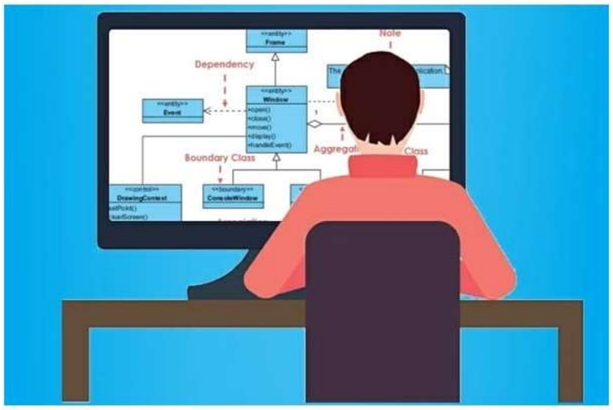
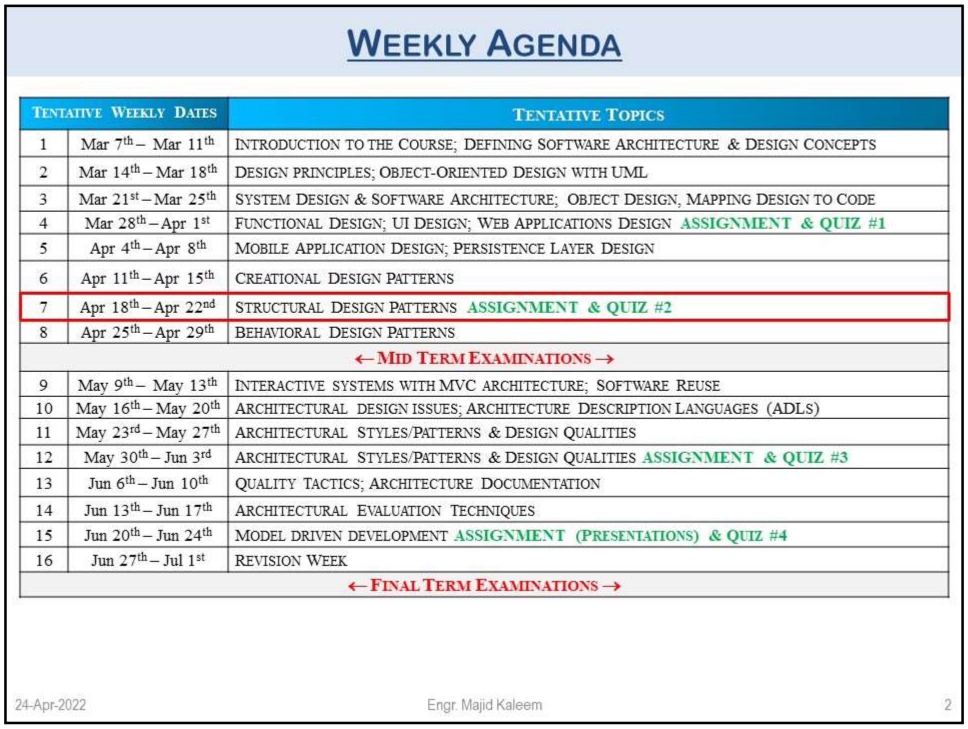
Software Design & Architecture Spring 2022 - Week-07



مدرس: مہندس ماجد کلیم

جامعہ بحریہ، واقعگاہ کراچی

Engr. Majid Kaleem



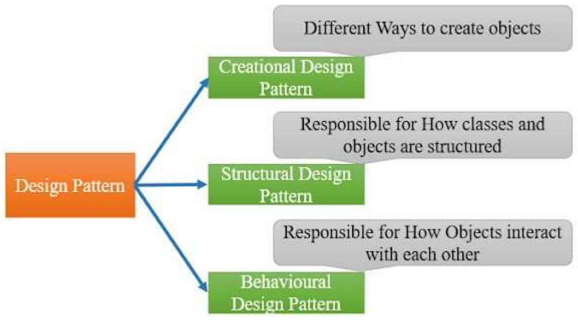
Weekly Agenda

Tentative Weekly Dates		Tentative Topics
1	Mar 7th – Mar 11th	Introduction to the Course; Defining Software Architecture & Design Concepts
2	Mar 14th – Mar 18th	Design principles; Object-Oriented Design with UML
3	Mar 21st – Mar 25th	System Design & Software Architecture; Object Design, Mapping Design to Code
4	Mar 28th – Apr 1st	Functional Design; UI Design; Web Applications Design **Assignment & Quiz #1**
5	Apr 4th – Apr 8th	Mobile Application Design; Persistence Layer Design
6	Apr 11th – Apr 15th	Creational Design Patterns
7	Apr 18th – Apr 22nd	Structural Design Patterns **Assignment & Quiz #2**
8	Apr 25th – Apr 29th	Behavioral Design Patterns
		← Mid Term Examinations →
9	May 9th – May 13th	Interactive systems with MVC architecture; Software Reuse
10	May 16th – May 20th	Architectural design issues; Architecture Description Languages (ADLs)
11	May 23rd – May 27th	Architectural Styles/Patterns & Design Qualities
12	May 30th – Jun 3rd	Architectural Styles/Patterns & Design Qualities **Assignment & Quiz #3**
13	Jun 6th – Jun 10th	Quality Tactics; Architecture Documentation
14	Jun 13th – Jun 17th	Architectural Evaluation Techniques
15	Jun 20th – Jun 24th	Model driven development **Assignment (Presentations) & Quiz #4**
16	Jun 27th – Jul 1st	Revision Week
		← Final Term Examinations →

24-Apr-2022 Engr. Majid Kaleem 2

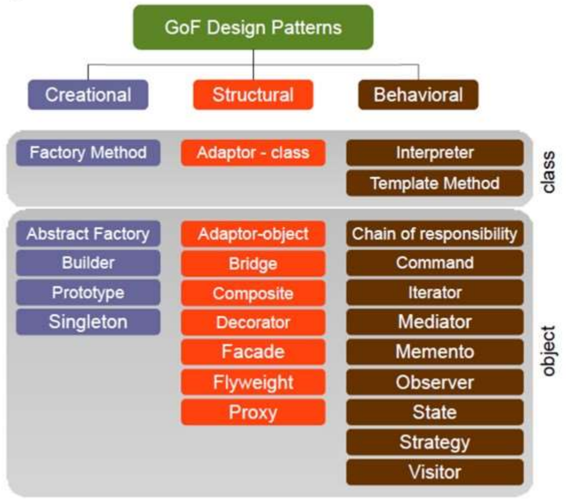
Classification of Design Patterns

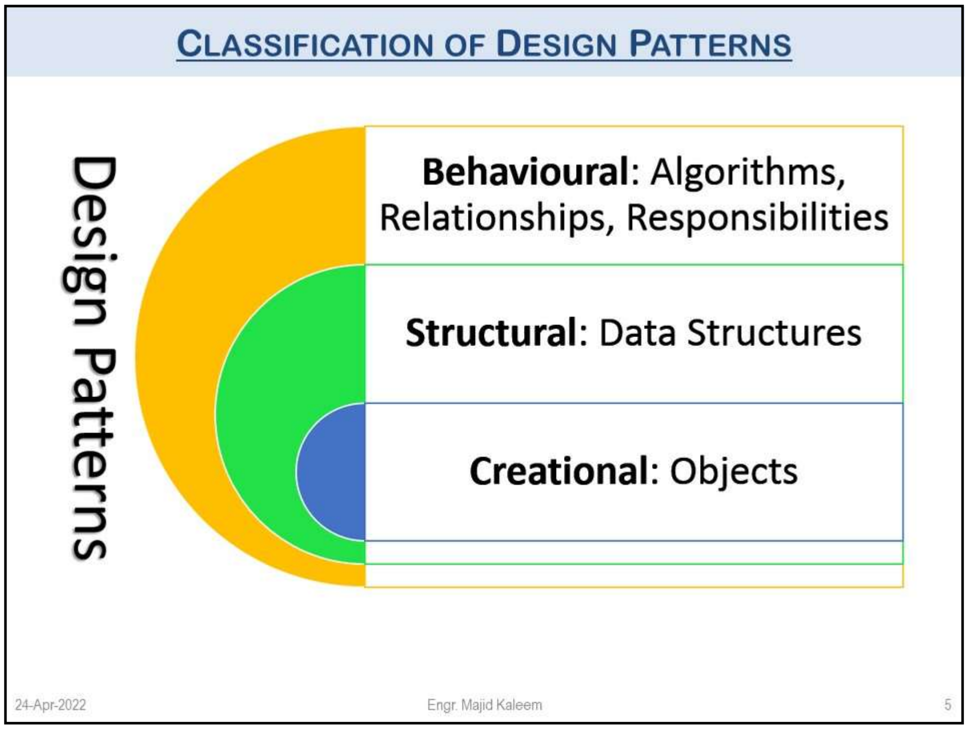
- They are categorized in three groups: Creational, Structural, and Behavioral as listed below:


Design Pattern
Creational Design Pattern
Different Ways to create objects
Structural Design Pattern
Responsible for How classes and objects are structured
Behavioural Design Pattern
Responsible for How Objects interact with each other

Classification of Design Patterns

- 23 GoF Design Patterns:





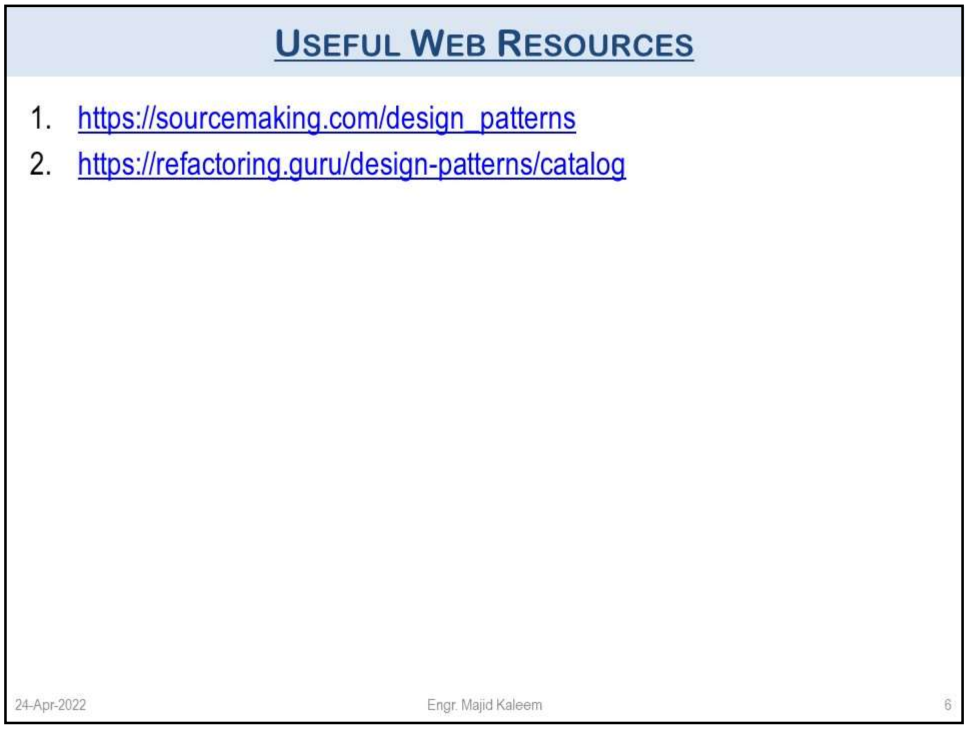
Classification of Design Patterns

Design Patterns

Behavioural: Algorithms, Relationships, Responsibilities

Structural: Data Structures

Creational: Objects



Useful Web Resources

1. https://sourcemaking.com/design_patterns
2. https://refactoring.guru/design-patterns/catalog

STRUCTURAL PATTERNS

- These design patterns are all about Class and Object *composition*.
- Structural class-creation patterns use *inheritance* to compose interfaces.
-
- Structural object-patterns define ways to compose objects to obtain *new functionality*.

ADAPTOR PATTERN

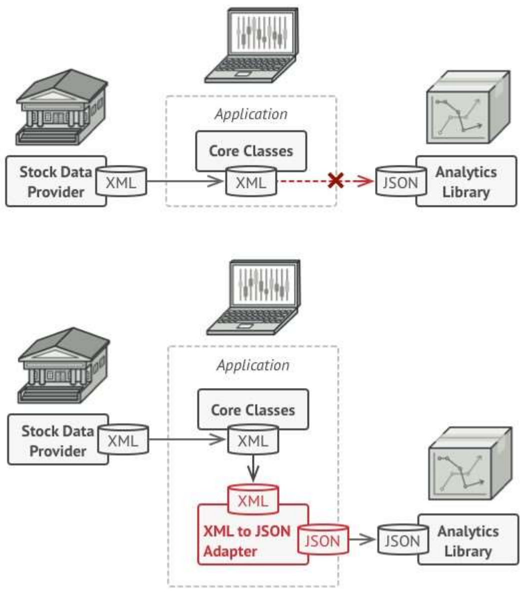
- The Adapter Pattern is a Structural Design Pattern, its work as a bridge between two *incompatible* interfaces.
- It's involves a *single* class which is responsible to join functionalities of independent or *incompatible* interfaces.
- Converts the .interface of a class into another interface that a client wants.

ADAPTOR PATTERN

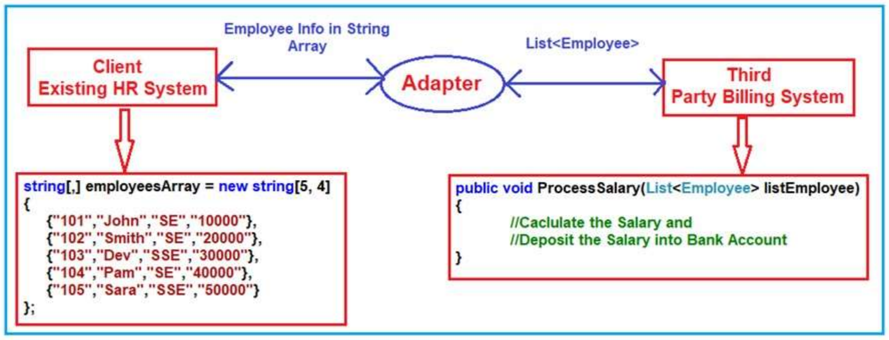
Real-Life Example

A common use of this pattern is when you use an electrical outlet adapter/AC power adapter in international travels. These adapters can act as middlemen so that an electronic device, say a laptop that accepts a U.S. power supply, can be plugged into a European power outlet.

ADAPTOR - ANOTHER EXAMPLE



ADAPTOR - ANOTHER EXAMPLE



```
string[,] employeesArray = new string[5, 4]
{
    {"101","John","SE","10000"},
    {"102","Smith","SE","20000"},
    {"103","Dev","SSE","30000"},
    {"104","Pam","SE","40000"},
    {"105","Sara","SSE","50000"}
};
```

```
public void ProcessSalary(List<Employee> listEmployee)
{
        //Caclulate the Salary and
        //Deposit the Salary into Bank Account
}
```

11

BRIDGE PATTERN

- The Bridge Pattern is a Structural Design Pattern, it split a class or set of closely related classes into two separate hierarchies, *abstraction* and *implementation*.
- It's allow the abstraction and implementation to be developed independently.
- The client code can access only abstraction part without being concerned about the implementation part.

OR

- The Bridge Design Pattern helps us to separate an interface (*or an abstract class*) from its implementer class so that both entities can vary (*developed*) independently.
- The pattern provides a way to change the implementation details without the need to changes the abstraction.

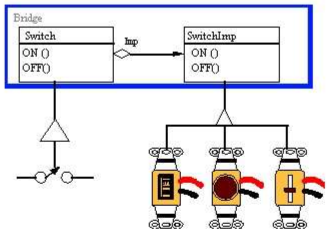
12

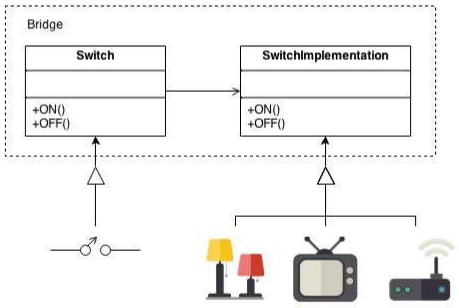
Bridge Pattern

Real-Life Example

In a software product development company, the development team and the marketing team both play crucial roles. The marketing team does a market survey and gathers the customer requirements. The development team implements those requirements in the product to fulfill the customer needs. Any change (say, in the operational strategy) in one team should not have a direct impact on the other team. In this case, you can think of the marketing team as playing the role of the bridge between the clients of the product and the development team of the software organization.

Bridge – Another Example



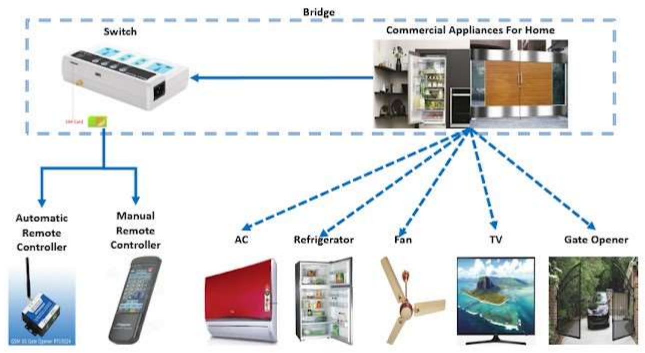


BRIDGE – ANOTHER EXAMPLE

- Consider you are creating **HouseHold Management Application**. In a house there are so many commercial appliances like AC, Refrigerator, Fan, TV, Gate Opener and so on…. that you can turn on or off. There are different ways to turn the appliance on or off, such as using switch or using automatic remote control mechanism or manual remote control mechanism. It's good in application but how about code? In this case automatic remote controller using same switch but different commercial home appliance, on the same time manual remote controller using same switch but different commercial home appliance. To overcome this problem bridge pattern come to this place. It work as bridge interface between abstraction (**(i.e.) Automatic Remote Controller, Manual Remote Controller**) and implementation (**(i.e.) TV, Refrigerator, Gate Opener and so on…..**)

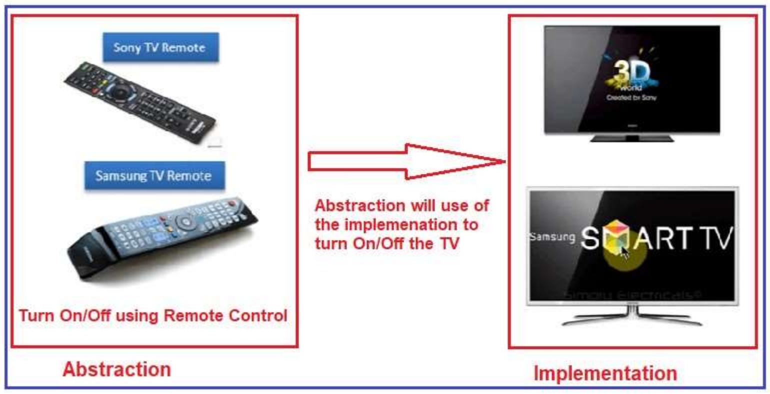
15

BRIDGE – ANOTHER EXAMPLE



16

BRIDGE – ANOTHER EXAMPLE



17

COMPOSITE PATTERN

- The Composite Pattern is a **Structural Design Pattern**, it also known as **partitioning design pattern**.
- It describes a group of objects that is **treated the same way as a single instance of the same type of object**.
- It composes objects in terms of **a tree structure** to represent part as well as whole hierarchy. Client treat individual objects and compositions of objects uniformly.

Real-Life Example

Think of an organization that consists of many departments. In general, each of these departments consists of multiple employees (in other words, all these participants are basically employees in the organization). Some employees are grouped together to form a department, and those departments can be further grouped together to build the whole organization.

"A conceptual whole made up of complicated and related parts"

18

COMPOSITE PATTERN

- The Composite Pattern shows how to combine several classes into an object that has the same behavior as its parts.

"A conceptual whole made up of complicated and related parts"

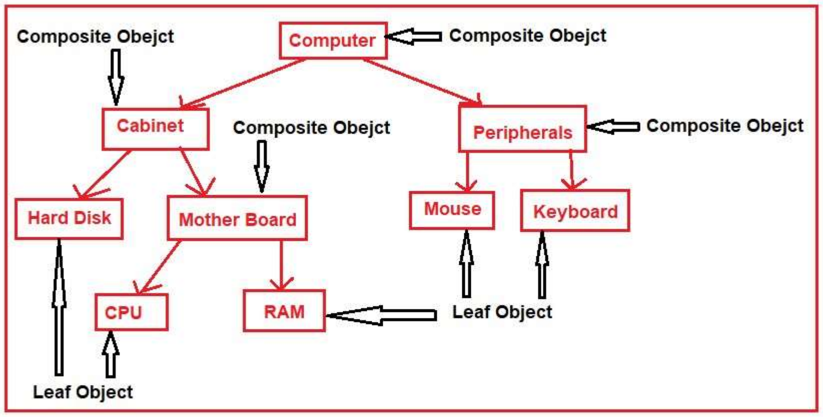
19

COMPOSITE – ANOTHER EXAMPLE

- Consider you are creating Employee Management Application. Normally organization have hierarchy of CEO ► Director ► Managers ► Team Leader► Developer.
- The owner of organization want to know all employee detail information.
- In normal application we have maintain lot of classes and subclass to get those of information.
- To overcome this problem composite pattern come to this place.
- It work as common interface component ((i.e.) IEmployee) and leaf tree node ((i.e.) Employee) and single full control composite class which is act as intermediator of component and leaf tree node.

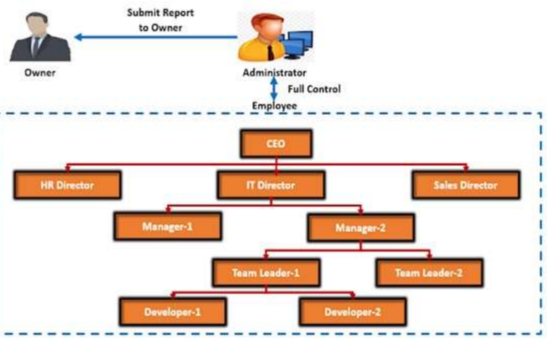
20

Composite – Another Example



21

Composite – Another Example



22

Decorator Pattern

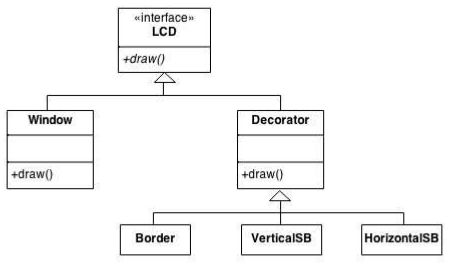
- Decorator pattern attach *additional functionalities* or responsibility to an *existing object at the runtime*.
- It adds new behavior to an individual object without affecting other objects of the same class.
- New behavior or functionalities can be added dynamically at the run time

Real-Life Example

Suppose you own a single-story house and you decide to build a second floor on top of it. Obviously, you may not want to change the architecture of the ground floor. But you may want to change the design of the architecture for the newly added floor without affecting the existing architecture.

Decorator – Another Example

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



Decorator – Another Example

- Consider you are developing student management application. First you're creatingbasic responsibility to handle and print information like name, age, gender & grade etc.
- After a while you have got a new requirement to add functionality to maintain lab reports for science students. Basically how will do this? May be extend the Student class to create ScienceStudent subclass for this responsibility. It's good enough for this requirement.
- You have got another requirement to add functionality to maintaining game reports for sports students. Again you may want to extend the Student class to create a SportsStudent subclass for this responsibility.

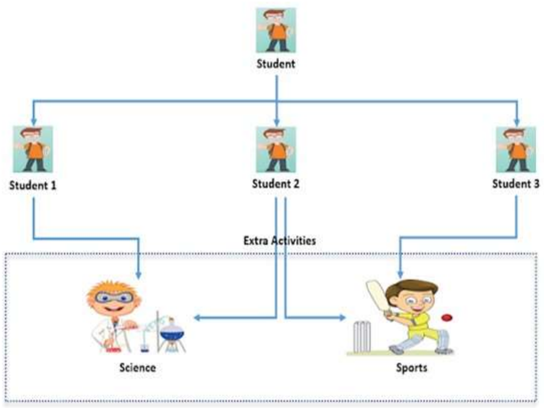
25

Decorator – Another Example

- You have got another requirement to maintain information of the student who is studying science and also playing sports.
- You will have to maintain lab reports, game reports along with name, age, gender and grade.
- You have already created two subclass's for science students and sports students, but there is no way you can reuse them in this scenario.
- You are left with no option but to create a subclass for Science Sports Student and this is a problem.
- How many subclasses you keep creating?
- Decorator pattern solves this problem.

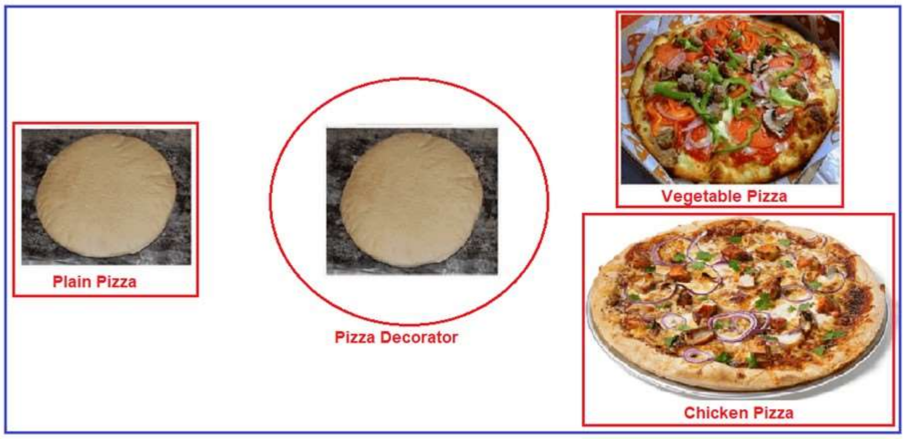
26

Decorator – Another Example



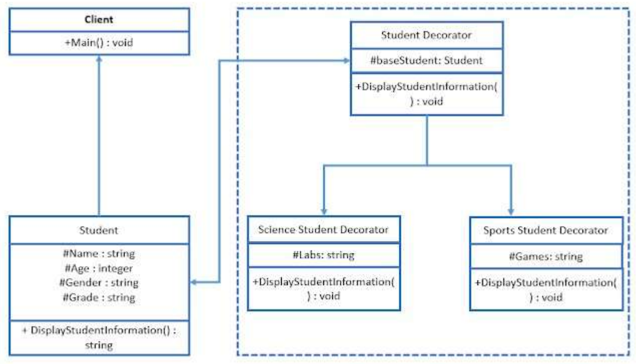
27

Decorator – Another Example



28

DECORATOR – ANOTHER EXAMPLE UML



29

FAÇADE PATTERN

- Facade pattern *hides the complexities* of the system and provide *flexible interface to the client*.
- Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.
- Facade as the name suggests means the *front of the building*. The people walking past the road can only see this glass front of the building. They do not know anything about it, the wiring, pipes and other complexities. The front hides all the complexities of the building and display a friendly look.

30

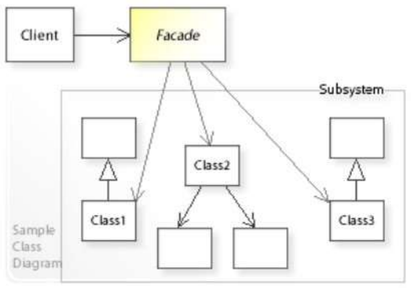
FAÇADE PATTERN

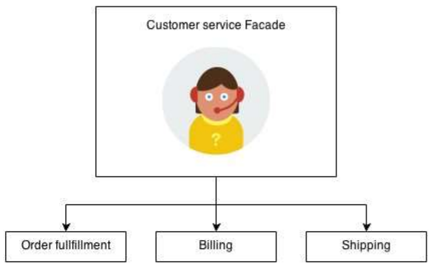
Real-Life Example

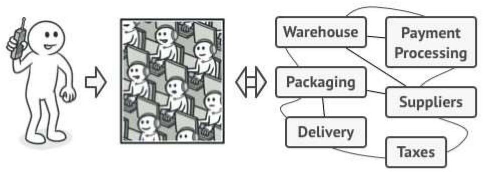
Suppose you are going to host a birthday party with 300 guests. Nowadays you can hire a party organizer and let them know the key information such as the party type, date and time of the party, number of attendees, and so on. The organizer will do the rest for you. You do not need to think about how they will decorate the party room, whether the food will be buffet style, and so on.

31

FAÇADE – ANOTHER EXAMPLE







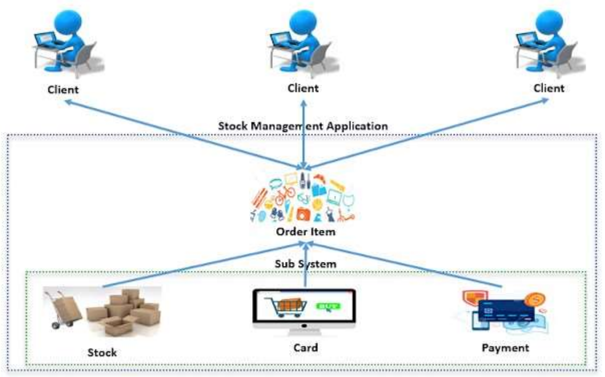
32

FAÇADE – ANOTHER EXAMPLE

- Imaging your using remote control to operate TV channel, volume & power button.
- Actually end user do not know the internal functionality of each action.
- Just they are clicking buttons for action, based on action the respective functionality work internal.
- It hide internal complexity to end user.
- Just show as user friendly display button.

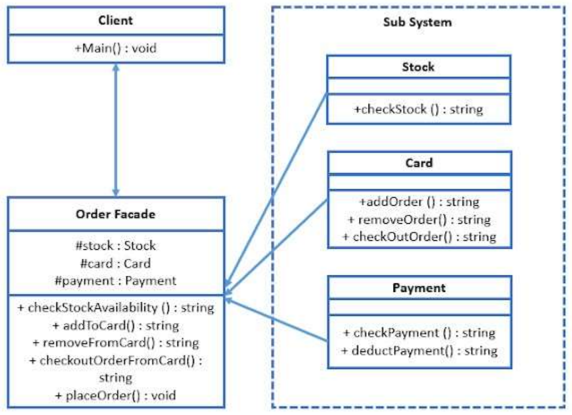
33

FAÇADE – ANOTHER EXAMPLE



34

FAÇADE – ANOTHER EXAMPLE UML



35

FLYWEIGHT PATTERN

- 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.

Real-Life Example
Suppose you have a pen. You can use different ink refills to write with different colors. So, the pen without the refill can be considered the flyweight with intrinsic data, and the refills can be considered the extrinsic data in this example.

36

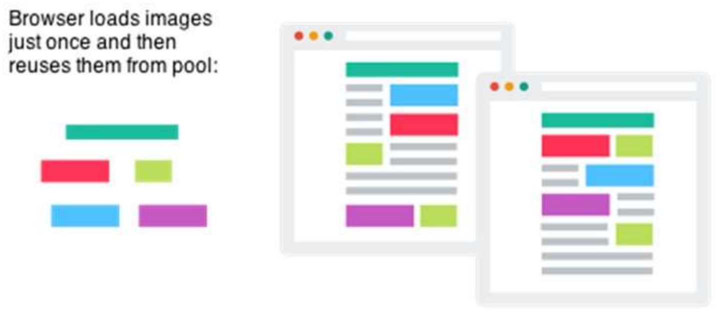
Flyweight Pattern

- Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by *sharing* common parts of state between multiple objects instead of keeping all of the data in each object.
- The Flyweight pattern promotes an efficient way to *share* common information present in small objects that occur in a system in large numbers.
- The Flyweight uses *sharing* to support large numbers of objects efficiently

37

Flyweight- Another Example

- Fly weight pattern is useful where we need to create many objects and all these objects share some kind of common data.
- We can minimize memory by just keeping one copy of the static data and referencing the same data in all objects of variable data.


Browser loads images just once and then reuses them from pool:

38

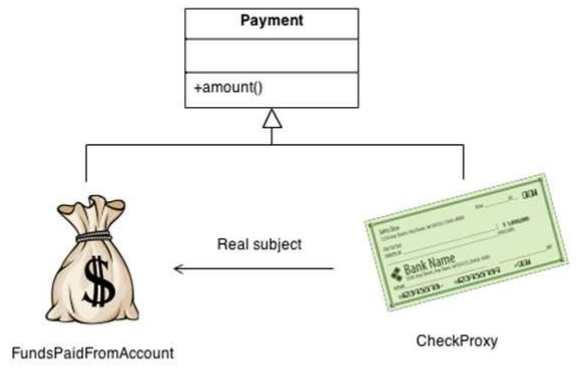
PROXY PATTERN

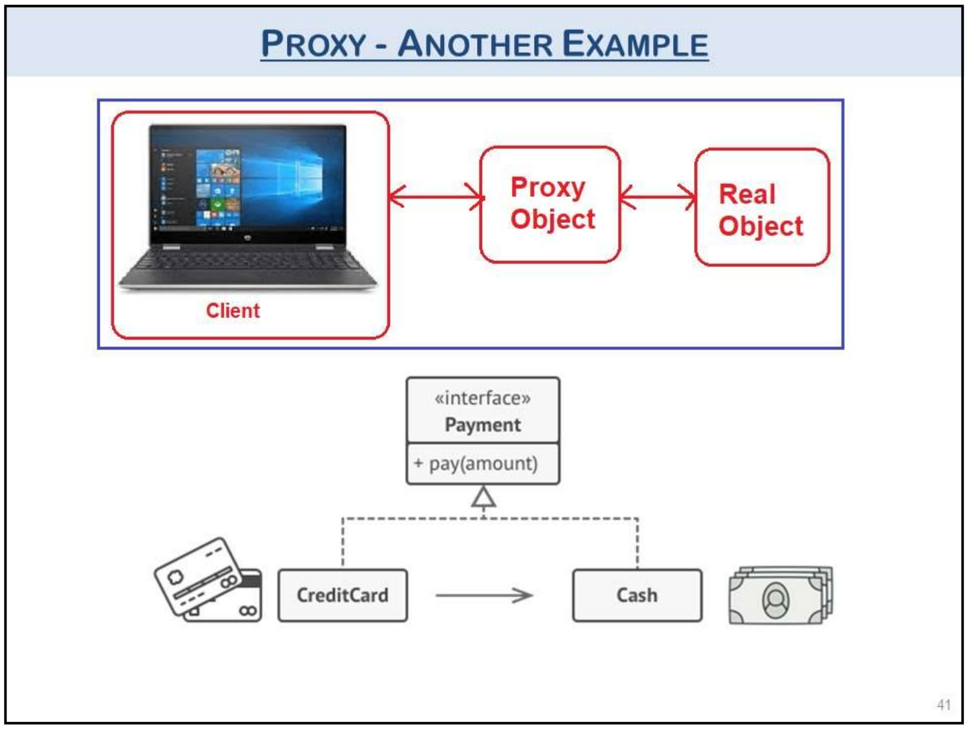
- In proxy pattern, a class represents functionality of another class.
- Proxy fundamentally is a class functioning as in interface which points towards the actual class which has data.
- *Proxy: A person authorized to act for another*

PROXY - ANOTHER EXAMPLE

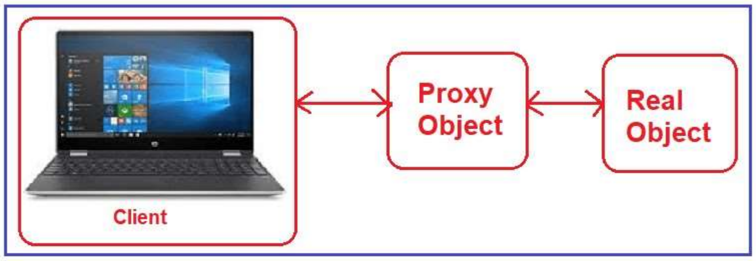
Real-Life Example

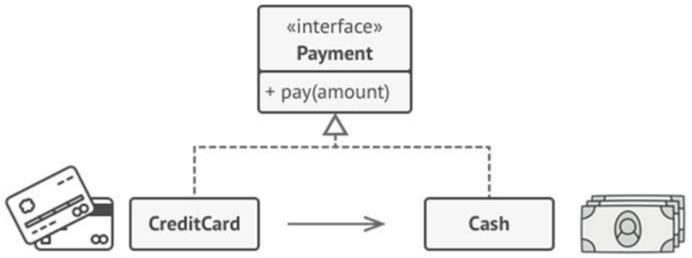
A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account. .



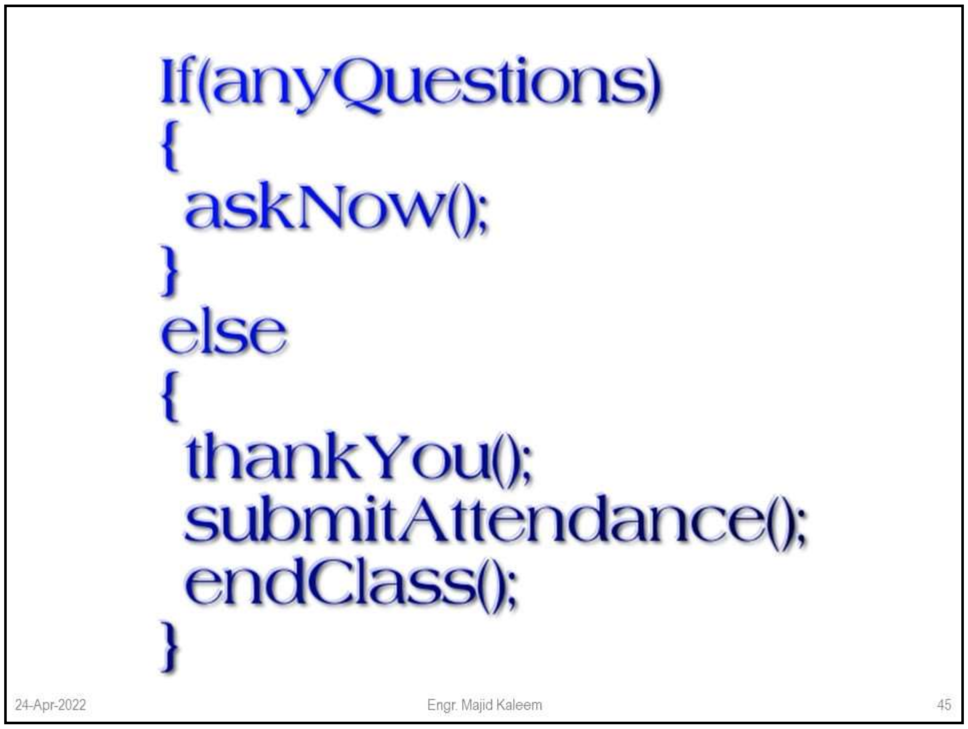


Proxy - Another Example



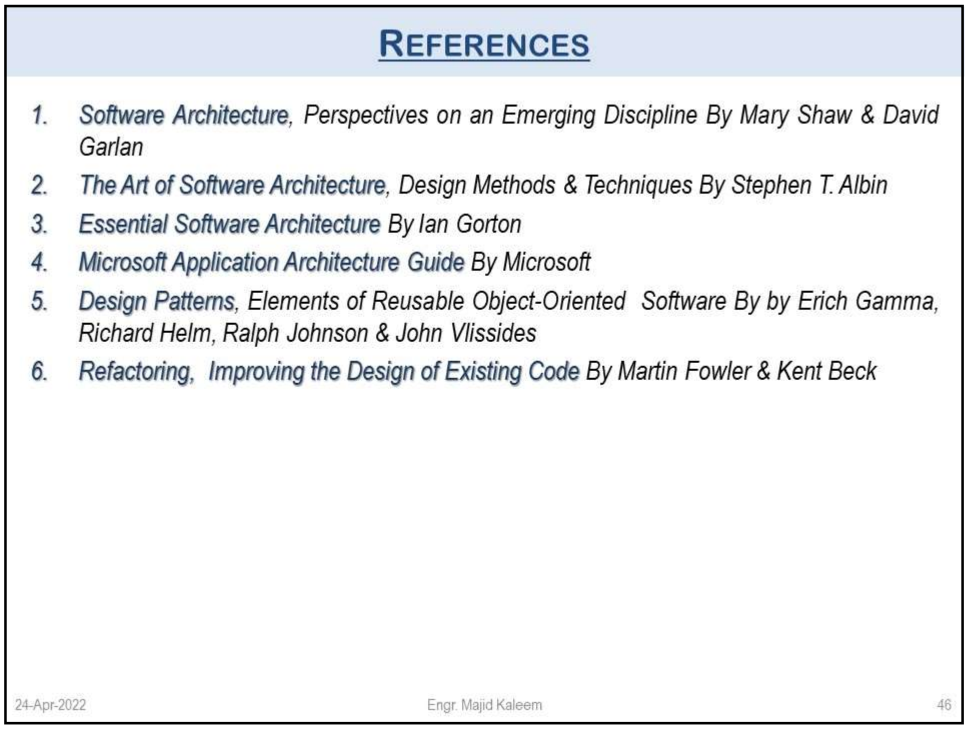


41



```
If(anyQuestions)
{
 askNow();
}
else
{
 thankYou();
 submitAttendance();
 endClass();
}
```

24-Apr-2022 Engr. Majid Kaleem 45



REFERENCES

1. *Software Architecture, Perspectives on an Emerging Discipline By Mary Shaw & David Garlan*
2. *The Art of Software Architecture, Design Methods & Techniques By Stephen T. Albin*
3. *Essential Software Architecture By Ian Gorton*
4. *Microsoft Application Architecture Guide By Microsoft*
5. *Design Patterns, Elements of Reusable Object-Oriented Software By by Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides*
6. *Refactoring, Improving the Design of Existing Code By Martin Fowler & Kent Beck*