Unit 3 : Designing the Architectures & Introduction to Design Patterns

Page No. 1
Date: / /

Syllabus Topic :-

- Architecture in Life Cycle -
- Designing the Architecture
- Forming the team structure
- Creating a skeletal system
- Case study - Flight Simulation, Design Patterns: What is Design Pattern? Describing Design Pattern, The catalog of DP, organizing the catalog, How design patterns solves design problems, How to select DP, How to use DP.

Architecture in Life Cycle :-

- Any organizatn that embraces architecture as a foundation for its slw development processes needs to understand its place in the LC.
- Several LC models exist in the literature, but one that puts architecture squarely in the middle of things is Evolutionary Delivery Life Cycle model shown

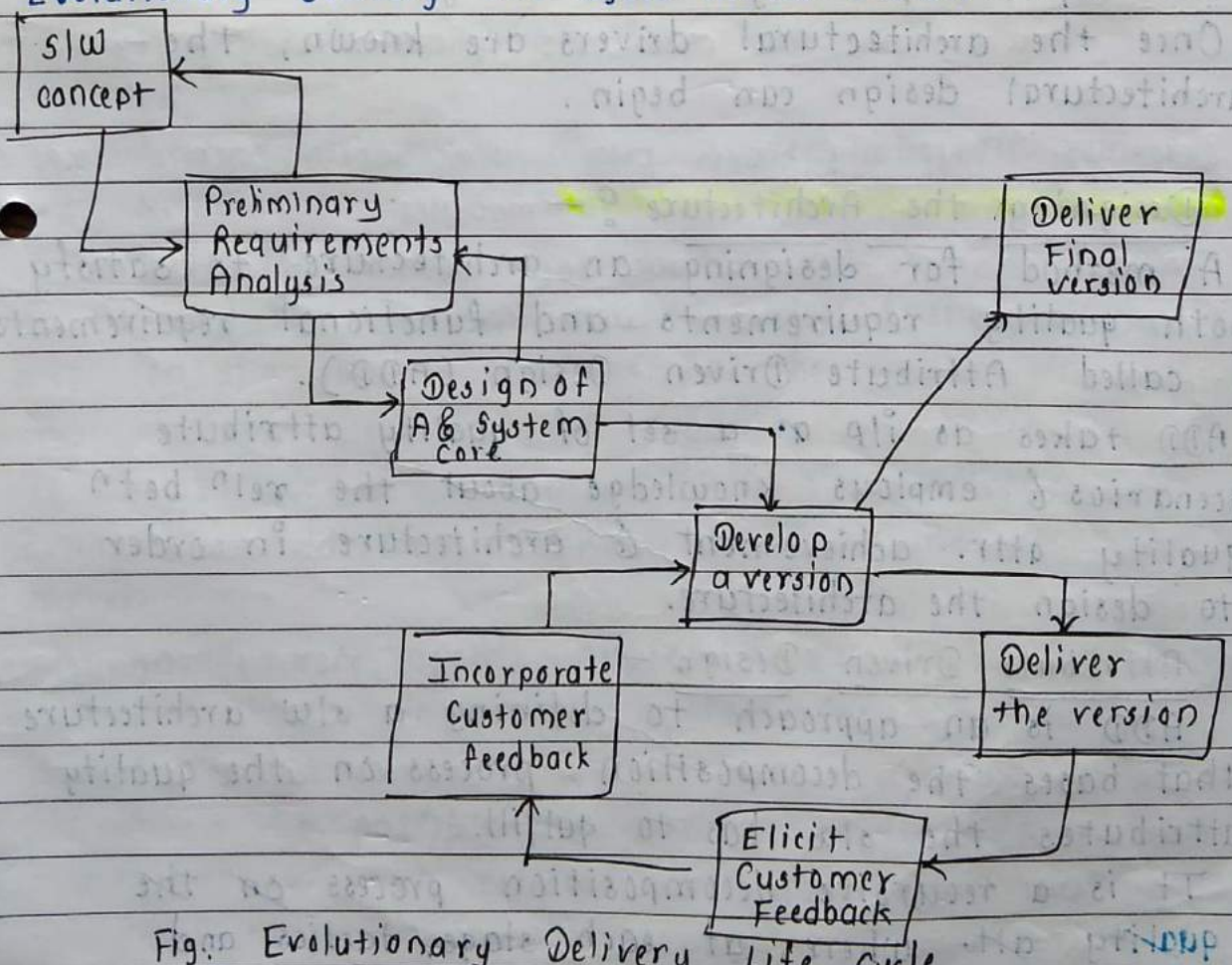


Fig: Evolutionary Delivery Life Cycle

the middle of things is the Evolutionary Delivery Life Cycle model shown in Fig.
- The intent of this model is to get user & customer feedback & iterate thr several releases before the final release.

* When can I Begin Designing?
- The LC model shows the design of the architecture as iterating with preliminary requirement analysis.
- You cannot begin the design until you have some idea of the system requirements.
- An architecture is "shaped" by some collection of functional, quality & business requirements.
- We call these shaping requirements architectural drivers.
- To determine the architectural drivers, identify the highest priority business goals.
- Once the architectural drivers are known, the architectural design can begin.

* Designing the Architecture :-
- A method for designing an architecture to satisfy both quality requirements and functional requirements is called Attribute Driven Design (ADD).
- ADD takes as i/p a set of quality attribute scenarios & employs knowledge about the reln betn quality attr. achievement & architecture in order to design the architecture.

* Attribute Driven Design -
- ADD is an approach to defining a slw architecture that bases the decomposition process on the quality attributes the slw has to fulfill.
- It is a recursive decomposition process ~~or the~~ quality att. where, at each stage, tactics and

and architectural patterns are chosen to satisfy a set of quality scenarios & then functionality is allocated to instantiate the module ~~sys~~ types provided by the pattern.
- The o/p of ADD is the 1st several levels of a module decompositn view of an architecture & other views as appropriate.

• Garage Door Opener Example :—
- Design a product line architecture for a garage door opener with a large home informatn system the opener is responsible for raising & lowering the door via a switch, remote control, or the home informatn system.
- It is also possible to diagnose problems with the opener from within the HIS.
- Input to ADD : a set of requirements
  - Functional requirements as use cases
  - Constraints
  - Quality req. expressed as system specific quality scenarios.
- Scenarios for garage door system
- Device and controls for opening & closing the door are diff. for the various products in the product line
- The processor used in diff. products will differ.
- If an obstacle is (person or object) is detected by the garage door during descent, it must stop within 0.1 sec.
- The garage door opener system needs to be accessible from the home informatn system for diagnosis & administration.
- It should be possible to directly produce an architecture that reflects this protocol.

* ADD steps —

- Steps involved in attribute driven design (ADD)

1. Choose the module to decompose
   - Start with entire system
   - Inputs for this module need to be available
   - Constraints, functional & quality requirements.
2. Refine the module
   a) Choose architectural drivers relevant to this decomposit^n
   b) Choose architectural pattern that satisfies these drivers.
   c) Instantiate modules & allocate functionality from use cases representing using multiple views.
   d) Define interfaces of child modules.
   e) Verify & refine use cases & quality scenarios
3. Repeat for every module that needs further decomposit^n.

• Discussion of the above steps in more detail :

1. Choose The module to Decompose —
   - the following are the modules : system → subsystem → sub module
   - Decomposit^n typically starts with system, which then decompose into subsystem & then into sub-modules
   - In our example, the garage door opener is a system
   - Opener must interoperate with the home informat^n system
2. Refine the module —
   1. Choose Architectural Drivers —
      - Choose the architectural drivers from the quality scenarios & functional requirements.
      - The drivers will be among the top priority requirements for the module
      - In the garage system, the 4 scenarios were architectural drivers.
      - By examining them, we see
        • Real-time performance requirement

• Modifiability requirements to support product line
- Requirements are not treated as equals
- Less important requirements are satisfied within constraints obtained by satisfying more important requirements
- This is a difference of ADD from other architecture design methods

2. Choose Architectural Pattern :-
- For each quality requirement there are identifiable tactics & then identifiable patterns that implement these tactics
- The goal of this step is to establish an overall architectural pattern for the module.
- The pattern needs to satisfy the architectural pattern for the module tactics selected to satisfy the drivers.
- Two factors involved in selecting tactics:
  ✓ Architectural drivers themselves
  ✓ side effects of the pattern implementing the tactic on other requirements.
- This yields the foll. tactics :
  → Semantic coherence & informatn hiding. Seperate responsibilities dealing with the user Int, interface, comm^n & sensors into their own modules.
- Increase computational efficiency, The performance-critical computatns should be made as efficient as possible.
- Schedule wisely, The performance-critical computat^ns should be scheduled to ensure that achievement of the timing deadline.

3. Instantiate Modules And Allocate Functionality Using Multiple Views :-
• Instantiate modules :-
- The non-performance critical module of fig. 7.2 becomes instantiated as diagnosis and raising / lowering door modules in fig. 7.3.
- We also identify several responsibilities of the virtual m/c : communicatn & sensor reading &

and actuator control. This yields two instances of the virtual mlc that are also shown in fig. 7.3.

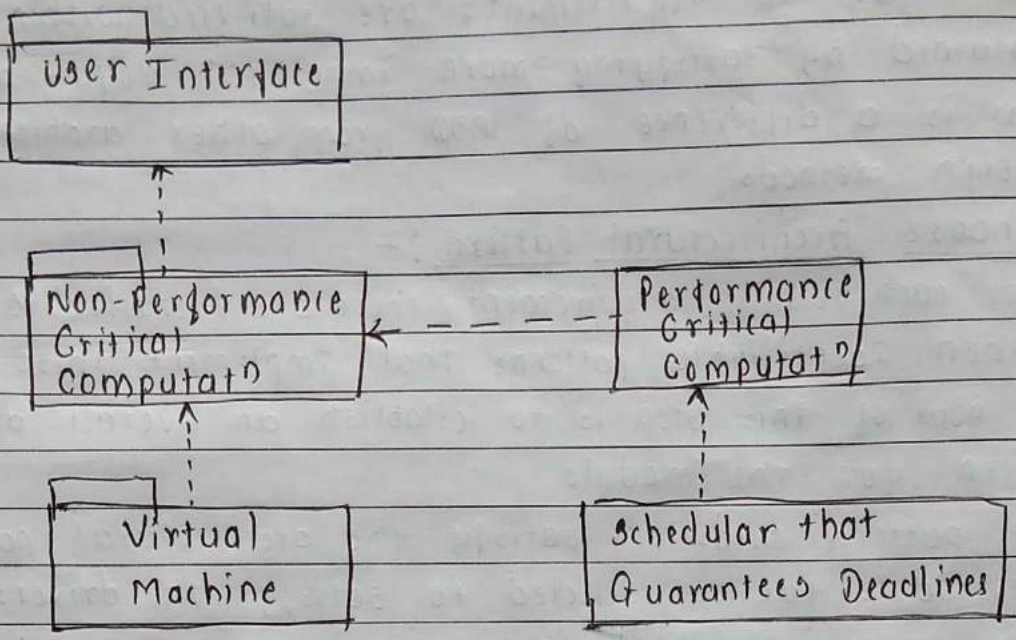


Fig. 7.2 Architectural pattern that utilizes tactics to achieve garage door drivers

- Allocate functionality –
  - Assigning responsibilities to the children in a decompositn also leads to the ~~delivery~~ discovery of necessary informatn exchange. At this point in the design, it is not important to define how the info is exchonged.
  - Is the informatn pushed or pulled? Is it passed as a msg or a call parameter? These are all questns that need to be answered later in the design process.
  - At this point only the informatn itself & the producer & consumer roles are of interest.
- Represent the architecture with multiple views –
  - Module decompositn view
  - Concurrency view
    - Two users doing similar things at the same time.
    - One user performing multiple activities simultaneously
    - Starting up the system
    - Shutting down the system

- ~~Shutting down the s~~
- Deployment view

4. Define Interfaces of Child Modules :-
- It documents what this module provides to others.
- Analyzing the decomposition into the 3 views provides interaction for the interface.
  - Module view :
    - Producers / consumers relations
    - Patterns of communication
  - Concurrency View :
    - Interactions among threads
    - Synchronization information
  - Deployment view
    - Hardware requirement
    - Timing —ll—
    - Communication —ll—

5. Verify And Refine Use Cases And Quality Scenarios As Constraints For the child modules —
- Functional Requirements —
- Using functional requirements to verify and refine.
- Decomposing functional requirements assigns responsibilities to child modules.
- We can use these responsibilities to generate use cases for the child module.
  - ✓ User interface :
    - Handle user requests
    - Translate for raising / lowering module
    - Display responses
  - ✓ Raising / lowering door module
    - Control actuators to raise / lower door
    - Stop when completed opening or closing
  - ✓ Obstacle detection -
    - Recognice when object is detected
    - Stop when completed opening or closing

- Obstacle detection :
  - Recognize when object is detected
  - Stop or reverse the closing of the door
- Communication virtual m|c
  - Manage comm^n with house informat^n system (HIS)
- Scheduler
  - Guarantee that deadlines are met when obstacle is detected
- Sensor / actuator VM
  - Manage interact^{ns} with sensors / actuators
- Diagnosis
  - Manage diagnosis interact^n with HIS

o Constraints :-

- The decomposit^n satisfies the constraint
  - OS constraint → satisfied if child module is OS
- The constraint is ~~inherited by the child module~~ satisfied by a single module
  - Constraint is inherited by the child module
- The constraint is satisfied by a coll^n of child modules
  - E.g., using client & server modules to satisfy a communicat^n constraint.

* Forming the Team structure/s :-

- Once the architecture is accepted we assign teams to work on diff. portions of the design & development.
- Once architecture for the system under construction has been agreed on, teams are allocated to work on the major modules & a work break down structure is created that reflects those teams.
- ~~For~~ Each team then creates its own internal work practices.
- For large systems, the teams may belong to diff. subcontractors.

- Teams adopt "work practices" including
  - Team communicat^n via website / bulletin boards
  - Naming conventions for files
  - Configurat^n / revision control system
  - Quality assurance & testing procedure
- The teams within an organizat^n work on modules, & thus within team high level of comm^n is necessary.

* Creating a skeletal System :-

- ① Develop a skeletal system for the incremental cycle
- Classical s/w engg. practice recommends → "stubbing out"
- Use the architecture as a guide for the implement^n sequence.
- First implement the s/w that deals with execution & interaction of architectural components.
  - Communicat^n bet^n components
  - Sometimes this is just install third-party middle-ware.
- Then add functionality
  - By risk-lowering
  - Or by availability of staff
- Once the elements providing the next increment of functionality have been chosen, you can employ the uses structure to tell you what additional s/w should be running correctly in the system to support that functionality.

* Case Study - Flight Simulation :-

- Among the most sophisticated s/w systems :
  - Highly distributed
  - Hard real-time performance requirements
  - Modifiability : changes in requirements, simulated vehicles & environment
  - Scalability : continuou improvement of the simulat^n of the real world
- Initial stages of the ABC for the flight simulator

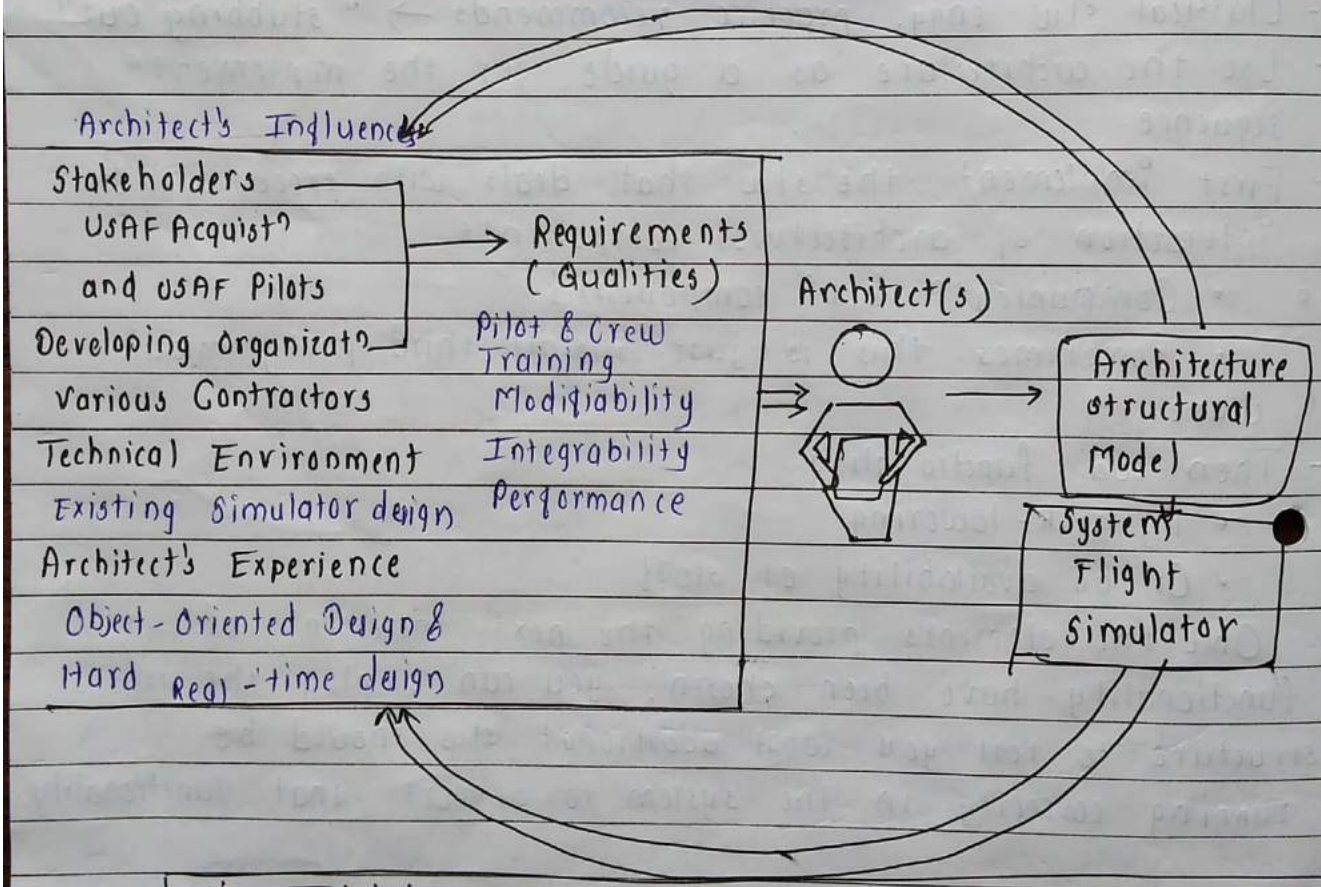
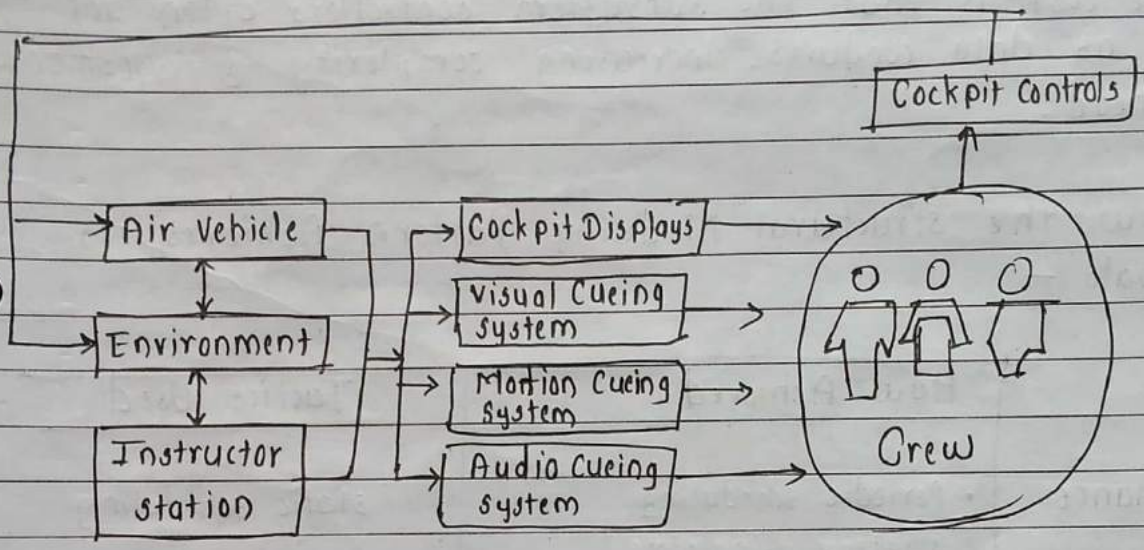


Fig. Initial stages of the ABC for the flight simulator

* Highlights —
  - Architectural soln
  - Treatment of time -
    - Periodic Time Management (for real-time)
    - Event-Based Time Management (non-RT)

- Skeletal System: Architecture Prototype
  - Flight Simulator has 6 module types
  - What are the benefits?

• Reference model for flight simulator –



Key: Data Flow →

* Integrability :–
  - Both the data connections and the control connectns have been minimized
  - Integrating another controller has been reduced to a problem that is linear, not exponential.
  - Integrating 2 subsystems is again reduced to ensuring that the 2 pass data consistently.
  - A driven concern in large systems.
  - Those developed by distributed teams or separate organizatns.

• Applicable Tactics –
  - Keep interfaces small, simple & stable.
  - Loose coupling or minimal dependencies betn elements.
  - Use a component framework.

* Pattern : a structural Model –
  - Simplicity & similarity of systems substructures
  - Decoupling of data & control passing strategies from computn

- Minimizing module types
- The pattern includes an object-oriented design to model the subsystems & controller children of the air vehicle.

• **Cost / Benefit Ratio** –

– The cost is that the subsystem controllers often act purely as data conduits, increasing complexity & performance overhead.

* How the structural Modeling Pattern Achieves its Goals –

Goal	How Achieved	Tactics Used
Performance	• Periodic scheduling strategy using time budgets	static scheduling
Integrability	• Seperatⁿ of computⁿ from coordinatⁿ	Restrict commⁿ
	• Indirect data & control connectⁿs	Use intermediary
Modifiability	• Few module types	Restrict commⁿ
	• Physically based decompositⁿ	Semantic coherence
		Interface stability

* Design Patterns :-

- The purpose is to record experience in designing object-oriented sw as design patterns.
- Each DP systematically names, explains & evaluates an important & recurring design in object-oriented systems.
- DP makes it easier to reuse successful design and architectures.
- DP can even improve the documentatn & maintenance of existing systems.

* What is a Design Pattern ?

- Alexander says, " Each pattern describes a problem which occurs over and over again in our environment, & then describes the core of the soln to that problem."
- Even though Alexander was talking about patterns in buildings & towns, what he says is true about object-oriented design patterns.
- In general, a pattern has four essential elements :

1. The pattern name is a handle we can use to describe a design problem, its solns & consequences in a word or two.
   - Naming a pattern immediately increases our design vocabulary.
   - It lets us design at a higher level of abstractn.
   - It makes it easier to think about designs & to communicate them & their trade-offs to others.
   - Finding good names has been one of the hardest parts of developing our catalog.
2. The problem describes when to apply the pattern. It explains the problem & its context.
   - It might describes 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.

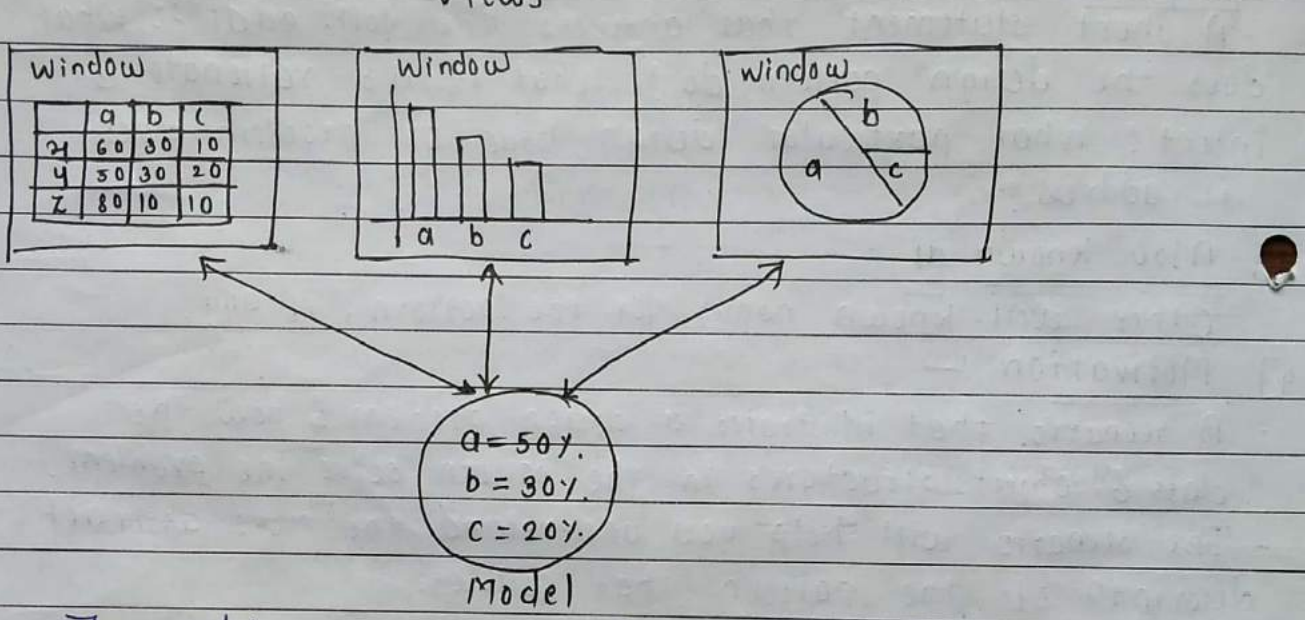
3. The solution describes the elements that make up the design, their relationships, responsibilities and collaborations. The soln doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many diff. solns.

4. The consequences are the results & trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives & for understanding the cost & benefits of applying the pattern.
- The consequences for s/w often concern space & time trade-offs.

• Design Patterns in Smalltalk MVC :-

- The Model/View/Controller (MVC) triad of classes is used to build user interfaces in Smalltalk-80.
- At the design patterns inside MVC should help you see what we mean by the term "pattern".
- MVC consists of 3 kinds of objects. The model is the appln object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user i/p.
- Before MVC, user interface designs tended to lump these objects together.
- MVC decouples them to increase flexibility & reuse.
- It decouples views & models by establishing a subscribe/notify protocol betn 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 to update itself.
- This approach lets you attach multiple views to a model to provide diff. presentatns.
- The foll. diagram shows a model & 3 views.



- The model contains some data values, & the views defining a spreadsheet, histogram, & pie chart display these data in various ways.
- The model communicates with its views when its values change, & the views communicate with the model to access these values.
- Another feature of MVC is that views can be nested.

* Describing Design Patterns :-

- How do we describe design patterns? Graphical notatns, while important & useful aren't sufficient.
- They simply capture the end product of the design process & relnships betn classes & objects.
- We describe design patterns using a consistent format. Each pattern is divided into sectns accd to the foll. template.

1] Pattern Name & Classification :-

- The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of ur design vocabulary.
- The pattern's classificatn reflects the scheme.

2] Intent :-

- A short statement that answers the foll. questns: What does the design pattern do? What is its rationale & intent? What particular design issue or problem does it address?

Also known as :-

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

4] Motivation :-

- A scenario that illustrates a design problem & how the class & object structures in the pattern solve the problem.
- The scenario will help you understand the more abstract descriptn of the pattern that follows.

5] Applicability :-

- What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can u recognize these situatns?

6] Structure :-

- A graphical representatn of these classes in the pattern using a notation based on the Object Modeling Technique (OMT)
- We also use interaction diagrams to illustrate sequences of requests & collaborations betn objects.

7] Participants :-

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

8] Collaborations :-

- How the participants collaborate to carry out their responsibilities.

9] Consequences :-
- How does the pattern support its objectives? What are the trade-offs & results of using the pattern? What aspect of system structure does it let u vary independently?

10] Implementation :-
- What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

11] Sample Code :-
- Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

12] Known Uses :-
- Examples of the pattern found in real systems. We include at least two examples from diff. domains.

13] Related Patterns :-
- What DP are closely related to this one? What are the important differences? With which other patterns should this one be used?

* The Catalog of Design Patterns :-

- The catalog beginning on page 79 contains 23 design patterns.
- Their names & intents are listed next to give you an overview.

1] Abstract Factory :-
Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

2] Adapter :-
Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

3] Bridge :-
Decouple an abstractn from its implementatn so that the two can vary independently.

4] Builder –

Separate the construct^{n} of a complex object from its represent^{n} so that the same construct^{n} process can create diff. represent^{ns}.

5] 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 & pass the request along the chain until an object handles it.

6] Command :–

Encapsulate a request as an object, thereby letting you parameterize clients with diff. requests, queue or long requests.

7] Composite :–

- Compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects & composit^{ns} of objects uniformly.

8] Decorator :–

- Attach additional responsibilities to an object dynamically Decorators provide a flexible alternative to subclassing for extending functionality.

9] Facade :–

- 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.

10] Factory Method :–

- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Factory method lets a class defer instantiat^{n} to subclasses.

11] Flyweight :–

- Use sharing to support large nos. of fine-grained objects efficiently.

12] Interpreter :-
- Given a lang., define a representation for its grammer along with an interpreter that uses the represent^n to interpret sentences in the lang.

13] Iterator :-
- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying represent^n.

14] 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.

15] Memento :-
- Without violating encapsulat^n, capture & externalize an object's internal state so that the object can be restored to this state later.

16] Observer :-
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified & updated automatically.

17] Prototype :-
- Specify the kinds of objects to create using a prototypical instance & create new objects by copying this prototype.

18] Proxy :-
- Provide a placeholder for another object to control access to it.

19] Singleton :-
- Ensure a class only has one instance, and provide a global point of access to it.

20] State :-
- Allow an object to alter its behavior when its internal state changes.

21] Strategy :-
- Define a family of algo., encapsulate each one, & make them interchangeable.

22] Template Method :-
- Define the skeleton of an algo. in an operatn, deferring some steps to subclasses.

23] Visitor :-
- Represent an operatn to be performed on the elements of an object structure. Visitor lets you define a new operatn without changing the classes of the elements on which it operates.

* Organizing the Catalog :-

- Design patterns vary in their granularity & level of abstraction.
- Because there are many DP, we need a way to organize them.
- This section classifies design patterns so that we can refer to families of related patterns.
The classificatn helps to learn the patterns in the catalog faster.
- We classify DP by two criteria (Table 1.1)
The 1st criterion, called purpose, reflects what a pattern does.
- Patterns can have either creational, structural, or behavioral purpose.
- Creational patterns concern the process of object creation.
- Structural patterns deal with the compositn of classes or objects.
- Behavioral patterns characterize the ways in which classes or objects interact & distribute responsibility.

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

Table 1.1 : Design Pattern space

- The 2nd criterion called scope, specifies whether the pattern applies primarily to classes or to objects.
- Class patterns deal with rel^n ships bet^n classes & their subclasses.
- These rel^n ships are established thr inheritance, so they are static fixed at compile-time.
- Object patterns deal with object rel^n ships, which can be changed at run-time & are more dynamic.
- Note that most patterns are in the Object scope.
- Creational class patterns defer some part of object creatn to subclasses, while Creational object patterns defer it to another object.
- The structural class patterns use inheritance to compose classes, while the structural object patterns describe ways to assemble objects.
- The behavioral class patterns use inheritance to describe algorithms & flow of control.

* How Design Patterns Solve Design Problems :-

- Design patterns solve many of the day-to-day problems object-oriented designers face, and in many diff. ways.
- Here are several of these problems & how design patterns solve them.

1] Finding Appropriate Objects :-

- Object-oriented programs are made up of objects. An object packages both data & the procedures that operate on that data.
- The procedures are typically called Methods or operations.
- An object performs an operat^n when it receives a request (or message) from a client.
- Requests are the only way to get an object to execute an operat^n.
- Operations are the only way to change an object's internal data.
- Because of these restrictions, the object's internal state is said to be encapsulated; it cannot be accessed directly, and its represent^n is invisible from outside the object.
- The hard part about object-oriented design is decomposing a system into objects.
- Object-oriented design methodologies favor many diff. approaches.
- You can write a problem statement, single out the nouns & verbs & create corresponding classes & operations.
- Many objects in a design come from the analysis model.
- Object-oriented designs often end up with classes that have no counter parts in the real world.

2] Determining Object Granularity :-

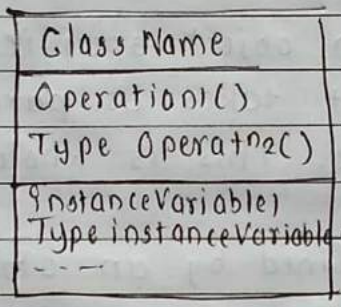
- Objects can vary tremendously in size & number. They can represent everything down to the hardware or all the way up to entire applns.
- How do we decide what should be an object ?
- Design patterns address this issue as well. The Facade pattern describes how to represent complete subsystems as objects, & Flyweight pattern describes how to support huge nos. of objects at the finest granularities.
- Other design patterns describe specific ways of decomposing an object into smaller objects.

3] Specifying Object Interfaces :-

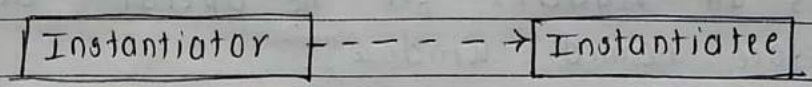
- Every operatn declared by an object specifies the operation's name, the objects it takes as parameters, & the operation's return value. This is known as the operation's signature.
- The set of all signatures defined by an object's operatns is called the interface to the object.
- An object's interface characterizes the complete set of requests that can be sent to the object.
- Any request that matches a signature in the object's interface may be sent to the object.
- A type is a name used to denote a particular interface.
- We speak of an object as having the type "window" if it accepts all requests for the operatns defined in the interface named "Window".
- Two objects of the same type need only share parts of their interfaces.
- Interfaces can contain other interface as subsets.

4] Specifying Object Implementations :-

- An object's implementat^n is defined by its class.
- The class specifies the object's internal data & representat^n & defines the operations the object can perform.
- Our OMT based notat^n depicts a class as a rectangle with the class name in bold.
- Operat^{ns} appear in normal type below the class name.
- Any data that the class defines comes after the operations. Lines separate the class name from the operat^{ns} & the operat^{ns} from the data.

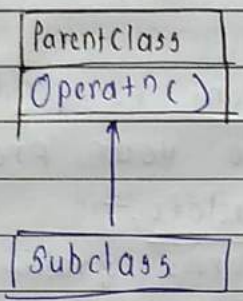


- Return types & instance variable types are optional, since we don't assume a statically typed implement^n lang.
- Objects are created by instantiating a class. The object is said to be an instance of the class.
- The process of instantiating a class allocated storage for the object's internal data & associates the operat^{ns} with these data.



- New classes can be defined in terms of existing classes using class inheritance.
- When a subclass inherits from a parent class, it includes the def^{ns} of all the data & operat^{ns} that parent class defines.

- Objects that are instances of the subclass will contain all data defined by the subclass & its parent classes, & they'll be able to perform all operatns defined by this subclass & its parent classes.



- An abstract class is one whose main purpose is to define a common interface for its subclasses.
- An abstract class will defer some or all of its implemnt to operatns defined in subclasses. hence an abstract class can not be instantiated.
- The operatns that an abstract class declares but doesn't implement are called abstract operatns. Classes that aren't abstract are called concrete classes.
- Subclasses can refine and redefine behaviors of their parent classes.
- A class may override an operatn defined by its parent class.

* How to select a Design Pattern :-

- With more than 20 design patterns in the catalog to choose from, it might be hard to find the one that addresses a particular design problem, especially if the catalog is new and unfamiliar to you.
- Here are several diff. approaches to finding the design pattern :

i) Consider how design patterns solve design problems - how design patterns help you find appropriate objects,

determine object granularity, specify object interfaces, & several other ways in which design patterns solve design problems.

ii) Scan Intent sections :-
- The Intent sectns from all the patterns in the catalog.
- Read thr each pattern's intent to find one or more that sound relevant to your problem.

iii) Study how patterns ~~tr~~ Interrelate :-
- ~~Shou~~ Relns betn design patterns graphically.
Studying these reln ships can help direct you to the right pattern or group of patterns.

iv) Study patterns of like purpose :-
- The catalog has 3 chapters, one for creational patterns another for structural patterns, and a third for behavioral patterns.

v) Examine a cause of redesign :-
- Look at the causes of redesign starting on page 24 to see if your problem involves one or more of them.
- Then look at the patterns that help you avoid the causes of redesign.

vi) Consider what should be variable in your design :-
- This approach is the opposite of focusing on the causes of redesign. Instead of considering what might force a change to a design, consider what you want to be able to change without redesign.

* How to Use a Design Pattern :-

- Once you've picked a DP, how do you use it? Here's a step-by-step approach to applying a design pattern effectively.

1. Read the pattern once thr for an overview -
- Pay particular attention to the Applicability & Consequences sectns to ensure the pattern is right for ur problem.

2. Go back & study the structure :-
- Participants & collaboratns sectns. Make sure you understand the classes & objects in the pattern & how they relate to one another.
3. Look at the sample code section to see a concrete example of the pattern in code.
studying the code helps u learn how to implement the pattern.
4. Choose names for pattern participants that are meaningful in the applⁿ context -
The names for participants in DP are usually too abstract to appear directly in an applⁿ.
5. Define the classes :-
- Declare their interfaces, establish their inheritance relns & define the instance variables that represent data & object references
6. Define applⁿ-specific names for operations in the pattern -
The names generally depend on the applⁿ.
- Use the responsibilities & collaboratns associated with each operatn as a guide
7. Implement the operatns to carry out the responsibilities & collaboratns in the pattern.
- The implementn sectn offers hints to guide you in the implementn.