OOSD

Chapter 2 and 4

Design Patterns:

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design.

A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situation.

Object oriented design patterns typically show relationships between classes or objects without specifying the final application classes or objects that are involved

Importance of design pattern

→ Design patterns can speed up the development process by providing tested, proven development paradigms.

→ Effective design requires considering issues that may not became visible later in the implementation. Reusing design patterns helps to prevent such issues that may cause major problems.

→ It also improves code readability for coders and architects who are familiar with the patterns.

→ It allows developer to communicate using well-known well understand names for software interactions.

Scanned by CamScanner

1. Communication, Learning and Enhanced Sight

→ Design pattern helps in communication. one can easily tell another developer on the team. 'I've used command pattern here' and the other developer understands not just the design, but can also easily figure out the reason behind it.

→ Design pattern really help in learning especially when you are on new project.

→ Design pattern helps to provide better insight about parts of the application or the 3rd party frame-works that the developers use.

2. Decomposing ~~System~~ objects into ~~objects~~ System.

→ The hard part of OO design is finding the appropriate objects and decomposing the system. But the design patterns helps to make the more flexible and reuseable design by discovering those objects which are identified during analysis and early design phase.

3. Determining object granularity:

→ It helps to find right level of abstraction and granularity

4. Specifying Object Interface:

→ Design pattern helps in identifying the right interface and the relationship between various interface.

Scanned by CamScanner

Specifying Object implementation

It helps to provide guidence to implement an object which helps in good OO code

Ensuring right reuse mechanism:

It helps to make the right decision like when to use inheritance, when to use composition and so on to design highly reusable and maintainable code.

Relating run-time and compile-time structures:

Sometimes looking at the code-structure doesn't give us insights about run-time structure. In such case, knowing of design patterns helps to know some hidden structure ie (run-time structure).

Designing for change:

Design patterns make sure aspect of the system structure independent of other aspects, making the system more robust to a particular kind of change

Scanned by CamScanner

#1 Structure and Documentation of pattern:

→ The design pattern consists of following section

1. Name:
   → The unique name that helps to identify the particular pattern.

2. Context:
   → The situation in which the pattern is used.

3. Problem:
   → The description of the goal behind the pattern and the reason for using it.

4. Forces:
   → A scenario consisting of a problem and a context in which this pattern can be used.

5. Solution
   → The description of implementation of the pattern.

6 Antipatterns:
   → Antipatterns are certain patterns in software development that is considered a bad programming practise.

7. Related patterns:
   → other patterns that have some relationship with the pattern.

Scanned by CamScanner

8. References:

Sources from which the pattern is taken.
(Acknowledgements of those who developed or inspired the pattern)

Software Design Principles:

→ Software design principles represent a set of guidelines that helps us to avoid bad decision

Software Design concepts:

It is the idea behind the software 'design.
It is how we plan on solving the design problem in front of us.

technology change makes method changes but concept remain unchanged (eg: Poly, Abstraction, Modularity, Architecture, Pattern)

Drawbacks of Design Pattern: (critisim)

1. They Do not lead to direct code reuse.
2. They are complex in nature.
3. They are deceptively simple.
4. They are validated by experience and discussion.
5. Terms may suffer from pattern overload.
6. Integrating patterns into a software development process is a human-intensive activity.

Scanned by CamScanner

Programming paradigm means the approach we used for programming like: "functional programming", "procedural programming" and "OO programming"

Date
Page

#1 Programming Paradigm Vs Design Pattern:

→ The programming paradigm informs how the code gets written.

for example: in object oriented programming paradigm, the code is divided up into classes and typically supports inheritance and some type of polymorphism. The programmer creates the classes and then instances of the classes to carry out the operation of the program. similarly in functional programming paradigm codes written with a functional language involves lots of nested functions. In the same way, in procedural programming, you just say do step 1, do step 2 etc. without creating any classes.

→ A design pattern is a useful abstraction that can be implemented in any language. It is a general reusable solution to a commonly occuring problem in software.

for example:

if you have a bunch of steps you want to implement you might use 'compositive' and 'command' pattern to your implementation more generic.

→ A programming paradigm is a way of thinking when programming where first class concepts are used to organize the software.

→ A design pattern is a common successful use of software components.

Scanned by CamScanner

Date
Page

Classification of Design Pattern

→ In Software engineering, creational design patterns are design patterns that deals with object creating mechan trying to create objects in a manner suitable to situati The basic form of object creation could result in design problems or in added complexity to the design Creational design pattern solve this problem by some controlling object creation.

- consists of two dominant ideas:

1) Encapsulating knowledge about which concrete class system uses.

2) Hiding how instance of concrete classes are created and combined

eg: Abstraction occurance factory Method, Builder, Lazy initialization, object pool, prototype.

Structural Pattern: It is such design pattern that ease the design by identifying a simple way to realize relation ship between entities. eg: Adapter pattern, Bridge, composite, aggregation, facade etc.

Behavioral Pattern: It identify common communication patterns between objects and realize these patterns.

eg: Iterator, Observer, Visitor.

Scanned by CamScanner

Abstraction – occurance pattern:

- often in a domain model we find a set of objects (occurances)

1) Context:

-> This pattern is most frequently found in class diagram that form part of system domain model which contain a set of related objects which will call occurances.

Example of sets of occurances include.

- All the episodes of a television series. They have the same producer and the same series title, but different story-l
- The member of such a set share common information - but also differ from each other in important ways

2 Problem:

What is the best way to represent such sets of occurances in a class diagram?

3, Forces:

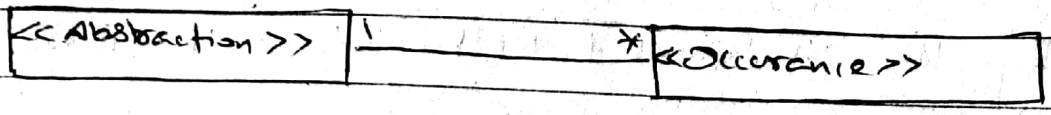
Suppose if we want to represent the members of each set of occurances without duplicating the common information. Duplication would consume unnecessary space and when the common information changes, it requires changing in all the occurances. furthermore, we want to avoid the risk of inconsistency that would result from changing common information.

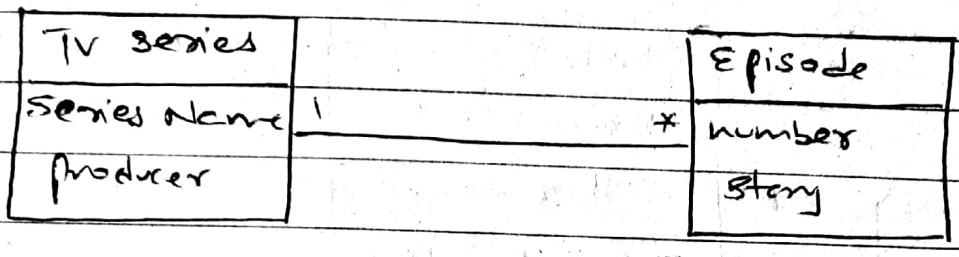
4 Solution:

Create an <<Abstraction>> class that containing the data that is common to all the members of set of occurances. Then create an <<occurance>> class representing the occurance of this abstraction. Connect this classes with

Scanned by CamScanner

a one-to-many association.

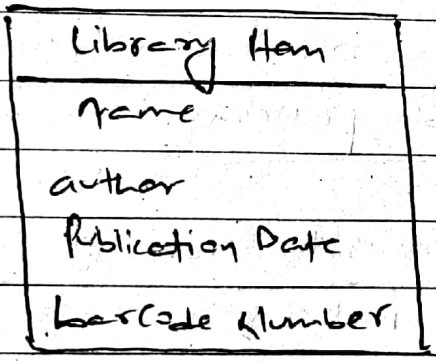




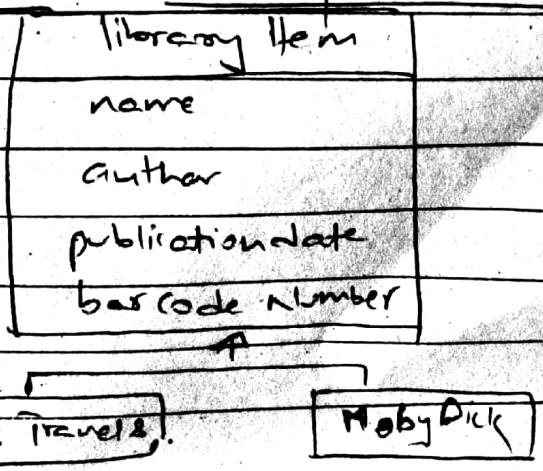
Example:

Antipattern.

Use of a single class which would result in duplication of information for each multiple of a class.

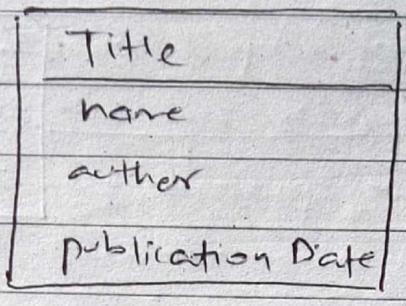


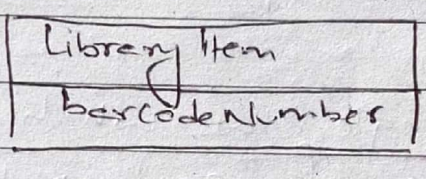
Create a separate "subclass" for each title. However, the information such as name, author etc would be duplicated in each of the instances of each subclass.



Scanned by CamScanner

- Making the abstraction class a superclass of a occurance class where the attributes in the superclass are inherited by the subclasses, but the data in those attributes are not.





6. Related pattern:

Abstraction occurance square pattern
Player role pattern

7. References:

It is the generalization of Title-Item pattern of Eriksson and Penker.

8. "what is the best way to represent related objects (occurence in class diagram).

Scanned by CamScanner

Date
Page

The General Hierarchy Pattern:

Context:

- Objects in a hierarchy can have one or more objects above them (superiors) and one or more objects below them (subordinates).
- Some objects cannot have any subordinates
- Example: the relationship between managers and subordinates in an organization.

Problem:

- How do you represent a hierarchy of objects in which some objects cannot have subordinates?

Forces:

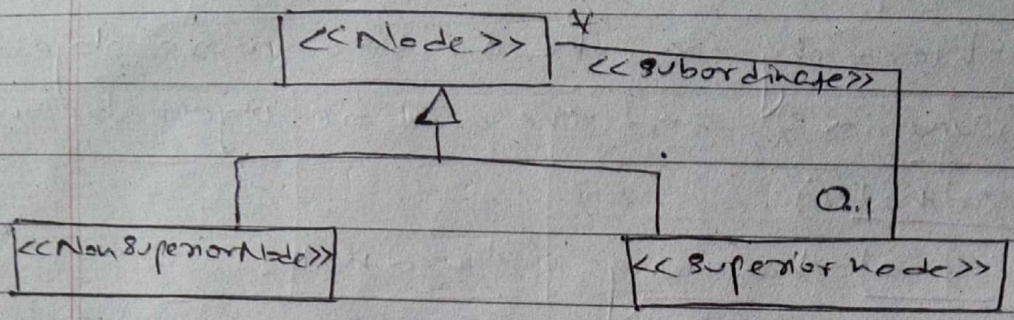
- If we want a flexible way of representing the hierarchy that prevents certain objects from having subordinates and we also have to consider the fact that all the objects share common features.

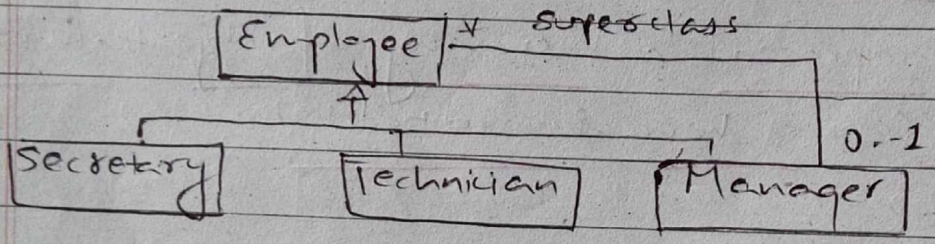
Solution:

- Create an abstract <<Node>> class. Then create at least two subclasses of the <<Node>> class. One of the subclasses <<SuperiorNode>> must be linked by a <<subordinates>> association to the superclass where as at least one subclass <<NonSuperiorNode>> must not be.
The subordinates of a <<SuperiorNode>> can thus be instances of either <<SuperiorNode>> or <<NonSuperior Node>>.

Scanned by CamScanner

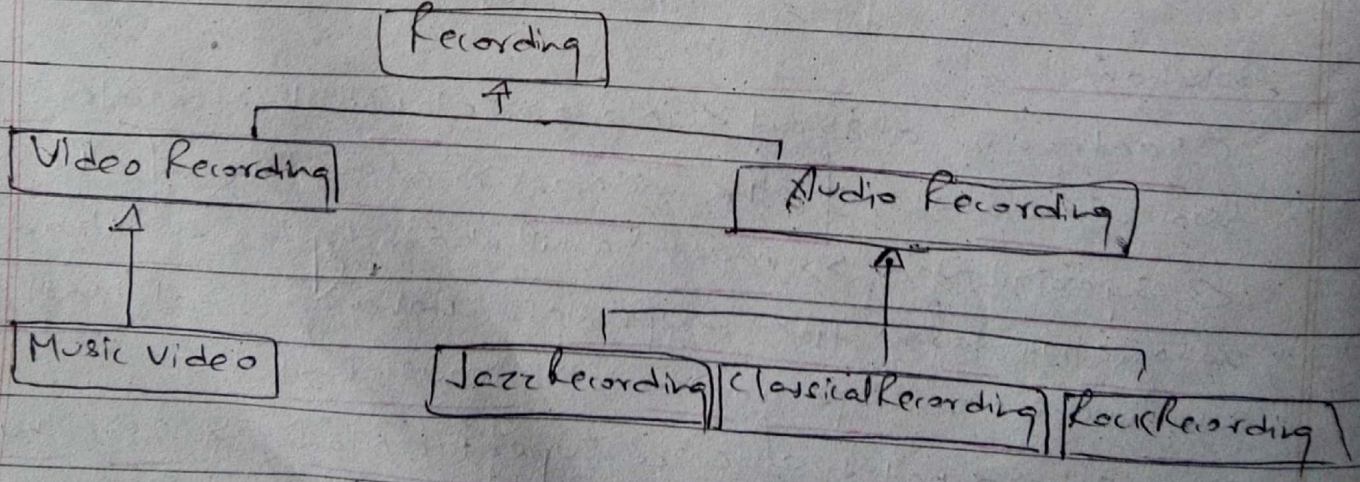
The multiplicity of the <<subordinates>> association can be optional 1-to-many or many-to-many.





Antipattern

→ Model a hierarchy of categories using a hierarchy of classes



Scanned by CamScanner

Related Pattern:

- The Reflexive Association pattern
- The composite pattern is a specialization of the General Hierarchy pattern.

References:

The composite pattern is one of the 'Gang of four Pattern'.

The Player-Role Pattern:

Context:

- A role is a particular set of properties associated with an object in a particular context.
- An object may play different roles in different context.

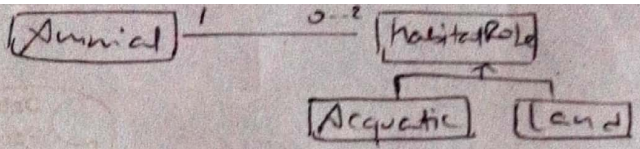
Problem:

- How do you best model players and roles so that a player can change roles or possess multiple roles?

Forces:

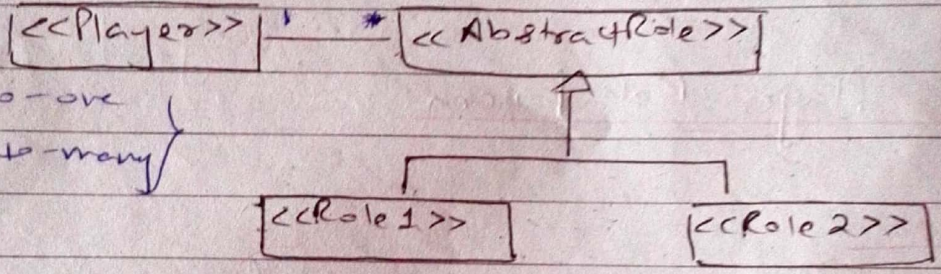
- we want to improve encapsulation by capturing the information associated with each separate role in a class. we want to avoid multiple inheritance
- Also we cannot allow an instance to change class.

Scanned by CamScanner



Solution:

Creat a <<Player>> class to represent the object that plays different roles. Create an association from this class to an abstract <<Role>> class, which is the superclass of a set of possible roles. The subclasses of this <<Role>> class encapsulate all the features associated with different roles.



If one role - One-to-one
more role - one-to-many association

Antipattern:

- Merge all the properties and behaviors into a single <<Player>> class and not have <<Role>> classes at all. which reduce power of object oriented
- Create roles as a subclasses of the <<Player>> class.

Related pattern:

Abstraction - occurance pattern

References:

OMT book by Rumbaugh (1991)

Scanned by CamScanner

The Singleton Pattern:

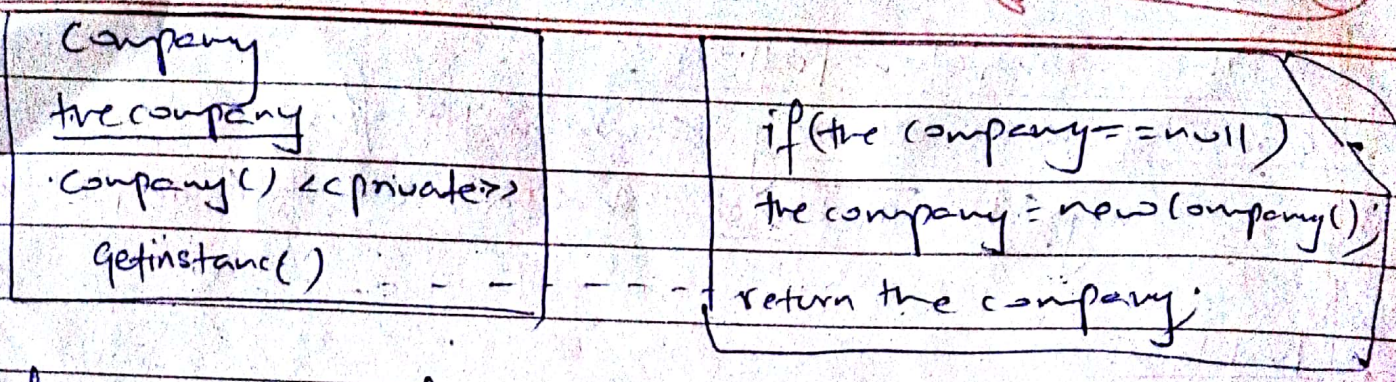
- Context: It is very common to find classes for which only one instance should exist (Singleton).

- Problem:
  - How do you ensure that it is never possible to create more than one instance of a Singleton class?

- Forces:
  - The use of a public constructor cannot quarantee that no more than one instance will be created.
  - The Singleton instance must also be accessible to all classes that require it; therefore it must often be public.

- Solution
  - In a Singleton class, create the following:
    - A private class variable, often called "theInstance" to store the single instance.
    - A public class method (static method) often called "getInstance". The first time this method is called, it creates the single instance and stores it in theInstance. Subsequent calls simply return theInstance.
    - A private constructor, which ensures that no other class will be able to create an instance of the Singleton class.

Scanned by CamScanner



Reference: Gang of four

The Observer Pattern:

Context:

- When an association is created between two classes the code for the classes becomes inseperable.
- If you want to reuse one class, then you also have to reuse the other.
- If change one other should be changed

Problem:

How do you reduce the interconnection between classes, especially between classes that belong to different modules or subsytems.

Forces:

You want to maximize the flexibility of the system to the greatest extent possible.

Solution: Create an abstract class called <<observable>> that maints a collection of <<observer>> instances. The <<observable>> class usually have mechanisms to add and remove observers (instances of <<observer>>), as well as a method notifyObserver that sends an update message to each

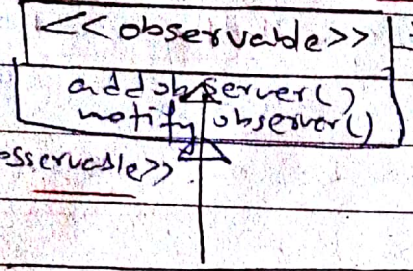
Scanned by CamScanner

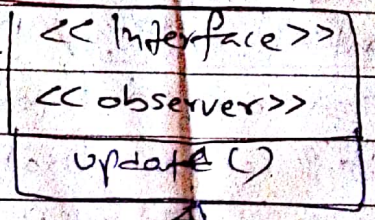
<<observer>>. <<observer>> is an interface defining only an abstract 'update' method. For a class to become observer:

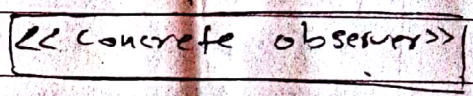
Any class can be made to observe an <<observable>> by: 1) declaring that it implements the interface <<observer>>

Solution 2) By becoming a member of the observer list in the <<observable>> class.

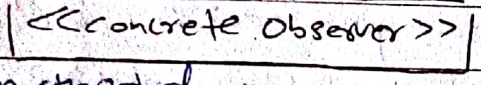
- Any application class can be declared as subclass of <<observable>> class

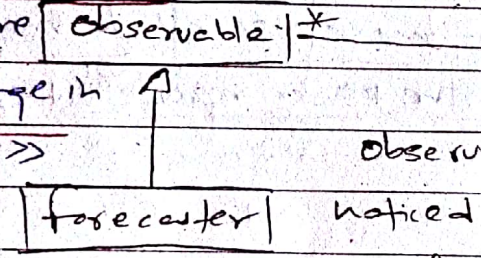


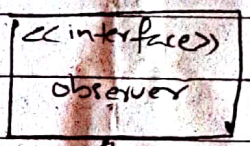


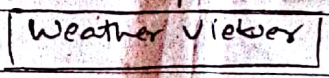


- A call to the update method of <<observer>> is made wherever there is some change in <<observable>>









Observer are notified when new forecast is ready.

- The <<observable>> need not know about the nature of the classes that will receive the update message and what they will do with this information.

Antipatterns

- Connect an observer directly to an observable so that they both have references to each other
- Make the observers subclasses of the observable.

References:

It is one of the 'Gang of Four' patterns.

Scanned by CamScanner

The Delegation Pattern:

Context: - Either because the 'Is-a' rule doesn't apply or you don't want to reuse all the methods of other class.

- You are designing a method in a class.
- You realize that another class has a method which provide the same service..
- ~~Inheritance is not possible by~~ However you don't want to make your class a subclass of that class ~~either~~ and inherit that method.

Problem:

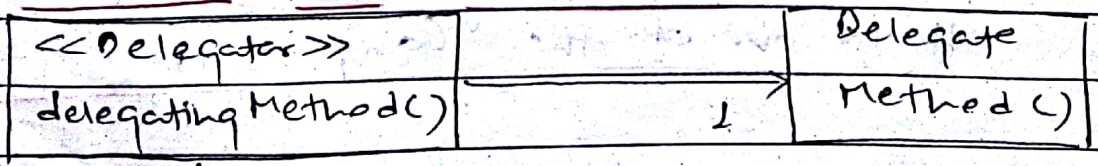
How can you effectively make use of a method that already exists in the other class?

forces:

- You want to minimize development cost by reusing method and complexity too. You want to reduce the linkage between class. And ensure that work is done in most appropriate class.

Solution:

Create a method in the <<Delegator>> class that calls a method in neighbouring << Delegate class >> The << Delegate >> is connected to << Delegator >> by association which is already existed. And may be unidirectional or bidirectional.




```
delegatingMethod()
{
  delegate.method();
}
```

Scanned by CamScanner

Antipatterns:

Inherit the method that is to be reused.

Having many different method in the << Delegator >> instead of one method to call delegate's method.

Access non-neighbouring classes.

Related-Patterns:

Adapter and Proxy patterns.

Refrence: Book of GRASP

Principles that leads to good design:

The overall goals of good design are:

- Increasing profit by reducing cost and increasing revenue
- Ensuring that we actually conform all the requirement
- Accelerating development to meet the deadline.
- Increasing the qualities such as usability, efficiency, reliability, maintainability and reusecability.

Divide and Conquer:

Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things.

In software engineering, the divide and conquer principle is applied in many ways.

The process of development is divided into activities such as requirements gathering, design and testing.

A software system can be divided in many ways:

- A distributed system is divided up into clients and

Scanned by CamScanner

servers.

- A system is divided into sub systems.
- A subsystem is divided into one or more packages
- A package is composed of classes.
- A class is composed of methods.

Dividing the system into pieces has many advantages

- Separate people can work on each part. The original development work therefore can be done in parallel.
- An individual software engineer can specialize in his or her component, becoming expert at it.
- Each individual component is smaller and therefore easier to understand.
- When one part needs to be replaced or changed, this can hopefully be done without replacing other parts.
- Components can be made reuseable.

2. Increase cohesion where possible:

- Cohesion means divide things up, but keep things together that belong together.
- A subsystem or module has high cohesion. It keeps together things that are related to each other and keeps out other things.
- It makes the system easier to understand and change.

Scanned by CamScanner

Types :

1) functional cohesion
2) Layer cohesion
3) Communicational cohesion
4) Sequential cohesion
5) Procedural cohesion
6) Temporal cohesion
7) Utility cohesion.

3) Reduce coupling where possible :

→ Reduce the interdependencies between the modeule or subsystem. It is because

→ when interdependencies exist, changes in one place will require changes somewhere else, which is problematic and time-consuming.

→ Similarly, when we want to reuse one module, we will have to import those which it is coupled.

→ To reduce coupling, we have to reduce the number of connections and strength of connections.

4) keep the level of abstraction as high as possible :

→ The design should allow us to hide the unnecessary details for reducing complexity.

→ Abstractions allow us to understand the essence of something and helps in decision making without knowing unnecessary details.

Scanned by CamScanner

5) Increase reuseability where possible:

-> The design of various aspects of the system should be done in such a way that they can be used again in other contexts, both in our system or in other systems.

-> Reuseability can be obtained by following strategies;

* Generalize your design as much as possible.
* Follow the preceding three design principles
* Design your system to contain books.
* Simplify your design as much as possible.

6) Reuse existing design and code where possible.

-> Reusing design and code allows you to take advantage of the investment we have made in reusable components

-> Normally, it is best to encapsulate the code in a separate method and call it from all the places it is needed.

7) Design for flexibility:

-> Designing for flexibility means actively anticipating changes that a design may have to undergo in the future and preparing for them.

-> The flexibility in the design can be implemented by

- Reducing coupling and increasing cohesion.
- Creating abstraction
- Not hard-coading anything
- Using reuseable code and making code reusable.
- leaving all options open.

Scanned by CamScanner

8) Anticipating obsolescence.

→ Anticipating obsolescence means planning for evolution of the technology or environment. So that the software will continue to run or can be easily changed

→ The following are the strategies for anticipating obsolescence

* Avoid using early releases of technology.
* Avoid using software libraries that are specific to particular environment
* Avoid using undocumented features.
* Avoid using reuseable component or special hardware from smaller companies
* use standard languages and technologies that are supported by multiple vendors.

9) Design for portability:

→ It means to design a software that run on as many platforms as possible

→ It can be achieved by
* avoid the use of facilities that are specific to one particular environment.

10) Design to testability:

→ It means to design software in which testing is easier

→ The most important way to design for testability is to ensure that all the functionality of the code can be executed without going through GUI

Scanned by CamScanner

11) Design Defensively:

- It means the design should trust how others will try to use a component that we are designing
- It can be achieved by:
  * Handle all cases where other people might attempt to use our component improperly.
  * check that all of the inputs of our components are valid.

Scanned by CamScanner

4) The Adapter Pattern:

Context:

- You are building an inheritance hierarchy and you want to use another class written by somebody else into your hierarchy
- Normally the methods of the reused class don't have the same name or argument types as the method in the hierarchy you are creating.
- The reused class may be part of its own inheritance hierarchy.

Problem:

- How do you obtain the power of polymorphism when reusing a class whose methods have the same functionality but do not have the same signature as the other method in the hierarchy?

Forces:

We don't have access to multiple inheritance or you don't want to use it.

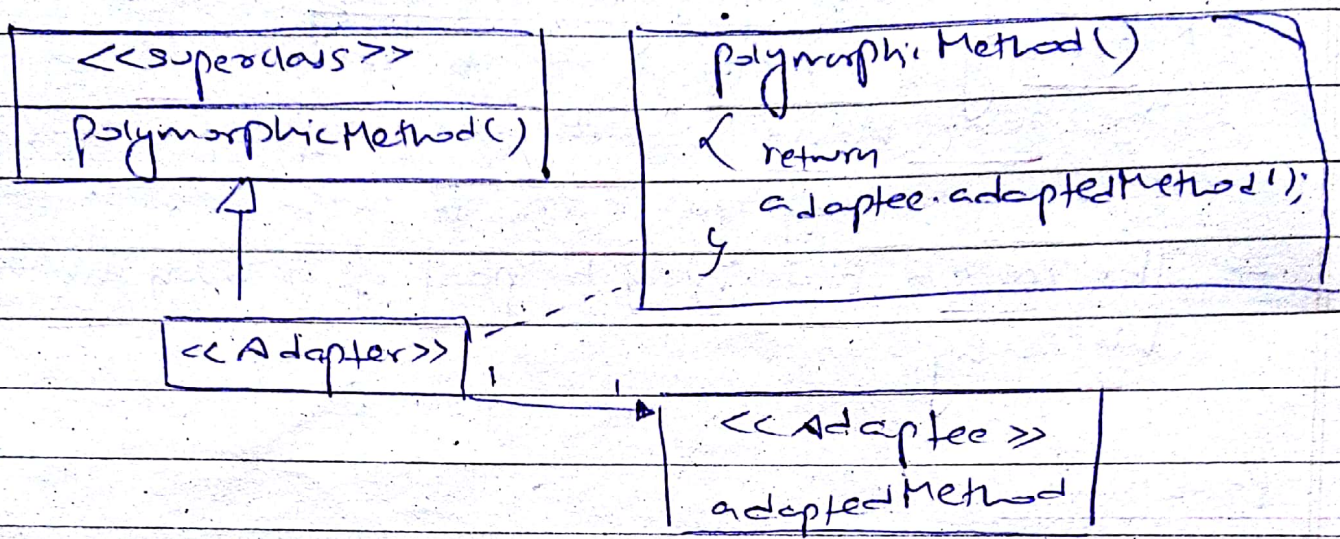
Solution:

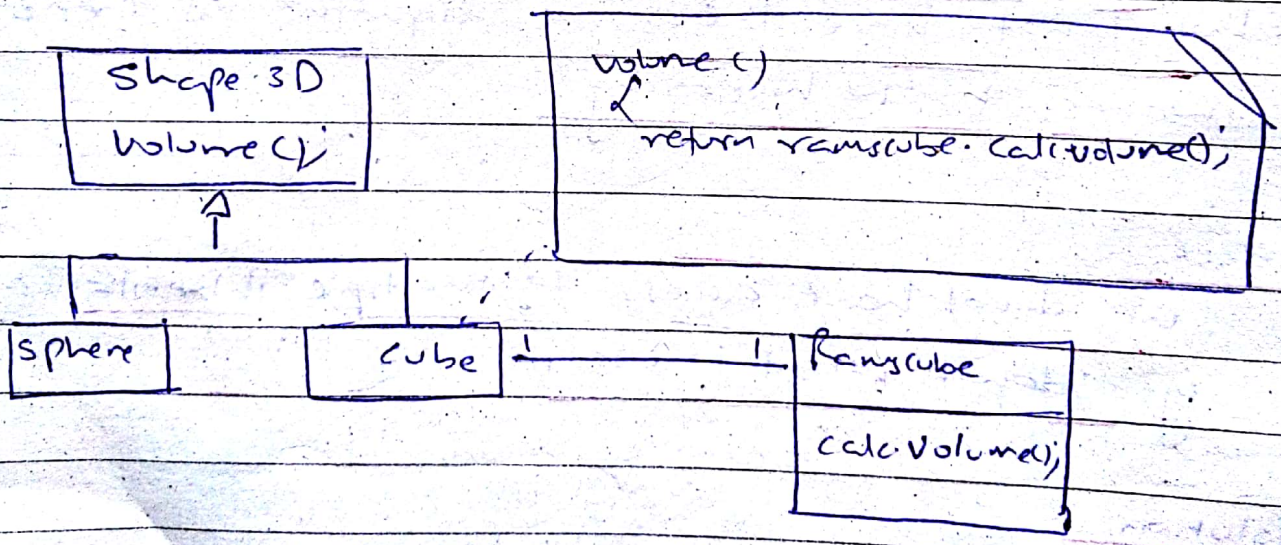
- Rather than directly using the reused class into your inheritance hierarchy, used the adapter class.
- The <<Adapter>> is connected to the reused class by an association. The reused class is called <<Adaptee>>.
- The polymorphic method of the <<adapter>> is deligated

Scanned by CamScanner

to the method of <<Adaptee>>

- The delegate method in the <<Adaptee>> may or may not have the same name as the delegating polymorphic method





Related Pattern: Facade
Proxy

References: The Adapter pattern is one of the 'Gang of four'.

Scanned by CamScanner

The facade Pattern:

Context:

- Often, an application contains several complex packages
- A programmer working with such packages helps to manipulate many different classes.

Problem:

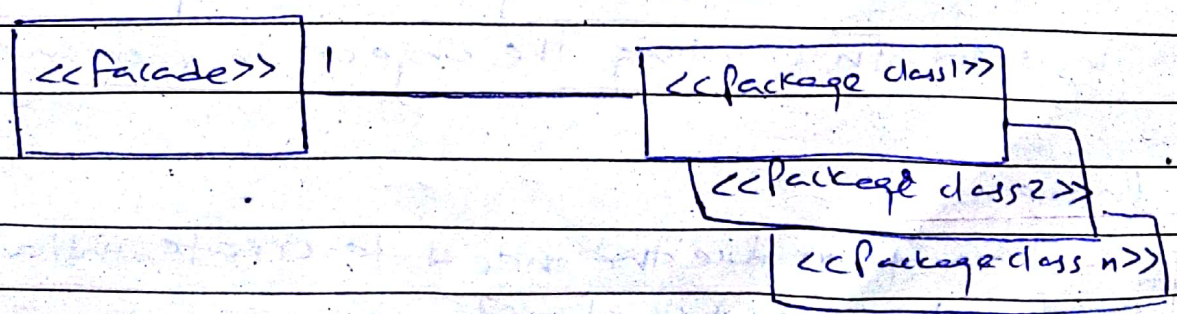
- How do you simplify complex packages?

Forces:

It is hard for a programmer to understand and use an entire subsystem.

Solution:

- Create a special class called a <<facade>> which will simplify the use of the package
- The <<facade>> will contain a simplified set of frequently used methods in the package.
- This ~~results~~ makes the use of package easy and reduces number of dependencies with other packages.



Scanned by CamScanner

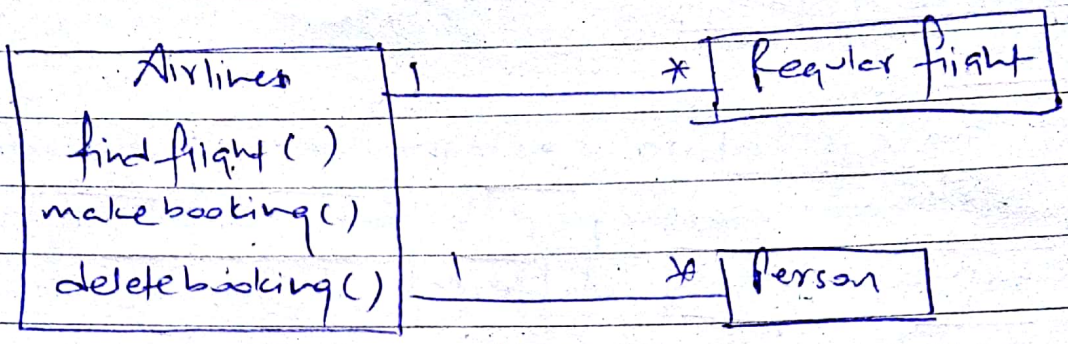


fig: Example of Facade Design Pattern

- Here, Airline is a facade class, findflight() is a method in class Regularflight and makeBooking() and deleteBooking() are the methods of class Person.
- Reference: This pattern is one of the 'Four Gang of Pattern'.

The Proxy Pattern:

Context:

- Often it is time consuming and complicated to create instances of a class.
- we called such classes heavyweight classes.
- There is a time delay and a complex mechanism involved in creating the object in memory.

Problem:

- How to reduce the need to create instances of a heavyweight class?

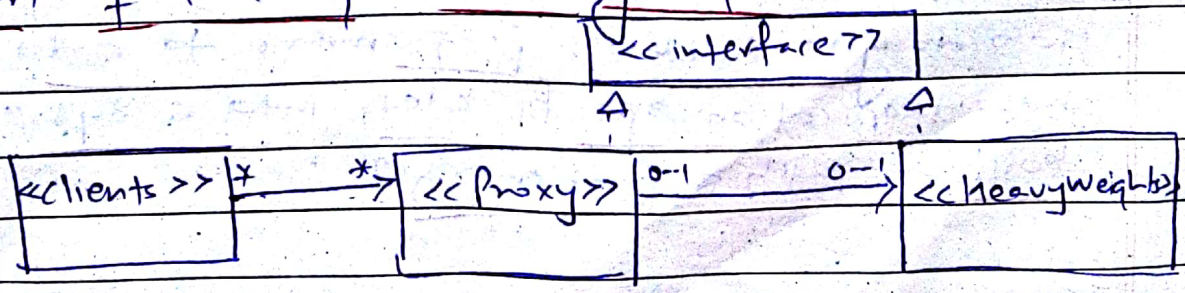
Scanned by CamScanner

Forces:

- We want all the objects in a domain model to be available for programs to use when they execute a system's various responsibilities.
- It is also important for many objects to persist from run to run of the same program.

Solution:

- Create a simpler version of the <<Heavyweight>> class called <<Proxy>>.
- The <<Proxy>> has the same interface as the heavyweight
- The <<proxy>> object normally acts only as a placeholder.
- If any attempt is made to perform an operation on the <<proxy>> then the <<proxy>> delegates the operation to the heavyweight.
- <<Proxy>> is available in memory. So the access is faster.
- Some proxies may have implementation of limited number of operations that can be performed without the effort of loading the heavyweight



Scanned by CamScanner

Antipattern:

- Beginners tend to load the heavyweight class into memory.

Related Pattern:

- Delegation Pattern

References:

The Proxy pattern is one of the "Gang of four" patterns.

b The factory Pattern:

Context:

- You have a reusable framework that needs to create object as part of its work.
- However, the class of the created objects will depend on the application

Problem:

How do you enable a programmer to add a new application specific class into a system built on such a framework?

forces:

You want the benefits of a framework but also retain the flexibility of having the framework create and work with application specific classes.

Scanned by CamScanner

Solution:

- The framework delegates the creation of instances of application specific classes to a specialized class, the factory.
- The factory is a generic interface defined in the framework.
- The factory interface declares a method whose purpose is to create some sub-class of a generic class.

Antipattern:

- Get rid of the factory and instead modify the framework code to force it to always instanciate the object of required application.

References:

The factory pattern is one of the "Gang of four" patterns.

Scanned by CamScanner