain of responsibility Pattern? Explain.	L2	5M
b)	Write some of the benifits and liabilities of Visitor Pattern.	L2	5M
15.i.a)	Illustrate Alexander's Pattern Language.	L3	5M
b)	Explain about target refactoring approach method.	L2	5M
[OR]			
ii.a)	Explain what to expect from Design Patterns.	L4	5M
b)	What can you do if you are interested in Patterns?	L4	5M

\*\*\*VJIT(A)\*\*\*

# Extra Topics



## **Topics Covered Under Content beyond Syllabus (Gap Analysis):**

### **Cloud Design Patterns**

#### **Centralized Remote Administration**

**How can diverse administrative tasks and controls be consolidated for central remote access by cloud consumers?**



#### **Problem**

Cloud consumers can end up having to manage a range of diverse cloud-based IT resources, each with distinct administrative functions. The disparity in user-interfaces and reporting features can make remote administration burdensome and prone to human error.

#### **Solution**

The cloud provider can consolidate diverse management features for different IT resources into a single, custom portal that standardizes administrative controls as well as providing cross-IT resource reporting features.

#### **Application**

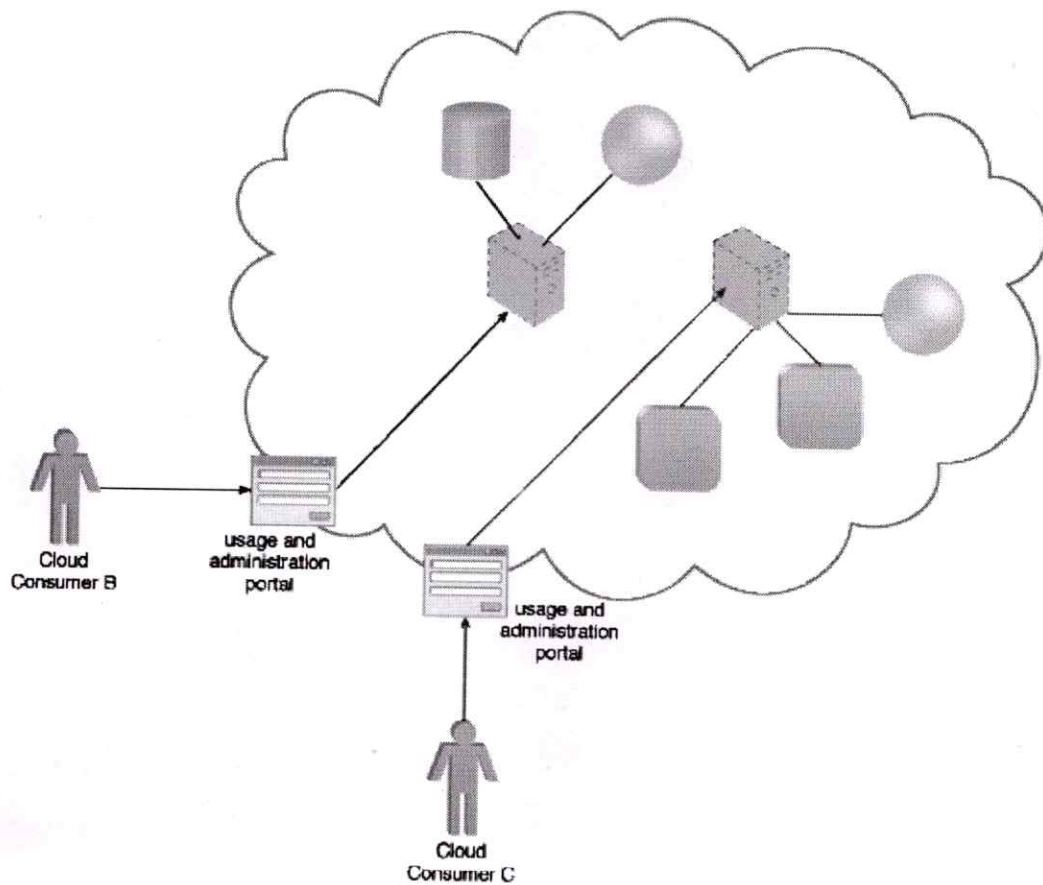
A usage and administration portal is developed by the cloud provider to interface with systems and APIs offered by back-end products, IT resources and mechanisms, and to further support different levels of access based on pre-assigned permissions.

#### **Mechanisms**

Audit Monitor, Billing Management System, Cloud Usage Monitor, Logical Network Perimeter, Multi-Device Broker, Pay-Per-Use Monitor, Remote Administration System, Resource Management System, SLA Monitor

#### **Compound Patterns**

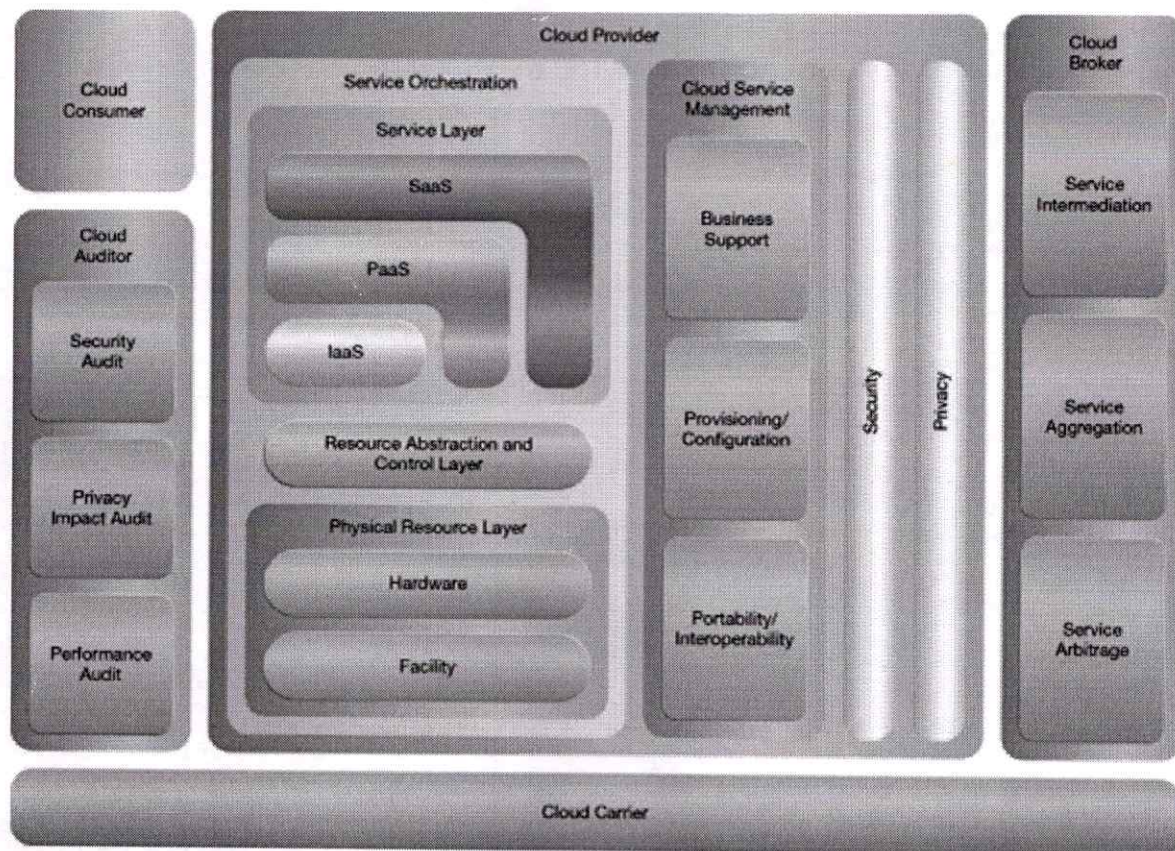
Burst In, Burst Out to Private Cloud, Burst Out to Public Cloud, Cloud Authentication, Elastic Environment, Infrastructure-as-a-Service (IaaS), Isolated Trust Boundary, Multitenant Environment, Platform-as-a-Service (PaaS), Private Cloud, Public Cloud, Resilient Environment, Resource Workload Management, Secure Burst Out to Private Cloud/Public Cloud, Software-as-a-Service (SaaS)



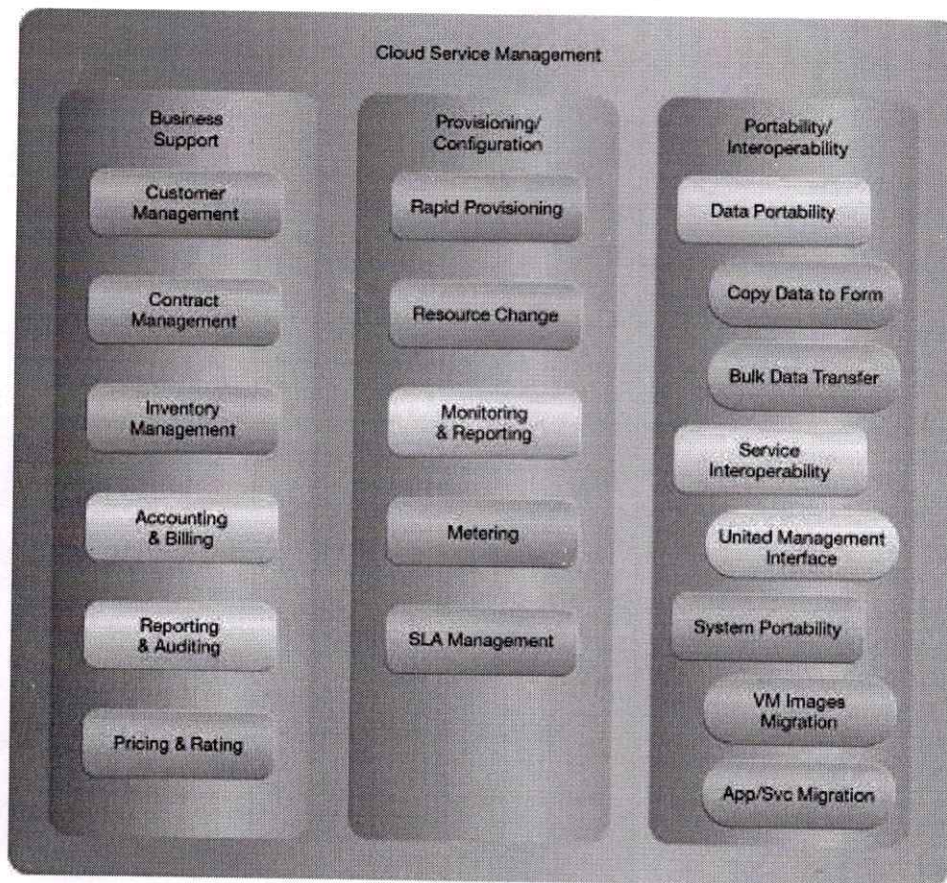
*Cloud Consumers B and C can access and manage their provisioned IT resources using the usage and administration portal.*

#### *NIST Reference Architecture Mapping*

This pattern relates to the highlighted parts of the NIST reference architecture, as follows:









# **Innovations in Teaching & Learning**



# Vidya Jyothi Institute of Technology

(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi, permanently affiliated to JNTUH)

(An AUTONOMOUS Institution)

## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

### **Innovative /Student Centric Teaching Method Form**

**Innovative Technique implemented: Online Quiz1**

**Subject:** Design Patterns

**Topic:** Unit-1

**Name of the Faculty:** R.R.S.Ravi Kumar

**Class/ Section:** IV B.Tech II-Sem CSE

**Implementation:**

**Objective:** "To assess the understanding of the subject"

**Out Class activity:** Send link of the quiz exam to student groups

**In class activity:** All the students write the test and submit answers.

**Outcome:**

### DESIGN PATTERNS UNIT-I

Form description

Email \*

Valid email

This form is collecting emails. [Change settings](#)

Roll Number

\*

Short answer text

# Design Patterns

Unit II Online Assessment Test

Which of the following describes the Builder pattern correctly?

- ☐ This pattern builds a complex object using simple objects and using a step by step approach
- ☐ This pattern refers to creating duplicate object while keeping performance in mind
- ☐ This pattern is used when creation of object directly is costly
- ☐ This pattern is used when we need to decouple an abstraction from its implementation so that the two can

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

- ☐ Prototype Pattern

*R.R.S. Ravi Kumar*  
(Course Coordinator)

  
(HOD-CSE)

**Innovative Technique implemented: Online Quiz 2**

**Subject:** Design Patterns

**Topic:** Unit-2

**Name of the Faculty:** R.R.S.Ravi Kumar

**Class/ Section:** IV B.Tech II-Sem CSE

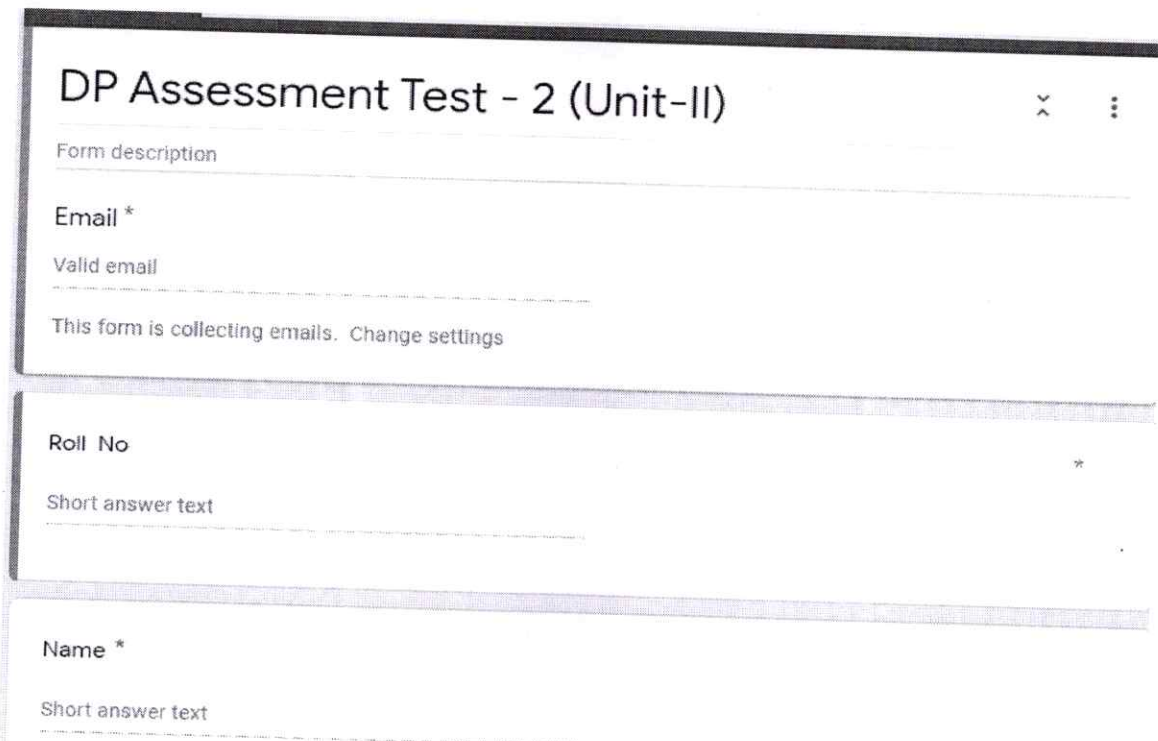
**Implementation:**

**Objective:** "To assess the understanding of the subject"

**Out Class activity:** Send link of the quiz exam to student groups

**In class activity:** All the students write the test and submit answers.

**Outcome:**



The screenshot shows a Google Form titled "DP Assessment Test - 2 (Unit-II)". The form includes a description field, an email field with a validation message "Valid email", and a note "This form is collecting emails. Change settings". Below this is a section for "Roll No" with a "Short answer text" input field. The final section is for "Name" with a "Short answer text" input field. The form is displayed in a standard Google Forms interface with a dark header bar.

DP Assessment Test - 2 (Unit-II)

Form description

Email \*

Valid email

This form is collecting emails. Change settings

Roll No

Short answer text

Name \*

Short answer text



Choose: Gang of Four in Design Patterns

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

Charles Babbage, Dennis Ritchie, Ralph Johnson, Grady Booch

Bjarne Stroustrup, Richard Helm, Dennis Ritchie, Grady Booch

Steve Jobs, Mark Zuckerberg, Bill Gates, Grady Booch

Find the Essential Elements in Design Patterns

Pattern Name, Problem, Solution, Consequences

Inheritance, Implementation, Iterator, Interpreter

Problem, Consequences, Sample code, Collaborations

Visitor Pattern Name, Solution, Consequences

R-R-S-Ravi Kumar  
(Course Coordinator)

  
(HOD-CSE)

# **Assessment Sheet – CO Wise (Direct Attainment)**

# Vidya Jyothi Institute of Technology(Autonomous)

Department of Computer Science Engineering

BATCH: 2016-20

Academic Year: 2019-20

IV B.Tech- II Sem

Course:DP

Faculty:

S.No	Reg.No	MID I Threshold 60%						MID II Threshold 60%						Threshold 60%		
		ASMI - I (5)	PART-A			PART-B			ASMI - II (5)	PART-A			PART-B			
			Q1(2M)	Q2(2M)	Q3 B (2M)	Q4(5M)	Q5(5M)	Q6(4M)		Q1(2M)	Q2(2M)	Q3 A (2M)	Q4(4M)		Q5(5M)	Q6(5M)
1	16911A0501	5	2	2	2	4	4	5	2	2	2	1	4	4	69	
2	16911A0502	5	2	2	2	2	3	5	2	2	2	1	3	2	53	
3	16911A0503	5	2	2	2	2	2	5	2	2	2	1	2	2	71	
4	16911A0505	5	2	2	0	2	2	5	2	2	2	1	2	2	70	
5	16911A0506	5	2	2	2	2	4	5	2	2	2	1	4	2	62	
6	16911A0507	5	2	2	2	3	3	5	2	2	2	2	3	3	67	
7	16911A0508	5	2	2	2	3	3	5	2	2	2	2	2	3	65	
8	16911A0509	5	2	2	2	4	4	5	2	2	2	2	4	4	68	
9	16911A0510	5	2	2	2	5	5	5	1	2	2	2	4	5	59	
10	16911A0511	5	2	2	2	3	3	5	1	2	2	2	1	3	67	
11	16911A0512	5	2	2	2	1	3	5	1	2	2	2	3	1	71	
12	16911A0513	5	2	2	2	3	3	4	5	2	2	2	4	3	70	
13	16911A0514	5	2	2	2	4	2	4	5	2	2	2	4	4	60	
14	16911A0515	5	2	2	2	5	3	4	5	2	2	2	4	5	69	
15	16911A0516	5	2	2	2	3	3	3	5	2	2	2	1	3	62	
16	16911A0517	5	2	2	2	1	3	3	5	2	2	2	3	1	69	
17	16911A0518	5	2	2	2	3	3	4	5	2	2	2	4	3	71	
18	16911A0519	5	2	2	2	4	5	4	5	2	2	2	4	4	73	
19	16911A0520	5	2	2	2	3	3	3	5	1	2	2	3	3	69	
20	16911A0521	5	2	2	2	3	3	2	5	1	2	2	2	3	71	
21	16911A0522	5	2	2	2	4	4	5	2	2	2	2	4	4	67	
22	16911A0523	5	2	2	2	3	3	3	5	2	2	2	3	3	58	
23	16911A0524	5	2	2	2	4	4	4	5	2	2	1	4	4	68	
24	16911A0525	5	2	2	2	2	2	3	5	2	2	1	3	2	68	
25	16911A0526	5	2	2	2	2	2	2	5	2	2	2	2	2	67	
26	16911A0527	5	2	2	0	2	2	5	2	2	2	1	2	2	70	
27	16911A0528	5	2	2	2	2	4	4	5	2	2	1	4	2	69	
28	16911A0529	5	2	2	2	3	3	3	5	2	2	2	3	3	70	
29	16911A0530	5	2	2	2	3	3	2	5	2	2	2	2	3	70	
30	16911A0532	5	2	2	2	4	4	4	5	2	2	2	4	4	62	
31	16911A0533	5	2	2	2	5	5	4	5	1	2	2	4	5	70	
62	16911A0534	5	2	2	2	3	3	3	5	1	2	2	1	3	62	
58	16911A0535	5	2	2	2	1	3	3	5	1	2	2	3	1	65	
34	16911A0536	5	2	2	2	3	3	4	5	2	2	2	4	3	70	
35	16911A0537	5	2	2	2	4	2	4	5	2	2	2	4	4	62	
36	16911A0538	5	2	2	2	5	3	4	5	2	2	2	4	5	59	
37	16911A0540	5	2	2	2	3	3	3	5	2	2	2	1	3	63	
38	16911A0541	5	2	2	2	1	3	3	5	2	2	2	3	1	55	
39	16911A0542	5	2	2	2	3	3	4	5	2	2	2	4	3	59	
40	16911A0543	5	2	2	2	4	5	4	5	2	2	2	4	4	60	

41	16911A0544	5	2	2	2	2	2	3	3	3	3	5	1	2	2	2	3	3	3	3	52
42	16911A0545	5	2	2	2	2	2	3	3	3	2	5	1	2	2	2	2	3	3	3	53
43	16911A0546	5	2	2	2	2	2	4	4	4	4	5	2	2	2	2	4	4	4	4	60
44	16911A0548	5	2	2	2	2	2	3	3	3	3	5	2	2	2	2	3	3	3	3	61
45	16911A0549	5	2	2	2	2	2	4	4	4	4	5	2	2	2	1	4	4	4	4	39
46	16911A0550	5	2	2	2	2	2	2	2	2	3	5	2	2	2	1	3	2	2	2	47
47	16911A0551	5	2	2	2	2	2	2	2	2	2	5	2	2	2	1	2	2	2	2	60
48	16911A0552	5	2	2	2	0	2	2	2	2	2	5	2	2	2	1	2	2	2	2	61
49	16911A0553	5	2	2	2	2	2	2	4	4	4	5	2	2	2	1	4	2	4	5	55
50	16911A0554	5	2	2	2	2	2	3	3	3	3	5	2	2	2	2	3	3	3	3	52
51	16911A0555	5	2	2	2	2	2	3	3	3	2	5	2	2	2	2	2	3	3	3	55
52	16911A0556	5	2	2	2	2	2	4	4	4	4	5	2	2	2	2	4	4	4	4	60
53	16911A0558	5	2	2	2	2	2	5	5	4	4	5	1	2	2	2	4	5	5	5	61
54	16911A0559	5	2	2	2	2	2	3	3	3	3	5	1	2	2	2	1	3	3	3	62
55	16911A0560	5	2	2	2	2	2	1	3	3	3	5	1	2	2	2	3	1	3	3	59
56	16911A0561	5	2	2	2	2	2	3	3	3	4	5	2	2	2	2	4	3	3	3	57
57	16911A0562	5	2	2	2	2	2	4	4	2	4	5	2	2	2	2	4	4	4	5	54
58	16911A0563	5	2	2	2	2	2	5	3	3	4	5	2	2	2	2	4	5	5	5	57
59	16911A0564	5	2	2	2	2	2	3	3	3	3	5	2	2	2	2	1	3	3	3	59
60	16911A0565	5	2	2	2	2	2	1	3	3	3	5	2	2	2	2	3	1	3	3	58
61	16911A0566	5	2	2	2	2	2	3	3	3	4	5	2	2	2	2	4	3	3	3	60
62	16911A0567	5	2	2	2	2	2	4	5	4	4	5	2	2	2	2	4	4	5	5	56
63	16911A0568	5	2	2	2	2	2	3	3	3	3	5	1	2	2	2	3	3	3	3	57
64	16911A0569	5	2	2	2	2	2	3	3	3	2	5	1	2	2	2	2	3	3	3	56
65	16911A0570	5	2	2	2	2	2	4	4	4	4	5	2	2	2	2	4	4	4	4	68
66	16911A0572	5	2	2	2	2	2	3	3	3	3	5	2	2	2	2	3	3	3	3	54
67	16911A0573	5	2	2	2	2	2	4	4	4	4	5	2	2	2	1	4	4	4	4	61
68	16911A0575	5	2	2	2	2	2	2	2	2	3	5	2	2	2	1	3	2	2	2	48
69	16911A0576	5	2	2	2	2	2	2	2	2	2	5	2	2	2	1	2	2	2	2	59
70	16911A0577	5	2	2	2	0	2	2	2	2	2	5	2	2	2	1	2	2	2	2	59
71	16911A0579	5	2	2	2	2	2	2	4	4	4	5	2	2	2	1	4	2	4	4	54
72	16911A0580	5	2	2	2	2	2	3	3	3	3	5	2	2	2	2	3	3	3	3	59
73	16911A0581	5	2	2	2	2	2	3	3	3	2	5	2	2	2	2	2	3	3	3	50
74	16911A0583	5	2	2	2	2	2	4	4	4	4	5	2	2	2	2	4	4	4	4	56
75	16911A0584	5	2	2	2	2	2	5	5	5	4	5	1	2	2	2	4	5	5	5	60
76	16911A0585	5	2	2	2	2	2	3	3	3	3	5	1	2	2	2	1	3	3	3	62
77	16911A0586	5	2	2	2	2	2	1	3	3	3	5	1	2	2	2	3	1	3	3	57
78	16911A0587	5	2	2	2	2	2	3	3	3	4	5	2	2	2	2	4	3	3	3	56
79	16911A0588	5	2	2	2	2	2	4	4	4	4	5	2	2	2	2	4	4	5	5	61
80	16911A0590	5	2	2	2	2	2	5	5	5	4	5	2	2	2	2	4	5	5	5	59
81	16911A0591	5	2	2	2	2	2	3	3	3	3	5	2	2	2	2	1	3	3	3	45
82	16911A0592	5	2	2	2	2	2	1	3	3	3	5	2	2	2	2	3	1	3	3	65
83	16911A0593	5	2	2	2	2	2	3	3	3	4	5	2	2	2	2	4	3	3	3	60
84	16911A0594	5	2	2	2	2	2	4	5	4	5	5	2	2	2	2	4	4	5	5	64
85	16911A0595	5	2	2	2	2	2	3	3	3	3	5	1	2	2	2	3	3	3	3	55
86	16911A0596	5	2	2	2	2	2	3	3	3	2	5	1	2	2	2	2	3	3	3	37
87	16911A0597	5	2	2	2	2	2	4	4	4	4	5	2	2	2	2	4	4	4	4	58
88	16911A0598	5	2	2	2	2	2	3	3	3	3	5	2	2	2	2	3	3	3	3	51
89	16911A0599	5	2	2	2	2	2	4	4	4	4	5	2	2	2	1	4	4	4	4	53
90	16911A05A0	5	2	2	2	2	2	2	2	2	3	5	2	2	2	1	3	2	2	2	49
91	16911A05A1	5	2	2	2	2	2	2	2	2	2	5	2	2	2	1	2	2	2	2	62
92	16911A05A2	5	2	2	2	0	2	2	2	2	2	5	2	2	2	1	2	2	2	2	63





145	16911A05F9	5	2	2	2	2	4	2	2	4	5	2	2	2	2	4	4	5	3	4	5	64
146	16911A05G0	5	2	2	2	2	5	3	4	5	5	2	2	2	2	4	5	5	3	5	5	--
147	16911A05G1	5	2	2	2	2	3	3	3	3	5	2	2	2	2	1	3	3	3	3	3	61
148	16911A05G2	5	2	2	2	2	1	3	3	3	5	2	2	2	2	3	1	3	3	3	3	44
149	16911A05G3	5	2	2	2	2	3	3	3	4	5	2	2	2	2	4	3	3	3	3	3	49
150	16911A05G4	5	2	2	2	2	4	5	4	5	5	2	2	2	2	4	4	5	5	5	5	56
151	16911A05G5	5	2	2	2	2	3	3	3	3	5	1	2	2	2	3	3	3	3	3	3	51
152	16911A05G6	5	2	2	2	2	3	3	3	2	5	1	2	2	2	2	3	3	3	3	3	62
153	16911A05G7	5	2	2	2	2	4	4	4	5	5	2	2	2	2	4	4	4	4	4	4	60
154	16911A05G8	5	2	2	2	2	3	3	3	3	5	2	2	2	2	3	3	3	3	3	3	51
155	16911A05G9	5	2	2	2	2	4	4	4	4	5	2	2	2	2	1	4	4	4	4	4	54
156	16911A05H0	5	2	2	2	2	2	2	3	3	5	2	2	2	2	1	3	2	2	2	2	60
157	16911A05H1	5	2	2	2	2	2	2	2	2	5	2	2	2	2	1	2	2	2	2	2	56
158	16911A05H2	5	2	2	2	0	2	2	2	2	5	2	2	2	2	1	2	2	2	2	2	43
159	16911A05H3	5	2	2	2	2	2	4	4	4	5	2	2	2	2	1	4	2	4	2	4	53
160	16911A05H4	5	2	2	2	2	3	3	3	3	5	2	2	2	2	2	3	3	3	3	3	58
161	16911A05H6	5	2	2	2	2	3	3	3	2	5	2	2	2	2	2	2	3	3	3	3	57
162	16911A05H7	5	2	2	2	2	4	4	4	4	5	2	2	2	2	4	4	4	4	4	4	59
163	16911A05H8	5	2	2	2	2	5	5	5	4	5	1	1	1	1	2	4	5	5	5	5	63
164	16911A05J0	5	2	2	2	2	3	3	3	3	5	1	1	1	2	1	3	3	3	3	3	41
165	16911A05J1	5	2	2	2	2	1	3	3	3	5	1	1	1	1	2	3	1	3	3	3	57
166	16911A05J2	5	2	2	2	2	3	3	3	4	5	2	1	1	2	4	3	3	3	3	3	52
167	16911A05J3	5	2	2	2	2	4	2	4	2	5	2	1	1	2	4	4	5	5	5	5	48
168	16911A05J4	5	2	2	2	2	5	3	4	4	5	2	1	1	2	4	5	5	5	5	5	51
169	16911A05J5	5	2	2	2	2	3	3	3	3	5	2	1	1	2	1	3	3	3	3	3	62
170	16911A05J6	5	2	2	2	2	1	3	3	3	5	2	1	1	2	3	1	3	3	3	3	56
171	16911A05J8	5	2	2	2	2	3	3	3	4	5	2	1	1	2	4	3	3	3	3	3	52
172	16911A05K0	5	2	2	2	2	4	5	4	4	5	2	2	2	2	4	4	5	5	5	5	56
173	16911A05K2	5	2	2	2	2	3	3	3	3	5	1	2	2	2	3	3	3	3	3	3	56
174	16911A05K3	5	2	2	2	2	3	3	3	2	5	1	2	2	2	2	3	3	3	3	3	33
175	16911A05K4	5	2	2	2	2	4	4	4	4	5	2	2	2	2	4	4	4	4	4	4	44
176	16911A05K5	5	2	2	2	2	3	3	3	3	5	2	2	2	2	3	3	3	3	3	3	51



177	16911A05K6	5	2	2	2	2	4	4	4	4	5	2	2	1	4	4	49
178	16911A05K7	5	2	2	2	2	2	2	3	5	5	2	2	1	3	2	26
179	16911A05K8	5	2	2	2	2	2	2	2	5	5	2	2	1	2	2	61
180	16911A05L0	5	2	2	0	2	2	2	2	2	5	2	2	1	2	2	53
181	16911A05L1	5	2	2	2	2	2	4	4	5	5	2	2	1	4	2	41
182	16911A05L2	5	2	2	2	2	3	3	3	5	5	2	2	2	3	3	57
183	16911A05L3	5	2	2	2	2	3	3	2	5	2	2	2	2	3	3	48
184	16911A05L4	5	2	2	2	2	4	4	4	5	5	2	2	2	4	4	55
185	16911A05L5	5	2	2	2	5	5	5	4	5	1	1	2	4	5	5	60
186	16911A05L6	5	2	2	2	3	3	3	3	5	5	1	1	2	1	3	58
187	16911A05L7	5	2	2	2	1	3	3	3	5	5	1	1	2	3	1	49
188	16911A05L8	5	2	2	2	3	3	3	4	5	5	2	1	2	4	3	42
189	16911A05L9	5	2	2	2	4	4	4	5	5	2	1	2	4	4	5	58
190	16911A05M0	5	2	2	2	5	3	4	5	5	2	1	2	4	5	5	50
191	16911A05M1	5	2	2	2	3	3	3	3	5	5	2	1	2	1	3	61
192	16911A05M2	5	2	2	2	1	3	3	3	5	5	2	1	2	3	1	60
193	16911A05M3	5	2	2	2	3	3	3	4	5	5	2	1	2	4	3	62
194	16911A05M4	5	2	2	2	4	5	4	5	5	2	2	2	4	4	5	62
195	16911A05M6	5	2	2	2	3	3	3	3	5	5	1	2	2	3	3	67
196	16911A05M7	5	2	2	2	3	3	3	2	5	1	2	2	2	3	3	68
197	16911A05M8	5	2	2	2	4	4	4	4	5	2	2	2	2	4	4	66
198	16911A05M9	5	2	2	2	3	3	3	3	5	2	2	2	2	3	3	65
199	16911A05N0	5	2	2	2	4	4	4	4	5	5	2	2	1	4	4	71
200	16911A05N1	5	2	2	2	2	2	2	3	5	2	2	2	1	3	2	71
201	16911A05N2	5	2	2	2	2	2	2	2	5	5	2	2	1	2	2	70
202	16911A05N3	5	2	2	0	2	2	2	2	5	2	2	2	1	2	2	56
203	16911A05N4	5	2	2	2	2	4	4	4	5	5	2	2	1	4	2	57
204	16911A05N5	5	2	2	2	3	3	3	3	5	5	2	2	2	3	3	62
205	16911A05N6	5	2	2	2	3	3	3	2	5	2	2	2	2	2	3	74
206	16911A05N7	5	2	2	2	4	4	4	4	5	2	2	2	2	4	4	68
207	16911A05N8	5	2	2	2	5	5	5	4	5	1	1	2	4	5	5	70
208	16911A05N9	5	2	2	2	3	3	3	3	5	5	1	1	2	1	3	71
209	16911A05P0	5	2	2	2	1	3	3	3	5	1	1	1	2	3	1	54
210	16911A05P1	5	2	2	2	3	3	3	4	5	5	2	1	2	4	3	63
211	16911A05P2	5	2	2	2	4	2	4	5	5	2	1	2	4	4	5	64
212	16911A05P3	5	2	2	2	5	3	4	4	5	2	1	2	4	5	5	72
213	16911A05P4	5	2	2	2	3	3	3	3	5	2	1	2	1	3	3	59
214	16911A05P5	5	2	2	2	1	3	3	3	5	5	2	1	2	3	1	72
215	16911A05P6	5	2	2	2	3	3	3	4	5	2	1	2	4	3	3	65
216	16911A05P7	5	2	2	2	4	5	4	4	5	2	2	2	4	4	5	66
217	16911A05P8	5	2	2	2	3	3	3	3	5	1	2	2	3	3	3	73
218	16911A05P9	5	2	2	2	3	3	3	3	5	1	2	2	2	3	3	58
219	16911A05Q0	5	2	2	2	4	4	4	4	5	2	2	2	4	4	4	55
220	17915A05O1	5	2	2	2	3	3	3	3	5	2	2	2	2	3	3	63
221	17915A05O2	5	2	2	2	5	3	4	4	5	2	1	2	4	5	5	63
222	17915A05O3	5	2	2	2	3	3	3	3	5	2	2	1	2	3	3	52
223	17918A05O1	5	2	2	2	1	3	3	3	5	5	2	1	2	1	3	64
Average marks		5	2.0	2.0	1.9	2.3	3.0	3.2	3.3	2.3	2.3	1.8	1.9	2.3	3.1	3.0	58.6
No of students attempted		223	223	223	223	223	223	223	223	223	223	223	223	223	223	223	220
No of students scored		223	223	223	213	162	162	183	183	223	193	173	193	162	162	193	209
No of students scored		100	100	100	95.52	72.65	82.06	82.06	82.06	100	77.58	86.55	77.58	72.65	86.55	95.00	
CO ATTAINMENT		3	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3	3.0	3.0	3.0	3.0	3.0	3.0	3.0

ASSESSMENT OF COs FOR THE COURSE						
CO	Method	value	Avg	CO Attainment	CO Attainment (End	Overall CO Attainment
CO 1	ASM I	3				
	MID I - PART A - Q1	3.0	3.0			
	MID I - PART B - Q4	3.0				
	ASM I	3				
CO 2	MID I - PART A - Q2	3.0	3.0			
	MID I - PART B - Q5	3.0				
	ASM I	3				
	ASM II	3				
CO 3	MID I - PART A - Q3	3.0				
	MID I - PART B - Q6	3.0	3.0	3.00	3.00	3.00
	MID II - PART A - Q1	3.0				
	MID II - PART B - Q4	3.0				
	ASM II	3				
	MID II - PART A - Q2	3.0	3.0			
CO 4	MID II - PART B - Q5	3.0				
	ASM II	3				
	MID II - PART A - Q3	3.0	3.0			
CO 5	MID II - PART B - Q6	3.0				

*R. S. Narayanan*  
COURSE-COORDINATOR

*[Signature]*  
HOD/CSE



# VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

R15

16-20

DP					
	OP1	OP2	OP3	TOTAL	ATTAIN
CO1	3	10	210	223	2.93
CO2	5	20	198	223	2.87
CO3	3	20	200	223	2.88
CO4	7	16	199	223	2.85
CO5	5	8	210	223	2.92
					<b>2.89</b>

CO OVERALL ATTAINMENT=80% OF Direct + 20% of Indirect

$$0.8 \times 3 + 0.2 \times 2.89 = 2.98$$

# **Course End Survey Form**



# VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

(Accredited by NBA, Approved by AICTE New Delhi & Permanently Affiliated to JNTUH)  
Aziz Nagar Gate, C.B. Post, Hyderabad-500 075.

## **Department of Computer Science & Engineering** **Course End Survey Form Academic year: 2019-20**

Name of the student		Year & sem	IV – II
Roll number		Regulations	R 15

Dear Student,

We need your help in evaluating the courses offered, by responding the short survey below.

Your feedback is very valuable for us in order to continually improve our program. The aim of this survey is to evaluate how well each of the courses has prepared you to have necessary skills.

Your responses will be kept confidential and will not be revealed to anyone outside the department without your permission.

Please indicate (✓) the level to which you agree with the following criterion:  
(3: Strongly agree      2: Agree      1: Strongly disagree)

Name of The Course: DESIGN PATTERNS		RATING		
After completing this course the student must demonstrate the knowledge and ability to		3	2	1
CO 1	Understand the Design patterns in software applications			
CO 2	Discuss the Creational Patterns			
CO 3	Categorize the Structural Pattern			
CO 4	Investigate Behavioral Patterns			
CO 5	Construct the good design pattern structures			

Any other comments / suggestions: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

Signature