Design Patterns

UNIT-I

Define design pattern —

design pattern are evaluated of solution for a problem, which occur over and over again in our environment (like Town planning, Building architecture, and programming).

Alexander christopher firstly discuss patterns in programming using classes and objects instead of walls, pillers.

Who is the father of Design pattern

christopher Alexandres.

In view of Alex christopher every design pattern discussed with help of four essential elements —

1. pattern name
2. problem
3. solution
4. consequences.

⇒ Pattern name — The pattern name is a handle we can use to describe a design problem, its solutions and consequences in a word or two.

- naming a pattern immediately increases our design vocabulary.

- Having a vocabulary for patterns let us talk about them with our colleagues, in our documentation and even to ourselves
- it makes it easier to think about designs and communicate them and their trade-offs other.

Problem

The problem describes when to apply the pattern.

✓ it explains the problem and its context.
✓ it might describe specific design problems such as how to represent algorithms as objects
✓ it might describe class or object structures that are symptomatic of an inflexible design.
✓ the problem will include a list of conditions that must be met before it makes sense to apply them.

Solution:

The solution describes the elements that make up the design, their relationships responsibilities, and collaborations.
✓ the solution doesn't describe a particular concrete design or implementation, because pattern is like a template that can be applied in many different situations.

✓

Consequences:

The consequences are the results and trade-offs of applying the pattern

✓ Though consequences are often unvoiced when we describe design decisions, They are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern.

*/ The consequences for s/w often concern space and time trade-offs.

✓ They may address language and implementation issues as well.

✓ The consequences of a pattern include its impacts on a system's flexibility, extensibility, or portability.

Design patterns in MVC

The model/View/controller (MVC) triad of classes is used to build user interfaces in Small Talk-80.

✓ MVC consists of three kinds of objects. The model is the application object, The View is its screen presentation, and controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them into increase flexibility and reuse.

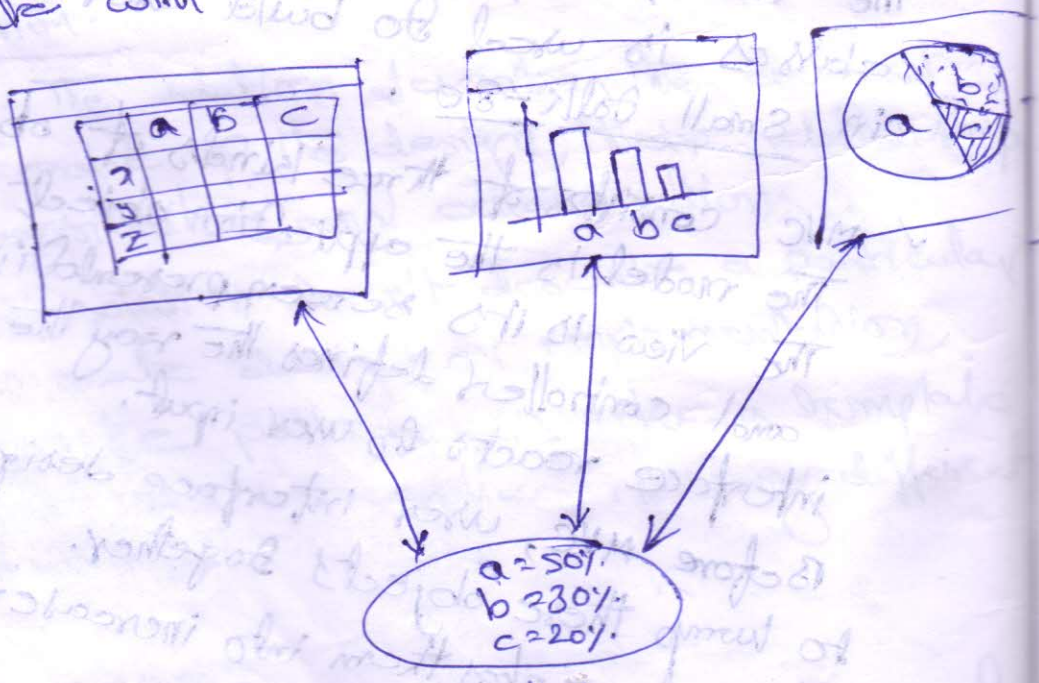
✓ MVC decouples views and models by establishing a subscribe/notify protocol between them.

a view must ensure that its appearance reflects the state of the model.

✓ whenever the model's data changes, the model notifies views that depend on it. In response, each view gets an opportunity update itself.

✓ The following 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 views when its values change, and the views communicate with the model to access these values.



From diagram we can know that decouples view from models.

but the design is applicable to more general problem: decoupling objects so that changes to one can affect any number of others without requiring the changed object to know details of the others. This more general design is described by the observer design pattern.

Another feature of MVC is that views can be nested. MVC supports nested views with the composite view class, a subclass of View. Composite View objects act just like View objects; a composite view can be used whenever a view can be used, but it also contains and manages nested views.

MVC also lets you change the way a view responds to user inputs without changing its visual presentation. you might want to change way it responds to the keyboard, for example or have it use a pop-up menu instead of command keys. MVC encapsulates the response mechanism in a controller object. There is a class hierarchy of controller making it easy to create a new controller as a variation on an existing one.

✓ A view uses an instance of a controller subclass to implement a particular response strategy. To implement a different strategy simply replace the instance with a different kind of controller. It's even possible to change a view's controller at run-time to let the view change the way it responds to user input.

✓ The view-controller relationship is an example of the strategy design pattern. A strategy is an object that represents an algorithm.

✓ It's useful when you want to replace the algorithms either statically or dynamically, when

⇒ MVC uses other design patterns such as Factory Method to specify the default controller class for a view and Decorator to add scrolling to a view. But the main relationships in MVC are given by the Observer, Composite and Strategy design patterns.

= GOF gang of Four persons prepared a template to describing all existing patterns in particular order.

- it helps us to easy remember, study and applying select a pattern for solving the design issues more effectively.
- To reuse the design, we must also record decisions, & alternatives, and trade-offs that led to it.

Template for describing a design pattern:-

Pattern name and classification

The pattern's name conveys the essence of the pattern succintly.

the pattern's classification reflects the schemes (like Health insurance, accident, family insurance, ...)

Intent:

A short statement that answers the following questions:

What does the design pattern do?
What particular design issue or problem does it address? so on..

Also known as

Other well-known name for the pattern, if any.

Motivation:

A scenario that illustrates a design problem and how the class and object structures in the ~~template~~ pattern solve the problem.

the scenario will help us to understand the more abstract description of the pattern that follows.

Applicability:

It discuss the situations in which the design pattern can be applied.

→ it list out few poor designs, how the pattern can address problem.

→ How can you recognize these situations?

Structure:

it is a graphical representation of the classes in the pattern using a notation based on the object modeling technique (OMT). we also use interaction diagrams to illustrate sequences of requests and collaborations between objects.

Participants:

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

Collaboration
it d
collaborat
Consequenc
How c
what a
of usi
Implementa
pi
or techn
when im
ex:
the
Sample coo
it d
we migh
Smallta
/Snow u
Examp
syste
Related patt
speci
to t
betw

Collaborations:

it discuss, how the participants collaborate to carry out their responsibilities

Consequences:

How does the pattern support its objectives what are the trade-offs and results of using the pattern?

Implementation:

pitfalls, hints, or techniques should you be aware of when implementing the pattern?

ex: if we are for client server application then we know RMI concepts CORBA

Sample code:

it discuss on code fragments, how you might implement the pattern in C++ or Smalltalk/Java.

Known uses:

Examples of the pattern found in real systems.

Related patterns:

specifies the other patterns closely related to this one. discuss the major differences between them, with which other patterns should this one be used.

Catalog of Design patterns

A
Abstract factory
Adapter

B
Bridge
Builder

C
Chain of Responsibility
Command
Composite

D
Decorator

F
Facade
Factory method
Flyweight

I
Interpreter
Iterator

M
Mediator
Memento

O
observer

P
Prototype
Proxy

S
Singleton
State
Strategy

T
Template method

V
Visitor

Abstract factory

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

Adapter:
Converts the interface of class into another interface client expect.

A.

Organizing the Catalog of DP's

✓ The classification helps us to learn the patterns in the catalog faster, and it can direct efforts to find new patterns as well.

✓ We classify the patterns by two criteria
1. ~~creational~~ & Purpose — what a pattern does
2. ~~pu~~ Scope

Again under purpose these design patterns are ~~divided in~~ separated into three group. 1. Creational patterns concern the process of object creation.

2. Structural
Structural patterns deal with the composition of classes or objects

3. Behavioral
Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

b) Scope
The second criterion called scope, specifies whether the pattern applies primarily to classes or to objects.

class

class patterns deal with relationship between classes and their subclasses. these relationships are established through inheritance, so they are static - fixed at compile time.

object

object patterns deals with object relationships, between classes and ~~their subclasses~~ which can be changed at run-time and more dynamic.

✓ ~~some patterns are often used together~~

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

Define the following terms:

a) abstract class b) concrete class

c) Delegation d) Frameworks.

e) Toolkits

=> Abstract class:

An abstract class is one whose main purpose is to define a common interface for its subclasses.

or

✓ An abstract class will defer some or all of its implementation to operations defined in subclasses;

✓ An abstract class cannot be instantiated.

✓ The operations that an abstract class declares but doesn't implement are called abstract operations.

Concreate Class - classes that fully implement operations/methods of the class it's a Concreate class.

Delegation:

Delegation refers to evaluating a member (property or method) of one object (the receiver) in the context

to another, original object (the sender).

✓ Delegation can be done explicitly, by passing the sending object to the receiving object, which can be done in any object oriented language

Frame Works

A framework is a set of cooperating classes, that make up a reusable design for a specific class of s/w

ex:- a frame work can be geared toward building graphical editors for different domains like artistic drawing, music composition, and mechanical CAD.

✓ another framework can help us to build compilers for different programming languages and target machines.

✓

10/7/18

How to select a Design pattern

We follow the following approaches to find the appropriat designpattern for solving a problem while design S/w appl'ns.

→ consider
+ ✓ How to a design patterns solve design problems.

it help us to find appropriate object, determine object granularity specify object interfaces, and several other ways in which design patterns solve design problem

+ ✓ their discussions can help as guide for search the right pattern.

+ Scan Intent Sections

Read all the pattern's intent to find one or more that sound relevant to our problem. we can use the classification scheme to narrow our search.

+ Study How Patterns Interrelate: In given figure, we observe the relationships between design patterns graphically.

✓ Studying these relationships can help us to the right pattern or group of patterns.

reffered figure in page no. 12

Fig 1.1

+ Study Patterns of like purpose.

The patterns are divided into three groups by Purpose of patterns i.e creational, structural and behavioral patterns.

each group starts off with introductory comments on the patterns and concludes with a section that compares and contrasts them.

- ~~these~~

+ Examine a cause of redesign:

~~ex~~ some common causes of redesign along with the design patterns that address solution for that. it help to avoid the causes of redesign.

~~Some~~ ~~of~~

✓ ~~Design~~ patterns help you

✓ The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so that they can evolve accordingly.

✓ A design that doesn't take change into account risks major redesign in the future.

✓ Those changes might involve class redefinition and reimplementation, client modification, and retesting.

✓ Redesign affects many parts of the S/W system, and unanticipated changes are invariably expensive.

✓ ~~code~~ each design pattern lets some aspect of system structure vary independently of other aspects thereby making a system more robust to a particular kind of change.

Ex:-

1. Creating an object by specifying a class explicitly
   Specifying a class name when you create an object commits you to a particular implementation instead of a particular interface. This commitment can complicate feature changes.
   To ~~create~~ avoid it, create object indirectly.
   Design patterns: Abstract Factory, Factory Method, Prototype.

2. Dependence on Specific operations -

when we specify a particular operation we commit to one way of satisfying a request. By avoiding hard-coded requests, we make it easier to change the way a request gets satisfied both at compile-time and run-time = 5

Dp's - chain of responsibility, Command

3. Dependence on H/w and S/w platform.

✓ External operating system interface and application programming interfaces (APIs) are different on different H/w and S/w platforms.

S/w that depends on a particular platform will be harder to port to other platforms

✓ It may even be difficult to keep it up to date on its native platform.

✓ for restricting limit dependency on its platform, use design patterns

AbstractFactory.

4. Dependence on object representations or implementations -

client that know how an object is represented, stored, located, or implemented might need to be changed

when the object changes.
Hiding this information from clients keeps changes from cascading, for that we use design patterns -
Abstract Factory, Bridge, Memento, Proxy.

5. Algorithmic dependencies -
Algorithms are often extended, optimized, and replaced during development and reuse.
Objects that depend on an algorithm will have to change when the algorithm changes. therefore algorithms that are likely to change should isolated.
for that DPs - Builder, Iterator, Strategy, Template Method, Visitor.

6. Tight Coupling -
classes that are tightly coupled are hard to reuse in isolation, since they depend on each other.
Tight coupling leads to monolithic systems, where you can't change or remove a class without understanding and changing many other classes. leads to the system hard to learn, port, and maintain.

loose coupling increases the probability that a class can be reused by itself and that a system can be learned, ported, modified, and extended more easily.

for that Abstract Factory, Bridge, chain of responsibility, command, Facade, Mediator, observer.

7. Extending functionality by Subclassing-

for that Design patterns - Bridge, chain of responsibility, composite, Decorator, observer, strategy.

8. Inability to alter classes conveiently

Sometimes we have to modify a class that can't be modified conveniently. perhaps we need the source code and don't have it. or may be any change would require modifying lots of existing subclasses.

Design patterns offer way to modify classes in such circumstances.
DPs — Adapter, Decorator, Visitor.

How to use a design pattern

we follow the several approaches to select a design pattern for solving a particular design problem from 23 patterns.

after selecting the pattern, Here we ~~how to use & study the How pattern use design pattern~~

Here we study a step-by-step approach to applying a design pattern effectively.

1. Read the pattern once through for an overview.
   In this case, we pay more attention to the Applicability and consequences sections.
2. Go back and study the structure, participants, and collaborations sections.
   it helps us to understand the classes and objects in the pattern.

and how they relate to one another.

3. Look at the sample code section.
   it help us to How to implement the pattern.

4. Choose names for pattern participants that are meaningful in the application context.
   ✓ The names of participants in design patterns are usually too abstract to appear directly in an application.
   ✓ it's useful to incorporate the participant name into the name that appears in application.
   ✓ it make the pattern more explicit in the implementation.

   ex: if use the Strategy pattern for a text compositing algorithm, the we kept the names classes either SimpleLayoutStrategy or TeX Layout Strategy.

5. Define the classes.
   w.r.to classes, declare their interfaces, establish their inheritance relationships, and define the instance variables that represent data and object references.

✓ identify existing classes in application that the pattern will effect, and modify them accordingly.

6 Define app'n Specific names for operation in the pattern.

Generally, the names depend on the application.

✓ we kept the name for an operation considering the responsibilities & collaborations associated with each operation,

✓ be consistent in our name conventions.

for example:

'Create' - prefix consistently to denote a factory method.

7 Implement the operations to carry out the responsibilities and collaborations in the pattern.

✓ The implementation section help us to a guide in the ~~implementation~~ implementation

objective

- ✓ Intention of design pattern.
- ✓ Intent of all the 23 design pattern
- ✓ Study how many design pattern involved in implening MVC in Smalltalk more effectively.
- ✓ Study the design pattern roles for designing an application, it must support platform indepent, reusability, flexibility etc.