MAR GREGORIOS COLLEGE OF ARTS & SCIENCE

Block No.8, College Road, Mogappair West, Chennai – 37

Affiliated to the University of Madras
Approved by the Government of Tamil Nadu
An ISO 9001:2015 Certified Institution

PG DEPARTMENT OF COMPUTER SCIENCE

SUBJECT NAME: OBJECT ORIENTED ANALYSIS AND DESIGN

SUBJECT CODE : PED2A

SEMESTER: II

PREPARED BY: PROF. N.VAISHALI

SUBJECT: OBJECT ORIENTED ANALYSIS AND DESIGN

Unit 1: System Development - Object Basics - Development Life Cycle - Methodologies -Patterns - Frameworks - Unified Approach – UML.

Unit-2: Use-Case Models - Object Analysis - Object relations - Attributes - Methods – Class andObject responsibilities - Case Studies.

Unit 3: Design Processes - Design Axioms - Class Design - Object Storage – ObjectInteroperability - Case Studies.

Unit-4: User Interface Design - View layer Classes - Micro-Level Processes - View LayerInterface - Case Studies.

Unit-5: Quality Assurance Tests - Testing Strategies - Object orientation on testing - Test Cases- test Plans - Continuous testing - Debugging Principles - System Usability - Measuring UserSatisfaction - Case Studies.

Recommended Texts

(i) Ali Bahrami, Reprint 2009, Object Oriented Systems Development, Tata McGraw Hill , International Edition.

Reference Books

(i) G. Booch, 1999, Object Oriented Analysis and design, 2nd Edition, Addison Wesley, Boston

(ii) Roger S.Pressman, 2010, Software Engineering A Practitioner's approach, Seventh Edition, Tata McGraw Hill, New Delhi.

(iii) Rumbaugh, Blaha, Premerlani , Eddy, Lorensen, 2003, Object Oriented Modeling And design , Pearson education, Delhi.

TABLE OF CONTENTS

S.NO	TITLE	PAGENO
UNIT I	**INTRODUCTION**	
1.1	System Development	5
1.2	Object Basics	6
1.3	Development Life Cycle	16
1.4	Methodologies	19
1.5	Patterns	25
1.6	Frameworks	29
1.7	Unified Approach	38
1.8	UML	39
UNIT II		
2.1	Use-Case Models	52
2.2	Object Analysis	68
2.3	Object relations	95
2.4	Attributes	104
2.5	Methods	105
2.6	Class and Object responsibilities	106
UNIT III		
3.1	Design Processes	114
3.2	Design Axioms	115
3.3	Class Design	117
3.4	Object Storage	126
3.5	Object Interoperability	127
UNIT IV		

4.1	User Interface Design.	140
4.2	View layer Classes	142
4.3	Micro-Level Processes	145
4.4	View Layer Interface	147
UNIT V		
5.1	Quality Assurance Tests	179
5.2	Testing Strategies	179
5.3	Object orientation on testing	181
5.4	Test Cases	189
5.5	Test Plans	190
5.6	Continuous testing	191
5.7	Debugging Principles	192
5.8	System Usability	193
5.9	Measuring User Satisfaction	197

UNIT-1

System Development

We know that the Object-Oriented Modeling (OOM) technique visualizes things in an application by using models organized around objects. Any software development approach goes through the following stages −

- Analysis,
- Design, and
- Implementation.

In object-oriented software engineering, the software developer identifies and organizes the application in terms of object-oriented concepts, prior to their final representation in any specific programming language or software tools

Phases in Object-Oriented Software Development

The major phases of software development using object–oriented methodology are object-oriented analysis, object-oriented design, and object-oriented implementation.

Object–Oriented Analysis

In this stage, the problem is formulated, user requirements are identified, and then a model is built based upon real–world objects. The analysis produces models on how the desired system should function and how it must be developed. The models do not include any implementation details so that it can be understood and examined by any non–technical application expert.

Object–Oriented Design

Object-oriented design includes two main stages, namely, system design and object design.

System Design

In this stage, the complete architecture of the desired system is designed. The system is conceived as a set of interacting subsystems that in turn is composed of a hierarchy of interacting objects, grouped into classes. System design is done according to both the system analysis model and the proposed system architecture. Here, the emphasis is on the objects comprising the system rather than the processes in the system.

Object Design

In this phase, a design model is developed based on both the models developed in the system analysis phase and the architecture designed in the system design phase. All the classes required are identified. The designer decides whether −

- new classes are to be created from scratch,
- any existing classes can be used in their original form, or
- new classes should be inherited from the existing classes.

The associations between the identified classes are established and the hierarchies of classes are identified. Besides, the developer designs the internal details of the classes and their associations, i.e., the data structure for each attribute and the algorithms for the operations.

Object–Oriented Implementation and Testing

In this stage, the design model developed in the object design is translated into code in an appropriate programming language or software tool. The databases are created and the specific hardware requirements are ascertained. Once the code is in shape, it is tested using specialized techniques to identify and remove the errors in the code.

Object Basics

The object model visualizes the elements in a software application in terms of objects. In this chapter, we will look into the basic concepts and terminologies of object–oriented systems.

Classes and Objects:

Nature of Object, Relationships among objects, nature of a class,

Relationshipamong classes, interplay of classes and objects, Identifying classes and objects, Importance of proper classification, Identifying classes and objects, Key abstractions and mechanisms

An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable.

State

The state of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties.

Consider a vending machine that dispenses soft drinks. The usual behavior of such objects is thatwhen someone puts money in a slot and pushes a button to make a selection, a drink emerges from themachine. What happens if a user first makes a selection and then puts money in the slot? Most vendingmachines just sit and do nothing because the user has violated the basic assumptions of their operation.Stated another way, the vending machine was in a state (of waiting for money) that the user ignored (bymaking a selection first). Similarly, suppose that the user ignores the warning light that says, "Correctchange only," and puts in extra money. Most machines are user-hostile; they will happily swallow theexcessmoney.

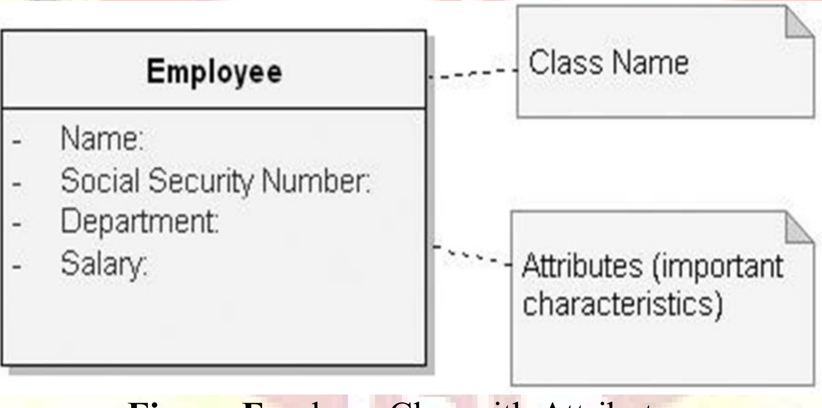


Figure: EmployeeClasswith Attributes

Employee	Employee
- Name: = Tom	- Name: = Kaitlyn
- Social Security Number:	- Social Security Number:
- Department:	- Department:
- Salary:	- Salary:

Figure: EmployeeObjectsTomandKaitlyn

Behavior

Behavior is how an object acts and reacts, in terms of its state changeable state of object affect itsbehavior. In vending machine, if we don't deposit change

sufficient for our selection, then the machinewill probably do nothing.So behavior of an object is a function of its state as well as the operationperformeduponit.Thestateof anobjectrepresentsthecumulativeresultsof itsbehavior.

Anoperationissomeactionthatoneobjectperformsonanotherinordertoelicitareaction.

For example, a client might invoke the operations append and pop to grow and shrink a queue object,respectively. A client might also invoke the operation length, which returns a value denoting the size ofthequeueobjectbutdoesnotalterthestateofthe queue itself.

Message passing is one part of the equation that defines the behavior of an object; our definition forbehavioralsonotesthatthestateofanobjectaffectsitsbehavioraswell.

Operations

Anoperationdenotes aservicethataclass offerstoitsclients.Aclient performs5kinds ofoperationsuponanobject.

Inpractice,wehavefoundthataclienttypicallyperformsfivekindsofoperati onsonanobjectThethreemostcommonkindsof operationsarethefollowing:

- **Modifier**:Anoperationthataltersthestateofanobject.
- **Selector:** An operationthataccessesthestateofanobjectbutdoesnotalter thestate.
- **Iterator**: An operation that permits all parts of an object to be accessed in some well definedorder. In queue example operations, clear, append, pop, remove) are modifies, const functions(length,isempty,frontlocation)areselectors.

Two other kinds of operations are common; they represent the infrastructure necessary to create anddestroyinstancesof aclass.

- Constructor: Anoperationthat createsanobjectand/or initializes itsstate.
- Destructor:Anoperationthatfreesthestateofanobjectand/ordestroystheobject itself.

RolesandResponsibilities

Aroleisamaskthatanobjectwears andsodefinesa contractbetweenanabstractionandits clients. Responsibilities are meant to convey a sense of the purpose of an object and its place in thesystem.The responsibilitiesofanobjectare alltheservicesitprovidesforallofthecontractsitsupports”.

The state and behavior of an object collectively define the roles that an object may play in the world,whichinturnfulfilltheabstraction’s responsibilities.

Examples:

1 bank account may have the role of a monetary asset to which the account owner may deposit orwithdraw money. However, to a taxing authority, the account may play the role of an entity whosedividendsmustbereportedonannually.

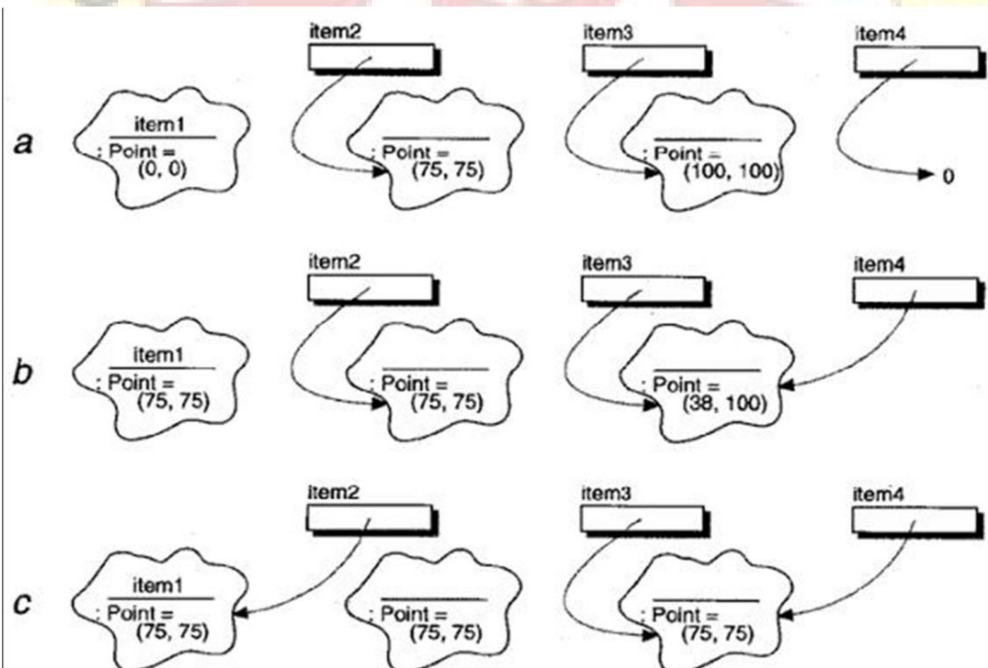
2 To a trader, a share of stock represents an entity with value that may be bought or sold; to a lawyer, thesameshare denotesalegalinstrumenttowhichareattachedcertainrights.
3 In the course of one day, the same person may play the role of mother, doctor, gardener, and moviecritic.

Identity

Identityisthatpropertyofanobjectwhich distinguishesitfromallothers.

Example:

Consider a class that denotes a display item. A display item is a common abstraction in all GUI-centricsystems: It represents the base class of all objects that have a visual representation on some window andso captures the structure and behavior common to all such objects. Clients expect to be able to draw,select,andmovedisplayitems,aswellasquerytheirselectionstateandlocation.Eachdisplayitem has locationdesignatedbythe coordinatesx andy. First declaration creates four names and 3 distinct objects in 4 diff location. Item 1 is the name of adistinct display item object and other 3 names denote a pointer to a display item objects. Item 4 is no suchobjects, we properly say that item 2 points to a distinct display item object, whose name we may properlyrefer to indirectly as * item2. The unique identity (but not necessarily the name) of each object inpreserved over the



lifetime of the object, even when its state is changed. Copying, Assignment, andEqualityStructuralsharingtakesplacewhenthe identityofanobjectisaliasedtoasecondname.

RelationshipsamongObjects

Objects contributetothebehaviorofasystembycollaboratingwithoneanother. E.g.objectstructureofanairplane. The relationship between any two objects encompasses the assumptions that each makes about theother including what operations can be performed. There

are Two kinds of objects relationships are linksandaggregation.

Links

- Alinkdenotesthespecificassociationthroughwhichoneobject(theclient)appliestheservicesofanotherobject(thesupplier) orthrough whichareobjectmaynavigatetoanother.
- Alinebetweentwoobject iconsrepresentsthe existenceofpass alongthispath.
- Messagesareshown asdirectedlinesrepresentingthedirectionofmessagepassingbetweentwoobjectsis typicallyunidirectional,maybebidirectionaldataflowin eitherdirectionacrossalink.

Asaparticipationinalink,anobjectmayplayone of threeroles:

- **Controller:** This object can operate on other objects but is not peratedon by otherobjects. Insomecontexts,thetermsactiveobject andcontroller areinterchangeable.
- **Server:**Thisobjectdoesn'toperateonotherobjects;itisonlyoperatedonbyother objects.
- **Proxy:** This object can both operate on other objects and be operated on by other objects. Aproxyis usuallycreatedtorepresenta real-worldobjectinthe domainof theapplication.

<u>Object</u>

An object is a real-world element in an object–oriented environment that may have a physical or a conceptual existence. Each object has −

- Identity that distinguishes it from other objects in the system.
- State that determines the characteristic properties of an object as well as the values of the properties that the object holds.
- Behavior that represents externally visible activities performed by an object in terms of changes in its state.

Objects can be modelled according to the needs of the application. An object may have a physical existence, like a customer, a car, etc.; or an intangible conceptual existence, like a project, a process, etc.

Class

A class represents a collection of objects having same characteristic properties that exhibit common behavior. It gives the blueprint or description of the objects that can be created from it. Creation of an object as a member of a class is called instantiation. Thus, object is an instance of a class.

The constituents of a class are −

- A set of attributes for the objects that are to be instantiated from the class. Generally, different objects of a class have some difference in the values of the attributes. Attributes are often referred as class data.
- A set of operations that portray the behavior of the objects of the class. Operations are also referred as functions or methods.

Example

Let us consider a simple class, Circle, that represents the geometrical figure circle in a two–dimensional space. The attributes of this class can be identified as follows −

- x–coord, to denote x–coordinate of the center
- y–coord, to denote y–coordinate of the center
- a, to denote the radius of the circle

Some of its operations can be defined as follows −

- findArea(), method to calculate area
- findCircumference(), method to calculate circumference
- scale(), method to increase or decrease the radius

During instantiation, values are assigned for at least some of the attributes. If we create an object my_circle, we can assign values like x-coord : 2, y-coord : 3, and a : 4 to depict its state. Now, if the operation scale() is performed on my_circle with a scaling factor of 2, the value of the variable a will become 8. This operation brings a change in the state of my_circle, i.e., the object has exhibited certain behavior.

Encapsulation and Data Hiding

Encapsulation

Encapsulation is the process of binding both attributes and methods together within a class. Through encapsulation, the internal details of a class can be hidden from outside. It permits the elements of the class to be accessed from outside only through the interface provided by the class.

Data Hiding

Typically, a class is designed such that its data (attributes) can be accessed only by its class methods and insulated from direct outside access. This process of insulating an object's data is called data hiding or information hiding.

Example

In the class Circle, data hiding can be incorporated by making attributes invisible from outside the class and adding two more methods to the class for accessing class data, namely −

- setValues(), method to assign values to x-coord, y-coord, and a
- getValues(), method to retrieve values of x-coord, y-coord, and a

Here the private data of the object my_circle cannot be accessed directly by any method that is not encapsulated within the class Circle. It should instead be accessed through the methods setValues() and getValues().

Message Passing

Any application requires a number of objects interacting in a harmonious manner. Objects in a system may communicate with each other using message passing. Suppose a system has two objects: obj1 and obj2. The object obj1 sends a message to object obj2, if obj1 wants obj2 to execute one of its methods.

The features of message passing are −

- Message passing between two objects is generally unidirectional.
- Message passing enables all interactions between objects.
- Message passing essentially involves invoking class methods.
- Objects in different processes can be involved in message passing.

Inheritance

Inheritance is the mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities. The existing classes are called the base classes/parent classes/super-classes, and the new classes are called the derived classes/child classes/subclasses. The subclass can inherit or derive the attributes and methods of the super-class(es) provided that the super-class allows so. Besides, the subclass may add its own attributes and methods and may modify any of the super-class methods. Inheritance defines an "is – a" relationship.

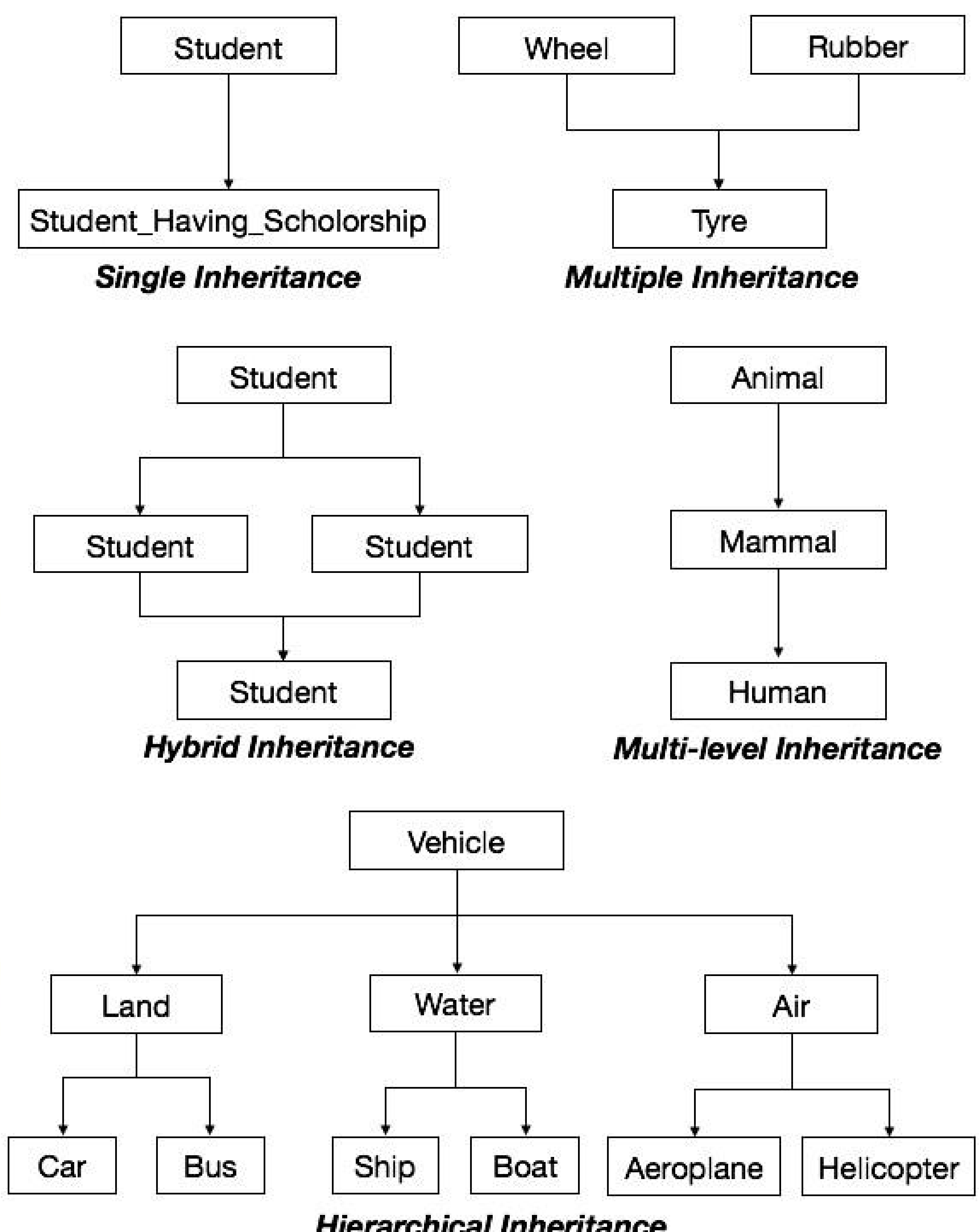
Example

From a class Mammal, a number of classes can be derived such as Human, Cat, Dog, Cow, etc. Humans, cats, dogs, and cows all have the distinct characteristics of mammals. In addition, each has its own particular characteristics. It can be said that a cow "is – a" mammal.

Types of Inheritance

- Single Inheritance − A subclass derives from a single super-class.
- Multiple Inheritance − A subclass derives from more than one super-classes.

- Multilevel Inheritance − A subclass derives from a super-class which in turn is derived from another class and so on.
- Hierarchical Inheritance − A class has a number of subclasses each of which may have subsequent subclasses, continuing for a number of levels, so as to form a tree structure.
- Hybrid Inheritance − A combination of multiple and multilevel inheritance so as to form a lattice structure.
- The following figure depicts the examples of different types of inheritance.



- **Polymorphism**

Polymorphism is originally a Greek word that means the ability to take multiple forms. In object-oriented paradigm, polymorphism implies using operations in different ways, depending upon the instance they are operating upon. Polymorphism allows objects with different internal structures to have a common external interface. Polymorphism is particularly effective while implementing inheritance.

Example

Let us consider two classes, Circle and Square, each with a method findArea(). Though the name and purpose of the methods in the classes are same, the internal implementation, i.e., the procedure of calculating area is different for each class. When an object of class Circle invokes its findArea() method, the operation finds the area of the circle without any conflict with the findArea() method of the Square class.

Generalization and Specialization

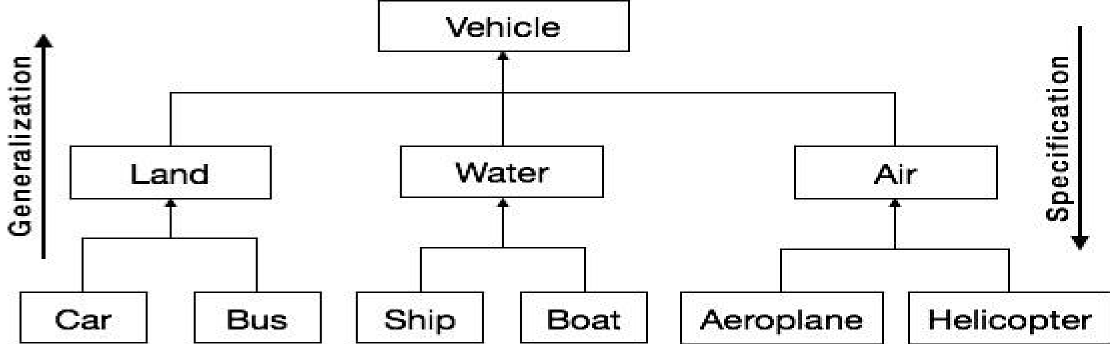
Generalization and specialization represent a hierarchy of relationships between classes, where subclasses inherit from super-classes.

Generalization

In the generalization process, the common characteristics of classes are combined to form a class in a higher level of hierarchy, i.e., subclasses are combined to form a generalized super-class. It represents an “is – a – kind – of” relationship. For example, “car is a kind of land vehicle”, or “ship is a kind of water vehicle”.

Specialization

Specialization is the reverse process of generalization. Here, the distinguishing features of groups of objects are used to form specialized classes from existing classes. It can be said that the subclasses are the specialized versions of the super-class.The following figure shows an example of generalization and specialization.



Links and Association

Link

A link represents a connection through which an object collaborates with other objects. Rumbaugh has defined it as “a physical or conceptual connection between objects”. Through a link, one object may invoke the methods or navigate through another object. A link depicts the relationship between two or more objects.

Association

Association is a group of links having common structure and common behavior. Association depicts the relationship between objects of one or more classes. A link can be defined as an instance of an association.

Degree of an Association

Degree of an association denotes the number of classes involved in a connection. Degree may be unary, binary, or ternary.

- A unary relationship connects objects of the same class.
- A binary relationship connects objects of two classes.
- A ternary relationship connects objects of three or more classes.

Cardinality Ratios of Associations

Cardinality of a binary association denotes the number of instances participating in an association. There are three types of cardinality ratios, namely −

- One–to–One − A single object of class A is associated with a single object of class B.
- One–to–Many − A single object of class A is associated with many objects of class B.
- Many–to–Many − An object of class A may be associated with many objects of class B and conversely an object of class B may be associated with many objects of class A.

Aggregation or Composition

Aggregation or composition is a relationship among classes by which a class can be made up of any combination of objects of other classes. It allows objects to be placed directly within the body of other classes. Aggregation is referred as a "part–of" or "has–a" relationship, with the ability to navigate from the whole to its parts. An aggregate object is an object that is composed of one or more other objects.

Example

In the relationship, "a car has–a motor", car is the whole object or the aggregate, and the motor is a "part–of" the car. Aggregation may denote −

- Physical containment − Example, a computer is composed of monitor, CPU, mouse, keyboard, and so on.
- Conceptual containment − Example, shareholder has–a share.

Benefits of Object Model

Now that we have gone through the core concepts pertaining to object orientation, it would be worthwhile to note the advantages that this model has to offer.

The benefits of using the object model are −

- It helps in faster development of software.
- It is easy to maintain. Suppose a module develops an error, then a programmer can fix that particular module, while the other parts of the software are still up and running.
- It supports relatively hassle-free upgrades.
- It enables reuse of objects, designs, and functions.
- It reduces development risks, particularly in integration of complex systems.

Software Development Life Cycle

The software development process consists of

(1) Analysis – Translates the user needs into system requirements and responsibilities.
(2) Design – It begins with a problem statement and ends with a detailed design that can be transformed into an operational system.
(3) Implementation – It regines the detailed design into the system deployment that will satisfy the users needs.
(4) Testing – Two basic approaches to system testing, they are (i) Test according to how it has been built for. (ii) What it should do?
(5) Maintenance.
Object Oriented Systems Development Life Cycle (SDLC)
This is also known as Classic Life Cycle Model (or) Linear Sequential Model (or) Waterfall Method. This model has the following activities.

- **<u>System/Information Engineering and Modeling</u>**

As software is always of a large system (or business), work begins by establishing the requirements for all system elements and then allocating some subset of these requirements to software. This system view is essential when the software must interface with other elements such as hardware, people and other resources. System is the basic and very critical requirement for the existence of software in any entity. So if the system is not in place, the system should be engineered and put in place. In some cases, to extract the maximum output, the system should be re-engineered and spruced up. Once the ideal system is engineered or tuned, the development team studies the software requirement for the system.

- **Software Requirement Analysis**

This process is also known as feasibility study. In this phase, the development team visits the customer and studies their system. They investigate the need for possible software automation in the given system. By the end of the feasibility study, the team furnishes a document that holds the different specific recommendations for the candidate system. It also includes the personnel assignments, costs, project schedule, target dates etc.. The requirement gathering process is intensified and focused specially on software. To understand the nature of the program(s) to be built, the system engineer or "Analyst" must understand the information domain for the software, as well as required function, behavior, performance and interfacing. The essential purpose of this phase is to find the need and to define the problem that needs to be solved.

System Analysis and Design

In this phase, the software development process, the software's overall structure and its nuances are defined. In terms of the client/server technology, the number of tiers needed for the package architecture, the database design, the data structure design etc.. are all defined in this phase. A software development model is thus created. Analysis and Design are very crucial in the whole development cycle. Any glitch in the design phase could be very expensive to solve in the later stage of the software development. Much care is taken during this phase. The logical system of the product is developed in this phase.

- **Code Generation**

The design must be translated into a machine-readable form. The code generation step performs this task. If the design is performed in a detailed manner, code generation can be accomplished without much complication. Programming tools like compilers, interpreters, debuggers etc.. are used to generate the code. Different high level programming languages like C, C++, Pascal, Java are used for coding. With respect to the type of application, the right programming language is chosen.

- Testing

Once the code is generated, the software program testing begins. Different testing methodologies are available to unravel the bugs that were committed during the previous phases. Different testing tools and methodologies are already available. Some companies build their own testing tools that are tailor made for their own development operations.

- Maintenance

The software will definitely undergo change once it is delivered to the customer. There can be many reasons for this change to occur. Change could happen because of some unexpected input values into the system. In addition, the changes in the system could directly affect the software

operations. The software should be developed to accommodate changes that could happen during the post implementation period.

Prototyping Model

This is a cyclic version of the linear model. In this model, once the requirements analysis is done and the design for a prototype is made, the development process gets started. Once the prototype is created, it is given to the customer for evaluation. The customer tests the package and gives his/her feed back to the developer who refines the product according the customer's exact expectation. After a finite number of iterations, the final software package is given to the customer. In this methodology, the software is evolved as a result of periodic shuttling of information between the customer and developer. This is the most popular development model in the contemporary IT industry. Most of the successful software products have been developed using this model – as it is very difficult to comprehend all the requirements of a customer in one shot. There are many variations of this model skewed with respect to the project management styles of the companies. New versions of a software product evolve as a result of prototyping.

Rapid Application Development (RAD) model

The RAD models a linear sequential software development process that emphasizes an extremely short development cycle. The RAD model is a "high speed" adaptation of the linear sequential model in which rapid development is achieved by using a component-based construction approach. Used primarily for information systems applications, the RAD approach encompasses the following phases,

- Business modeling,
- Data modeling,
- Process modeling,
- Application generation,
- Testing and turnover.

Component Assembly Model

Object technologies provide the technical framework for a component based process model for software engineering. The object oriented paradigm emphasizes the creation of classes that encapsulate both data and the algorithm that are used to manipulate the data. If properly designed and implemented, object oriented classes are reusable across different applications and computer based system architectures. Component Assembly Model leads to software reusability. The integration/assembly of the already existing software components accelerates the development process. Nowadays many component libraries are available on the Internet. If the right components are chosen, the integration aspect is made much simpler.

Methodologies

TheimportanceofModeling:

A model is a simplification of reality. A model provides the blueprints of a system. Everysystem may be described from different aspects using different models, and each model isthereforeasemanticallyclosedabstractionofthesystem.

A model may be structural, emphasizing the organization of the system, or it may bebehavioral, emphasizing the dynamics of the system. We build models so that we can betterunderstandthe system we aredeveloping.

Throughmodeling,weachievefouraims.

- Modelshelpustovisualizeasystemasitisoraswewantittobe.
- Modelspermitusto specifythestructureorbehaviorofasystem.
- Modelsgiveusatemplatethat guidesusinconstructingasystem.
- Modelsdocumentthedecisionswehavemade.

Modelingis notfor big systems. Even the software equivalentof a dog house canbenefit from some modeling. For example if you want to build a dog house ,you can start with apile of lumber, some nails, and a basic tools, such as a hammer,saw,and a tape measure. In fewhours,with littlepriorplanning,youwilllikelyend up a with a dog house that's reasonably functional. finally your will happy and get a lessdemandingdog.

if we want to build a house for your family , you can start with a pile of lumber, somenails, and a basic tools. But it's going to take you a lot longer, and your family will certainly bemore demanding than the dog.If you want to build a quality house that meets the needs ofyourfamily andyouwillneedtodraw someblueprints.

PrinciplesofModeling:
UMLisbasicallyamodelinglanguage;henceitsprincipleswillalsoberelatedtomodelingconcepts.Here are fewbasicprincipalofUML.

First:"Thechoiceofwhatmodelstocreatehasaprofoundinfluenceonhowaproblemisattackedandhow asolution isshaped"

In other words ,choose your models well. The right models will brilliantly illuminate themost wicked development problems. The wrong models will mislead you, causing you to focusonirrelevantissues.

Second:"Everymodelmaybeexpressedatdifferentlevelsofprecision".

Bestapproachtoagivenproblemresultsinabestmodel.Iftheproblemiscomplexmechanizedlevelofapp roach&iftheproblemissimpledecentapproachisfollowed.

Third: "Thebest modelsareconnectedtoreality."

Themodelbuiltshouldhavestrongresemblancewiththesystem.

Fourth:"Nosinglemodelissufficient.Everynontrivialsystemisbestapproachedthroughasmallsetofn early independentmodels."

If you constructing abuilding, there is no single set of blueprints that reveal all itsdetails. At the very least, you will need floor plans,elevations,electical plans, heating plans, andplumbingplans.

Object-OrientedModeling:

Insoftware , there are several waysto approachesa model.Thetwomostcommonwaysare

1 Algorithmicperspective
2 Object-Orientedperspective

Algorithmicperspective:

Inthisapproach,themainbuildingblocksofallsoftwareistheprocedureorfuncton.This view leads developers to focus on issues of control and decomposition of largeralgorithmsintosmallerones.

Object-Orientedperspective:

In this approach, the main building blocks of all software is the object or class. Simplyput, an object is a thing. A class is a description ofa set of common objects. Every object hasidentity,stateandbehavior.

For example, considera simple a three-tier -architecture for a billing system, involving auserinterface,middleware, andadatabase.Intheuserinterface,youwillfindconcreteobjects, such as buttons, menus, and dialog boxes. In the database, you will find concreteobjects ,such as tables. In the middle layer ,you will find objects such as transitions and businessrules.

Object modelling develops the static structure of the software system in terms of objects. It identifies the objects, the classes into which the objects can be grouped into and the relationships between the objects. It also identifies the main attributes and operations that characterize each class.

The process of object modelling can be visualized in the following steps −

- Identify objects and group into classes
- Identify the relationships among classes
- Create user object model diagram
- Define user object attributes
- Define the operations that should be performed on the classes
- Review glossary

Dynamic Modelling

After the static behavior of the system is analyzed, its behavior with respect to time and external changes needs to be examined. This is the purpose of dynamic modelling.

Dynamic Modelling can be defined as “a way of describing how an individual object responds to events, either internal events triggered by other objects, or external events triggered by the outside world”.

The process of dynamic modelling can be visualized in the following steps −

- Identify states of each object
- Identify events and analyze the applicability of actions
- Construct dynamic model diagram, comprising of state transition diagrams
- Express each state in terms of object attributes
- Validate the state–transition diagrams drawn

Functional Modelling

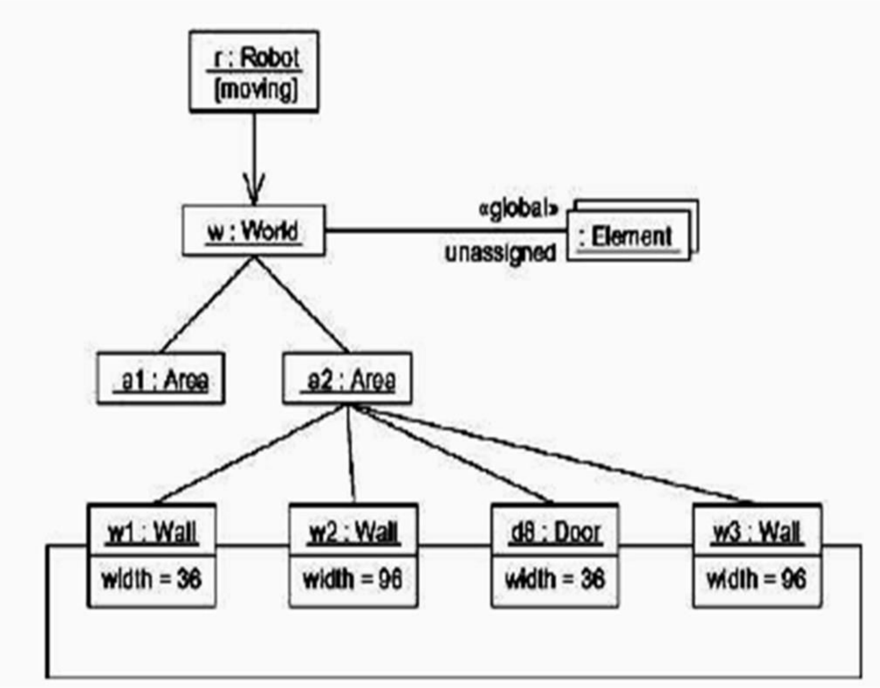
Functional Modelling is the final component of object-oriented analysis. The functional model shows the processes that are performed within an object and how the data changes as it moves between methods. It specifies the meaning of the operations of object modelling and the actions of dynamic modelling. The functional model corresponds to the data flow diagram of traditional structured analysis.

CommonModelingTechniquesofobjectdiagrams:

ModelingObjectStructures

Tomodelanobjectstructure,

- Identify the mechanism you'd like to model. A mechanism represents some function orbehavior of the part of the system you are modeling that results from the interaction ofasocietyofclasses,interfaces,andotherthings.
- Foreachmechanism,identifytheclasses,interfaces,andotherelementsthatparticipateinthiscollaboration;identifythe relationshipsamong thesethings,as well.
- Consider one scenario that walks through this mechanism. Freeze that scenario at amomentintime,andrendereachobjectthat participatesinthemechanism.
- Expose the state and attribute values of each such object, as necessary, to understandthescenario.
- Similarly, expose the links among these objects,



For example, Figureshows a setof objectsdrawn from the implementation of anautonomous robot. This figure focuses on some of the objects involved in the mechanism usedby the robot to calculate a model of the world in which it moves. There are many more objectsinvolved in a running system, but this diagram focuses on only those abstractions that aredirectlyinvolvedincreatingthisworldview.

As this figure indicates, one object represents the robot itself (r, an instance of Robot),and r is currently in the state marked moving. This object has a link to w, an instance of World,whichrepresentsanabstractionoftherobot'sworldmodel.Thisobjecthasalinkto amultiobject that consists of instances of Element, which represent entities that the robot hasidentified but not yet assigned in its world view. These elements are marked as part of therobot'sglobalstate.

At this moment in time, w is linked to two instances of Area. One of them (a2) is shownwith its own links to three Wall and one Door object. Each of these walls is marked with itscurrentwidth,andeachisshownlinkedtoitsneighboringwalls.Asthisobjectdiagrams uggests, the robothas recognized this enclosed area, which has walls on three sides and adooronthefourth.

Forward and ReverseEngineering

Forwardengineering(thecreationofcodefromamodel)anobjectdiagramistheo retically possible but pragmatically of limited value. In an object-oriented system,

instancesare things that are created and destroyed by the application during run time. Therefore, youcan'texactly instantiatetheseobjects fromtheoutside.

Reverse engineering (the creation of a model from code) an object diagram is a verydifferent thing. In fact, while you are debugging your system, this is something that you or yourtools will do all the time. For example, if you are chasing down a dangling link, you'll want toliterally or mentally draw an object diagram of the affected objects to see where, at a givenmomentintime,anobject'sstateoritsrelationshiptootherobjectsisbroken.

The process of functional modelling can be visualized in the following steps −

- Identify all the inputs and outputs
- Construct data flow diagrams showing functional dependencies
- State the purpose of each function
- Identify constraints
- Specify optimization criteria

Structured Analysis vs. Object Oriented Analysis

The Structured Analysis/Structured Design (SASD) approach is the traditional approach of software development based upon the waterfall model. The phases of development of a system using SASD are −

- Feasibility Study
- Requirement Analysis and Specification
- System Design
- Implementation
- Post-implementation Review

Now, we will look at the relative advantages and disadvantages of structured analysis approach and object-oriented analysis approach.

Advantages	Disadvantages
Focuses on data rather than the procedures as in Structured Analysis.	Functionality is restricted within objects. This may pose a problem for systems which are intrinsically procedural or computational in nature.

The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	It cannot identify which objects would generate an optimal system design.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	The object-oriented models do not easily show the communications between the objects in the system.
It allows effective management of software complexity by the virtue of modularity.	All the interfaces between the objects cannot be represented in a single diagram.
It can be upgraded from small to large systems at a greater ease than in systems following structured analysis.	

Advantages/Disadvantages of Object Oriented Analysis

Advantages/Disadvantages of Structured Analysis

Advantages	Disadvantages
As it follows a top-down approach in contrast to bottom-up approach of object-oriented analysis, it can be more easily comprehended than OOA.	In traditional structured analysis models, one phase should be completed before the next phase. This poses a problem in design, particularly if errors crop up or requirements change.
It is based upon functionality. The overall purpose is identified and then functional decomposition is done for developing the software. The emphasis not only gives a better understanding of the system but also generates more complete systems.	The initial cost of constructing the system is high, since the whole system needs to be designed at once leaving very little option to add functionality later.

The specifications in it are written in simple English language, and hence can be more easily analyzed by non-technical personnel.	It does not support reusability of code. So, the time and cost of development is inherently high.

Patterns

Software development design patterns were started as best practices that were applied again and again to similar problems encountered in different contexts.

Examples of common problems solved by design pattern

1. How to instantiate an object properly!
2. How to interact between two objects!

What is a Design pattern?

Design pattern is a solution approach to a common problem, It should be an industry standard without language dependent

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't 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 situations.

What are all the benefits of the design pattern?

The truth is that you might manage to work as a programmer for many years without knowing about a single pattern. A lot of people do just that. Even in that case, though, you might be implementing some patterns without even knowing it. So why would you spend time learning them?

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

2. Reusing the design patterns helps to prevent subtle issues that can cause major problems and it also improves code readability.

3. Design pattern provides general solutions, documented in a format that doesn't specifics tied to a particular problem.

4 In addition to that patterns allows developers to communicate well-known, well-understood names for software interactions, Common design patterns can be improved over time, making them more robust than ad-hoc design.

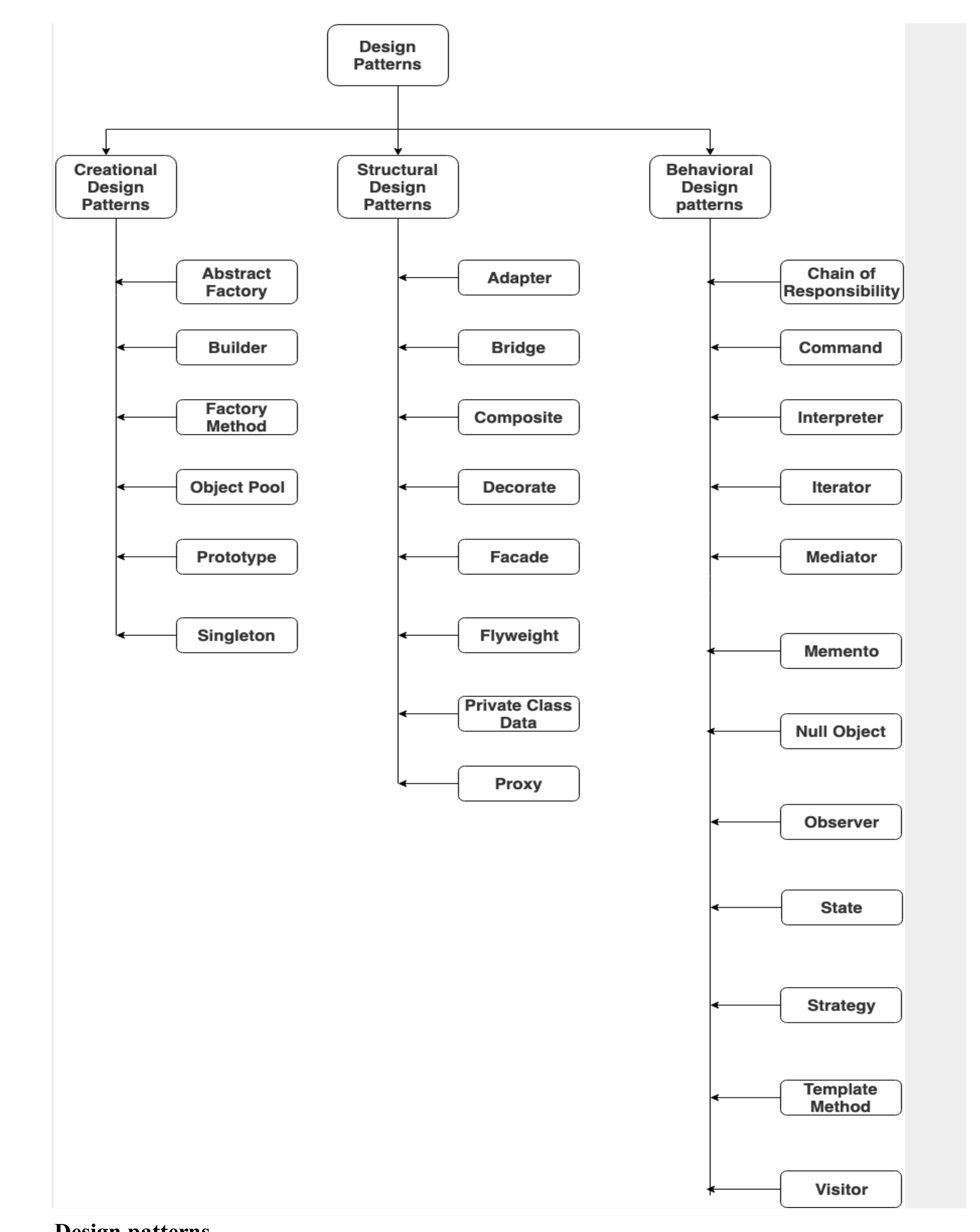
5. A standard solution to a common programming problem enables large scale reuse of software.

Let's see the classification of design patterns

Design Patterns Classification:

Design Pattern can be classified into three types

1. Creational design patterns
2. Structural design patterns
3. Behavioral design patterns



Design patterns

The above image illustrates all classification of design patterns

Creational Design Patterns:

Creational design patterns are concerned with the way of creating objects. These design patterns are used when a decision must be made at the time of the instantiation of a class (i.e. creating an object of a class).
This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Creational design patterns are mentioned in the above image.

Structural Design Patterns:

Structural design patterns are concerned with how classes and objects can be composed, to form larger structures. The structural design patterns simplify the structure by identifying relationships.

These patterns focus on, how the classes inherit from each other and how they are composed of other classes.

Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

Structural design patterns are mentioned in the above image.

Behavioral Design Patterns:

Behavioral design patterns are concerned with the interaction and responsibility of objects. In these design patterns, the interaction between the objects should be in such a way that they can easily talk to each other and still should be loosely coupled.

That means the implementation and the client should be loosely coupled to avoid hard coding and dependencies.

Behavioral design patterns are mentioned in the above image.

As we see the above contents are about introduction to design patterns and its importance in software development, Its not necessary to implement all of the design patterns in your software development, though you can leverage the appropriate design patterns to your problems in software development.

Object-Oriented Application Frameworks

Computing power and network bandwidth have increased dramatically over the past decade. However, the design and implementation of complex software remains expensive and error-prone. Much of the cost and effort stems from the continuous re-discovery and re-invention of core concepts and components across the software industry. In particular, the growing heterogeneity of hardware architectures and diversity of operating system and communication platforms makes it hard to build correct, portable, efficient, and inexpensive applications from scratch.

Object-oriented (OO) application frameworks are a promising technology for reifying proven software designs and implementations in order to reduce the cost and improve the quality of software. A framework is a reusable, ``semi-complete" application that can be specialized to produce custom applications [Johnson:88]. In contrast to earlier OO reuse techniques based on class libraries, frameworks are targeted for particular business units (such as data processing or cellular communications) and application domains (such as user interfaces or real-time avionics). Frameworks like MacApp, ET++, Interviews, ACE, Microsoft's MFC and DCOM, JavaSoft's RMI, and implementations of OMG's CORBA play an increasingly important role in contemporary software development.

The primary benefits of OO application frameworks stem from the *modularity*, *reusability*, *extensibility*, and *inversion of control* they provide to developers, as described below:

- *Modularity* -- Frameworks enhance modularity by encapsulating volatile implementation details behind stable interfaces. Framework modularity helps improve software quality by localizing the impact of design and implementation changes. This localization reduces the effort required to understand and maintain existing software.
- *Reusability* -- The stable interfaces provided by frameworks enhance reusability by defining generic components that can be reapplied to create new applications. Framework reusability leverages the domain knowledge and prior effort of experienced developers in order to avoid re-creating and re-validating common solutions to recurring application requirements and software design challenges. Reuse of framework components can yield substantial improvements in programmer productivity, as well as enhance the quality, performance, reliability and interoperability of software.
- *Extensibility* -- A framework enhances extensibility by providing explicit hook methods [Pree:94] that allow applications to extend its stable interfaces. Hook methods systematically decouple the stable interfaces and behaviors of an application domain from the variations required by instantiations of an application in a particular context. Framework extensibility is essential to ensure timely customization of new application services and features.
- *Inversion of control* -- The run-time architecture of a framework is characterized by an ``inversion of control." This architecture enables canonical application processing steps to be customized by event handler objects that are invoked via the framework's reactive dispatching mechanism. When events occur, the framework's dispatcher reacts by invoking hook methods on pre-registered handler objects, which perform application-

specific processing on the events. Inversion of control allows the framework (rather than each application) to determine which set of application-specific methods to invoke in response to external events (such as window messages arriving from end-users or packets arriving on communication ports).

Developers in certain domains have successfully applied OO application frameworks for many years. Early object-oriented frameworks (such as MacApp and Interviews) originated in the domain of graphical user interfaces (GUIs). The Microsoft Foundation Classes (MFC) is a contemporary GUI framework that has become the *de facto* industry standard for creating graphical applications on PC platforms. Although MFC has limitations (such as lack of portability to non-PC platforms), its wide-spread adoption demonstrates the productivity benefits of reusing common frameworks to develop graphical business applications.

Application developers in more complex domains (such as telecommunications, distributed medical imaging, and real-time avionics) have traditionally lacked standard ``off-the-shelf'' frameworks. As a result, developersin these domains largely build, validate, and maintain software systems from scratch. In an era of deregulation and stiff global competition, however, it has become prohibitively costly and time consuming to develop applications entirely in-house from the ground up.

Fortunately, the next generation of OO application frameworks are targeting complex business and application domains. At the heart of this effort are *Object Request Broker* (ORB) frameworks, which facilitate communication between local and remote objects. ORB frameworks eliminate many tedious, error-prone, and non-portable aspects of creating and managing distributed applications and reusable service components. This enables programmers to develop and deploy complex applications rapidly and robustly, rather than wrestling endlessly with low-level infrastructure concerns. Widely used ORB frameworks include CORBA, DCOM, and Java RMI.

Although the benefits and design principles underlying frameworks are largely independent of domain to which they are applied, we've found it useful to classify frameworks by their *scope*, as follows:

- *System infrastructure frameworks* -- These frameworks simplify the development of portable and efficient system infrastructure such as operating system [Campbell-Islam:93] and communication frameworks [Schmidt:97], and frameworks for user interfaces and language processing tools. System infrastructure frameworks are primarily used internally within a software organization and are not sold to customers directly.
- *Middleware integration frameworks* -- These frameworks are commonly used to integrate distributed applications and components. Middleware integration frameworks are designed to enhance the ability of software developers to modularize, reuse, and extend their software infrastructure to work seamlessly in a distributed environment. There is a thriving market for Middleware integration frameworks, which are rapidly becoming commodities. Common examples include ORB frameworks, message-oriented middleware, and transactional databases.

- *Enterprise application frameworks* -- These frameworks address broad application domains (such as telecommunications, avionics, manufacturing, and financial engineering [Birrer:93]) and are the cornerstone of enterprise business activities [Fayad-Hamu:97]. Relative to System infrastructure and Middleware integration frameworks, Enterprise frameworks are expensive to develop and/or purchase. However, Enterprise frameworks can provide a substantial return on investment since they support the development of end-user applications and products directly. In contrast, System infrastructure and Middleware integration frameworks focus largely on internal software development concerns. Although these frameworks are essential to rapidly create high quality sofware, they typically don't generate substantial revenue for large enterprises. As a result, it's often more cost effective to buy System infrastructure and Middleware integration frameworks rather than build them in-house [Fayad-Hmau:97].

Regardless of their scope, frameworks can also be classified by the techniques used to extend them, which range along a continuum from *whitebox frameworks* to *blackbox frameworks*. Whitebox frameworks rely heavily on OO language features like inheritance and dynamic binding to achieve extensibilty. Existing functionality is reused and extended by (1) inheriting from framework base classes and (2) overriding pre-defined hook methods using patterns like Template Method [Gamma:95]. Blackbox frameworks support extensibility by defining interfaces for components that can be plugged into the framework via object composition. Existing functionality is reused by (1) defining components that conform to a particular interface and (2) integrating these components into the framework using patterns like Strategy [Gamma:95] and Functor.

Whitebox frameworks require application developers to have intimate knowledge of the frameworks' internal structure. Although whitebox frameworks are widely used, they tend to produce systems that are tightly coupled to the specific details of the framework's inheritance hierarchies. In contrast, blackbox frameworks are structured using object composition and delegation more than inheritance. As a result, blackbox frameworks are generally easier to use and extend than whitebox frameworks. However, blackbox frameworks are more difficult to develop since they require framework developers to define interfaces and hooks that anticipate a wider range of potential use-cases [Johnson:95].

Frameworks are closely related to other approaches to reuse, including:

- *Patterns* -- Patterns represent recurring solutions to software development problems within a particular context. Patterns and frameworks both facilitate reuse by capturing successful software development strategies. The primary difference is that frameworks focus on reuse of concrete designs, algorithms, and implementations in a particular programming language. In contrast, patterns focus on reuse of abstract designs and software micro-architectures.

  Frameworks can be viewed as a concrete reification of families of design patterns that are targeted for a particular application-domain. Likewise, design patterns can be viewed as more abstract micro-architectural elements of frameworks that document and motivate the semantics of frameworks in an effective way. When patterns are used to structure and

document frameworks, nearly every class in the framework plays a well-defined role and collaborates effectively with other classes in the framework.

- *Class libraries* -- Frameworks extend the benefits of OO class libraries in the following ways:
    - *Frameworks define ``semi-complete'' applications that embody domain-specific object structures and functionality* -- Components in a framework work together to provide a generic architectural skeleton for a family of related applications. Complete applications can be composed by inheriting from and/or instantiating framework components. In contrast, class libraries are less domain-specific and provide a smaller scope of reuse. For instance, class library components like classes for Strings, complex numbers, arrays, and bitsets are relatively low-level and ubiquitous across many application domains.
    - *Frameworks are active and exhibit ``inversion of control'' at run-time* -- Class libraries are typically *passive*, *i.e.*, they perform their processing by borrowing threads of control from self-directed application objects. In contrast, frameworks are *active*, *i.e.*, they control the flow of control within an application via event dispatching patterns like Reactor and Observer. The ``inversion of control'' in the run-time architecture of a framework is often referred to as *The Hollywood Principle*, *i.e.*, ``Don't call us, we'll call you.''

  In practice, frameworks and class libraries are complementary technologies. For instance, frameworks tyically utilize class libraries like the C++ Standard Template Library (STL) internally to simplify the development of the framework. Likewise, application-specific code invoked by framework event handlers can utilize class libraries to perform basic tasks such as string processing, file management, and numerical analysis.

- *Components* -- Components are self-contained instances of abstract data types (ADTs) that can be plugged together to form complete applications. Common examples of components include VBX controls and CORBA Object Services. In terms of OO design, a component is a blackbox that defines a cohesive set of operations, which can be reused based solely upon knowledge of the syntax and semantics of its interface. Compared with frameworks, components are less tightly coupled and can support binary-level reuse. For example, applications can reuse components without having to subclass from existing base classes.

  The relationship between frameworks and components is highly synergistic, with neither subordinate to the other. Frameworks can be used to develop components, whereby the component interface provides a *Facade* for the internal class structure of the framework. Likewise, components can be used as pluggable strategies in blackbox frameworks. In general, frameworks are often used to simplify the development of infrastructure and middleware software, whereas components are often used to simplify the development of end-user application software. Naturally, components are also effective for developing infrastructure and middleware, as well.

When used in conjunction with patterns, class libraries, and components, OO application frameworks can significantly increase software quality and reduce development effort. However,

a number of challenges must be addressed in order to employ frameworks effectively. Companies attempting to build or use large-scale reusable framework often fail unless they recognize and resolve challenges such as *development effort*, *learning curve*, *integratability*, *maintainability*, *validation and defect removal*, *efficiency*, and *lack of standards*, which are outlined below:

- *Development effort* -- While developing complex software is hard enough, developing high quality, extensible, and reusable frameworks for complex application domains is even harder. The skills required to produce frameworks successfully often remain locked in the heads of expert developers. One of the goals of this theme issue is to demystify the software process and design principles associated with developing and using frameworks.
- *Learning curve* -- Learning to use an OO application framework effectively requires considerable investment of effort. For instance, it often takes 6-12 months become highly productive with a GUI framework like MFC or MacApp, depending on the experience of developers. Typically, hands-on mentoring and training courses are required to teach application developers how to use the framework effectively. Unless the effort required to learn the framework can be amortized over many projects, this investment may not be cost effective. Moreover, the suitability of a framework for a particular application may not be apparent until the learning curve has flattened.
- *Integratability* -- Application development will be increasingly based on the integration of multiple frameworks (*e.g.* GUIs, communication systems, databases, etc.) together with class libraries, legacy systems, and existing components. However, many earlier generation frameworks were designed for internal extension rather than for integration with other frameworks developed externally. Integration problems arise at several levels of abstraction, ranging from documentation issues [Fayad-Hamu 97], to the concurrency/distribution architecture, to the event dispatching model. For instance, while inversion of control is an essential feature of a framework, integrating frameworks whose event loops are not designed to interoperate with other frameworks is hard.
- *Maintainability* -- Application requirements change frequently. Therefore, the requirements of frameworks often change, as well. As frameworks invariably evolve, the applications that use them must evolve with them.

  Framework maintenance activities include modification and adaptation of the framework. Both modification and adaptation may occur on the *functional level* (*i.e.*, certain framework functionality does not fully meet developers' requirements), as well as on the *non-functional level* (which includes more qualitative aspects such as portability or reusability).

  Framework maintenance may take different forms, such as adding functionality, removing functionality, and generalization. A deep understanding of the framework components and their interrelationships is essential to perform this task successfully. In some cases, the application developers and/or the end-users must rely entirely on framework developers to maintain the framework.

- *Validation and defect removal* -- Although a well-designed, modular framework can localize the impact of software defects, validating and debugging applications built using frameworks can be tricky for the following reasons:
    - *Generic components are harder to validate in the abstract* -- A well-designed framework component typically abstracts away from application-specific details, which are provided via subclassing, object composition, or template parameterization. While this improves the flexibility and extensibility of the framework, it greatly complicates module testing since the components cannot be validated in isolation from their specific instantiations.

      Moreover, it is usually hard to distinguish bugs in the framework from bugs in application code. As with any software development, bugs are introduced into a framework from many possible sources, such as failure to understand the requirements, overly coupled design, or an incorrect implementation. When customizing the components in framework to a particular application, the number of possible error sources will increase.

    - *Inversion of control and lack of explicit control flow* -- Applications written with frameworks can be hard to debug since the framework's ``inverted'' flow of control oscillates between the application-independent framework infrastructure and the application-specific method callbacks. This increases the difficulty of ``single-stepping'' through the run-time behavior of a framework within a debugger since the control flow of the application is driven implicitly by callbacks and developers may not understand or have access to the framework code. This is similar to the problems encountered trying to debug a compiler lexical analyser and parser written with LEX and YACC. In these applications, debugging is straightforward when the thread of control is in the user-defined action routines. Once the thread of control returns to the generated DFA skeleton, however, it is hard to trace the program's logic.
- *Efficiency* -- Frameworks enhance extensibility by employing additional levels of indirection. For instance, dynamic binding is commonly used to allow developers to subclass and customize existing interfaces. However, the resulting generality and flexibility often reduce efficiency. For instance, in languages like C++ and Java, the use of dynamic binding makes it impractical to support Concrete Data Types (CDTs), which are often required for time-critical software. The lack of CDTs yields (1) an increase in storage layout (*e.g.*, due to embedded pointers to virtual tables), (2) performance degradation (*e.g.* due to the additional overhead of invoking a dynamically bound method and the inability to inline small methods), and (3) a lack of flexibility (*e.g.*, due to the inability to place objects in shared memory).
- *Lack of standards* -- Currently, there are no widely accepted standards for designing, implementing, documenting, and adapting frameworks. Moreover, emerging industry standard frameworks (such as CORBA, DCOM, and Java RMI) currently lack the semantics, features, and interoperability to be truly effective across multiple application domains. Often, vendors use industry standards to sell proprietary software under the guise of open systems. Therefore, it's essential for companies and developers to work with standards organizations and middleware vendors to ensure the emerging

specifications support true interoperability and define features that meet their software needs.

Over the next several years, we expect the following framework-related topics will receive considerable attention by researchers and developers:

- *Reducing framework development effort* -- Traditionally, reusable frameworks have been developed by generalizing from existing systems and applications. Unfortunately, this incremental process of organic development is often slow and unpredictable since core framework design principles and patterns must be discovered ``bottom-up.'' However, since many good framework exemplars now exist, we expect that the next generation of developers will leverage this collective knowledge to conceive, design, and implement higher quality frameworks more rapidly.
- *Greater focus on domain-specific enterprise frameworks* -- Existing frameworks have focused largely on system infrastructure and middleware integration domains (such as user interfaces [Gamma:95,Pree:94] and OS/communication systems [Schmidt:97,Johnson:95,Cambell-Islam:93]). In contrast, there are relatively few widely documented exemplars of enterprise frameworks for key business domains such as manufacturing, banking, insurance, and medical systems. As more experience is gained developing frameworks for these business domains, however, we expect that the collective knowledge of frameworks will be expanded to cover an increasing wide range of domain-specific topics and an increasing number of Enterprise application frameworks will be produced. As a result, benefits of frameworks will become more immediate to application programmers, as well as to infrastructure developers.
- *Blackbox frameworks* -- Many framework experts [Johnson:88] favor black-box frameworks over white-box frameworks since black-box frameworks emphasize dynamic object relationships (via patterns like Bridge and Strategy Gamma:95) rather than static class relationships. Thus, it is easier to extend and reconfigure black-box frameworks dynamically. As developers become more familiar with techniques and patterns for factoring out common interfaces and components, we expect that an increasing percentage of black-box frameworks will be produced.
- *Framework documentation* -- Accurate and comprehensible documentation is crucial to the success of large-scale frameworks. However, documenting frameworks is a costly activity and contemporary tools often focus on low-level method-oriented documentation, which fails to capture the strategic roles and collaborations among framework components. We expect that the advent of tools for reverse-engineering the structure of classes and objects in complex frameworks will help to improve the accuracy and utility of framework documentation. Likewise, we expect to see an increase in the current trend [Johnson:95,Schmidt:97] of using design patterns to provide higher-level descriptions of frameworks.
- *Processes for managing framework development* -- Frameworks are inherently abstract since they generalize from a solution to a particular application challenge to provide a family of solutions. This level of abstraction makes it difficult to engineer their quality and manage their production. Therefore, it is essential to capture and articulate development processes that can ensure the successful development and use of frameworks. We believe that extensive prototyping and phased introduction of

framework technology into organizations is crucial to reducing risk and helping to ensure successful adoption.

- *Framework economics* -- The economics of developing framework includes activities such as the following:
  - *Determining effective framework cost metrics* -- which measure the savings of reusing framework components vs. building applications from scratch;
  - *Cost estimation* -- which is the activity of accurately forecasting the cost of buying, building, or adapting a particular framework;
  - *Investment analysis and justification* -- which determines the benefits of applying frameworks in terms of return on investment;

We expect that the focus on framework economics will help to bridge the gap among the technical, managerial, and financial aspects of making, buying, or adapting frameworks [Hamu-Fayad:97].

The articles in this theme issue describe how OO application frameworks provide a powerful vehicle for reuse, as well as a way to capture the essence of successful patterns, architectures, components, policies, services, and programming mechanisms. The feature articles lead off with ``Framework Development for Large Systems" by Dirk Baumer, Guido Gryczan, Rolf Knoll, Carola Lilienthal, Dirk Riehle, and Heinz Zullighoven. These authors draw on their experience developing large-scale industrial banking projects to present concepts and techniques for domain partitioning, framework layering and framework construction. The second feature article on "Evolving Custom-Made Applications into Domain-Specific Frameworks" by WimCodenie, Koen De Hondt, Patrick Steyaert, Arlette Vercammen discusses solutions to common framework development challenges such as avoiding the proliferation of versions, estimating effort, and alleviating the tendency towards ``architectural drift."

The Theme section also contains several short articles, starting with "Frameworks = (Patterns + Components)/2" by Ralph Johnson, which compares and contrasts frameworks with other object-oriented reuse techniques such as patterns and components. The second short article is "An Adaptive Framework for Developing Multimedia Software Components" by Edward Posnak, Greg Lavender, and Harrick Vin describes a framework that simplifies the development of dynamically adaptive multimedia software components by promoting the reuse of code, design patterns, and domain expertise. The next short article is "Frameworks Design by Systematic Generalization with Hot Spots and Patterns" by Hans Albrecht Schmid, which presents a systematic method for designing frameworks based on identifying "hot spots," which capture key sources of variation in an application domain. Serge Demeyer, Theo Meijler, Oscar Nierstrasz, and Patrick Steyaert also focus on hot spots in their article on "Design Guidelines for Tailorable Frameworks," which presents design guidelines to develop frameworks for open systems.

Several case studies are also covered in the theme issues, including "The Framework Life Span: a Case Study for Flexible Manufacturing Systems" by A. Aarsten, DavideBrugali, and G. Menga, which highlights the relationships between application frameworks, patterns, and pattern languages in the domain of manufacturing systems. Likewise, the SEMATECH CIM Framework, by David Doscher and Robert Hodges describes the structure of a framework for computer integrated manufacturing of semiconductors. In addition, Adele Goldberg, Steve Abell,

and David Leibs describe LearningWorks, which is a framework for exploring ideas about computing and software system construction.

Finally, the theme section contains three sidebars: "Achieve Bottom-Line Improvements with Enterprise Frameworks" by Hamu and Fayad, "Framework Integration problems, Causes, and Solutions by M. Mattsson et al., and "Lessons Learned Building Reusable OO Frameworks for Distributed Software" by Schmidt and Fayad.

The articles in this theme issue reinforce our believe that object-oriented application frameworks will be at the core of leading-edge software technology in the twenty-first century. As software systems become increasingly complex, object-oriented application frameworks are becoming increasingly important for industry and academia. The extensive focus on application frameworks in the object-oriented community offers software developers an important vehicle for reuse and a means to capture the essence of successful patterns, architectures, components, and programming mechanisms.

The good news is that framework are becoming mainstream and developers at all levels are increasingly adopting and succeeding with framework technologies. However, OO application frameworks are ultimately only as good as the people who build and use them. Creating robust, efficient, and reusable application frameworks requires development teams with a wide range of skills. We need expert analysts and designers who have mastered patterns, software architectures, and protocols in order to alleviate the inherent and accidental complexities of complex software. Likewise, we need expert middleware developers who can implement these patterns, architectures, and protocols within reusable frameworks. In addition, we need application programmers who have the motivation, skills, and training to learn how to use these frameworks effectively. We encourage you to get involved with others working on frameworks by attending conferences, participating in online mailing lists and newsgroups, and contributing your insights and experience.

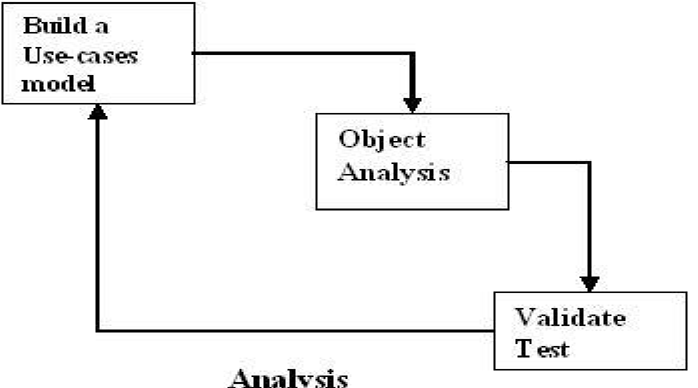
Unified Approach

UML as the name suggests has come this far through the process of unification and thus combines ideas of leading thinkers and gives access to their methodology of modeling in consistency with that of the others. Now we are planning to get down a level further and to represent these modeling methodologies on the same canvas.
To bring these entities under the same frame of reference we need first to look for a commonality that is not far stretched or bolted out of blue sky. Such point of joining between the static logical view and the dynamic interaction view is in terms of objects in the interaction view as also the object view that materialize as instances of classifiers (classes) in the logical view (class diagram).

The unified approach to software development revolves around the following processes and concepts. The processes are,

1. Object oriented analysis 3. Object oriented design
2. Testing 4. Developing and Prototyping

Object oriented analysis: Analysis is the first stage of the development process. It is also the first step in producing high-performance software. Analysis is central to the performance tuning process.The analysis part having these below steps –



OOA process consists of the following steps:

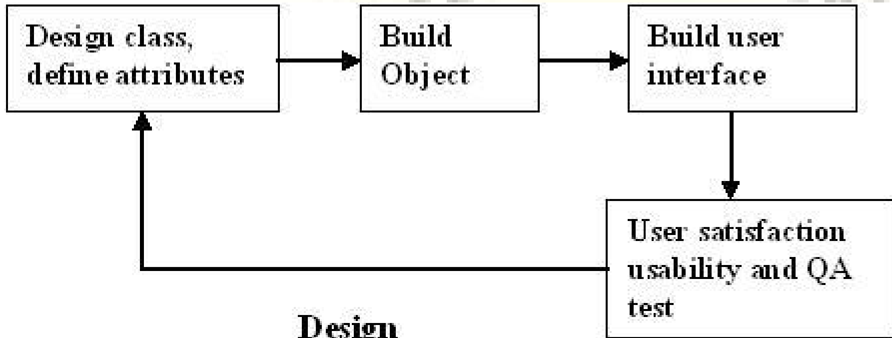
- Identify the actors
- Develop a simple business process model using UML Activity diagram
- Develop the usecase
- Develop interaction diagrams
- Identify classes

Object-Oriented Design

Object – oriented design plays a major role in the performance process. While there are many factors that contribute to a good design, and all of them are important, there is one concept that is especially critical to creating high-performance systems. This concept is called encapsulation.

Making encapsulation part of your design from the start enables you to:

- Quickly evaluate different algorithms and data structures to see which is most efficient.
- Easily evolve your design to accommodate new or changed requirements.



OOD process consist of

- Designing classes, their attributes, methods, associations, structures and protocols, apply design axioms
- Design the Access Layer
- Design the prototype user interface
- User satisfaction and usability tests based on the usage/use cases
- Iterate and refine the design

Iterative development and continuous testing:

Iterate and Reiterate until you are satisfied with the system. Testing uncovers the design weaknesses or provides additional information. Repeat the entire process, taking what you have learned and reworking you design more onto re-prototyping and retesting.

UML

Unified Approach encourages integration testing from the day 1 of the project usage scenarios can become test scenarios. Therefore use cases will drive the usability testing.

Modeling based on the Unified Modeling Language:

UML is becoming the universal language for modeling systems; it is intended to be used to express models of many different kinds and purposes. UML has become the standard notation for object oriented modeling systems.

The Unified Approach uses the UML to describe and model the analysis and design phases of system development.

Unified Approach Proposed Repository:

Best practice sharing eliminates duplication of problem solving. It must be applied to application development, if quality and productivity are to be added. The idea promoted here is to create a repository that allows the maximum reuse of previous experience and previously defined objects, patterns, frameworks etc in an easily accessible manner. It should be accessible to many people.

Advantages:

- If your organization has done projects in the past, objects in the repositories from those projects might be useful.
- Creating additional applications will require no more than assembling components from the library.
- Applying lessons learned from the part will increase the quality of the product and reduce the cost and development time.

Layered Approach to Software Development:

Three layered approaches,

- Business Layer:

The business layer contains all the objects that represent the business.
The responsibilities of business layer are very straightforward.
"Model the objects of the business and how they interact to accomplish the business processes."
A business model captures the static and dynamic relationships among a collection of business objects.
These objects should not be responsible for Displaying details and Data Access details.

- View Layer / User Interface Layer:

It consists of objects with which the user interacts as well as the objects needed to manage or control the interface.
There objects are identified during the object oriented design phase.
This layer is responsible for 2 major aspects. They are,
Responding to user interaction and displaying business objects

Access Layer:
It contains objects that know how to communicate with the place where the data actually reside, whether it be a relational database, mainframe, Internet or file.
It has two major responsibilities, Translate request and Translate result.

TounderstandtheUML,we needtoform aconceptualmodelofthelanguage,andthisrequireslearning threemajor elements:

* BasicBuildingblocksoftheUML
* RulesoftheUML
* Common mechanisms intheUML.

**<u>BasicBuildingBlocksoftheUML:</u>

ThevocabularyoftheUMLencompassesthreekindsofbuildingblocks:

1.1 ThingsintheUML
1.2 RelationshipsintheUML
1.3 Diagramsinthe UML

1 <u>ThingsintheUML:</u>-

Thingsaretheabstractionsthatarefirst-classcitizensinamodel.Therearefourkindsofthingsinthe UML:

1.1 Structuralthings
1.2 Behavioralthings
1.3 Groupingthings
1.4 Annotationalthings
1.5

1 Structuralthings:

Structuralthingsarethe nounsofUMLmodels.Thesearethemostlystaticpartsofamodel.Thereareseven kindsofstructural things.They area.Class

1.b Interface
1.c Collaboration
1.d Usecase
1.e ActivityClass
1.f Component
1.g Node

a Class:

A**class** is a description of a set of objects that share the same attributes, operations,relationships, and semantics. A class implements one or more interfaces. Graphically, a class isrenderedasarectangle,usuallyincludingitsname,attributes,andoperations.

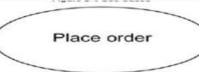
b Interface:

An**i**nterfaceisacolle aclassorcomponent.

Graphically,aninter ame.

C)Collaboration:

Collaboration defines an y of roles and other elements thatwork together to provide so at's bigger than the sum of all theelements.Graphically,acollabo ithdashedlines,usuallyincludi ngonlyitsname.



d Usecase:

Ause case is a description of set of sequence of actions that a system performs thatyields an observable result of value to a particular actor. A use case is used to structure thebehavioral things in a model. A use case is realized by a collaboration. Graphically, a use case isrenderedasanellipsewithsolidlines, usuallyincludingonlyitsname.

e Activeclass

An active class is a class whose objects own one or more processes or threads andtherefore can initiate control activity.Graphically, an active class is rendered just like a class, butwithheavylines,usuallyincluding itsname, attributes,andoperations.

f Component:

A Componentis a physicalandreplaceable partof a system that conformsto andprovidestherealizationof asetofinterfaces.Graphically,acomponentisrendered asarectanglewithtabs, usuallyincluding onlyitsname.

g Node:

A **node** is a physical element that exists at run time and represents a computationalresource,generallyhavingatleastsomememoryand,often,processingcapability.Graphically,anodeisrenderedas acube, usuallyincludingonlyitsname.

Behavioralthings:

Behavioral things are the dynamic parts of UML models. These are the verbs of a model,representing behavior over time and space. There are two primary kinds of behavioral things.Theyare

1.a Interaction
1.b Statemachine

a Interaction

Interaction is a abehavior that comprises a set of messages exchanged among a set ofobjects within a particular context to accomplish specific purpose. An interaction involves anumber of other elements, including messages, action sequences (the behavior invoked by amessage), and links (the connection between objects). Graphically, a message is rendered as a**directedline**,almostalwaysincludingthe nameofitsoperation.

b Statemachine

A State machine a number of other elements, including states, transitions (the flowfrom state to state), events (things that trigger a transition), and activities (the response to atransition).Graphically, a state is rendered as a **rounded rectangle**, usually including its nameanditssubstates.

2 Groupingthings:

Grouping things are the organizational parts of UML models. These are the boxes intowhich a model can be decomposed. There is one primary kind of groupingthing, namely,packages.

package:

Apackageisageneral-purposemechanismfororganizingelementsintogroups.Structuralthings,behavioralthings,andevenothergroupingthingsmaybeplacedinapackage.Graphically, a package is rendered as a **tabbed folder**, usually including only its nameand,sometimes, itscontents.

Annotationalthings:

Annotational things are the explanatory parts of UML models. These are the commentsyou may apply to describe, illuminate, and remark about any element in a model. There is oneprimarykindofannotationalthing,calleda note.

Note:

Anoteissimplyasymbolforrenderingconstraintsandcommentsattachedtoanelement or a collection of elements. Graphically, a note is rendered as a rectangle with a **dog-earedcorner**,togetherwith atextualorgraphicalcomment.

return copy
of self

1 Relationships intheUML:

Relationshipsareusedtoconnectthings.Therearefourkindsofrelationshipsinthe UML:

1.1 Dependency
1.2 Association
1.3 Generalization
1.4 Realization

1 Dependency:

A dependency is a semantic relationship between two things in which a change to onething (the independent thing) may affect the semantics of the other thing (the dependentthing).Graphically,adependencyisrenderedasadashedline,possiblydirected,andoccasionallyincludingalabel.

2 Association:

An **association** is a structural relationship that describes a set of links, a link being aconnectionamongobjects.Aggregationisaspecialkindofassociation,representingastructural relationship between a whole and its parts. Graphically, an association is rendered asasolidline,possiblydirected,occasionallyincludingalabel,andoftencontainingotheradornments,suchas multiplicityand rolenames.

MAR GREGORIOS COLLEGE, CHENNAI, INDIA
LET YOUR LIGHT SHINE

3 **Generalization:**

A Generalization is a specialization/generalization relationship in which objects of thespecialized element (the child) are substitutable for objects of the generalized element (theparent).Inthisway,thechildsharesthestructureandthebehavioroftheparent.Graphically,ageneralizationrelationshipis renderedasasolidlinewithahollowarrowheadpointingtotheparent.

4 **Realization:**

A**Realization**isasemanticrelationshipbetweenclassifiers ,whereinoneclas classifier guarantees , a realizationrelatic onandade pendencyrelatio

3 **Diagrams in theUML:-**

Adiagramisthegraphicalpresentationofasetofelements,mostoftenrendered as a connected graph of vertices (things) and arcs (relationships). The UMLincludesninesuch diagrams:

1. Classdiagram
2.Objectdiagram
3.Use casediagram
4.Sequencediagram
5.Collaborationdiagram
6.Statechartdiagram
7.Activitydiagram
8.Componentdiagram
9.Deploymentdiagram

1 **Class diagram:**

Aclassdiagramshowsasetofclasses,interfaces,andcollaborationsandtheirrelationships.Thesediagramsarethemostcommondiagramfoundinmodelingobject-orientedsystems.Class diagrams addressthestatic designviewofasystem.

2 **Objectdiagram:**

Anobjectdiagramshowsasetofobjectsandtheirrelationships.Object diagramsrepresent static snapshotsof instances ofthethingsfound in

class diagrams.These diagrams addressthestatic designview orstatic process view of asystem .

3 usecasediagram:

A use case diagram shows a set of use cases and actors (a special kind of class) and theirrelationships. Use case diagrams address the static use case view of a system. These diagrams areespeciallyimportantinorganizingandmodeling thebehaviorsofasystem.

4 Sequence diagram:

A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages;Interaction diagramsaddressthedynamicview ofasystem.Sequencediagramsand collaborationdiagramsareisomorphic,meaningthatyoucantakeoneandtransformit intothe other.

5 collaborationdiagram:

A **collaboration diagram** is an interaction diagram that emphasizes the structural organization ofthe objects that send and receive messages. Interaction diagrams address the dynamic view of a system.Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one andtransformitintotheother.

6 statechartdiagram:

Astatechartdiagramshowsa state machine, consistingof states,transitions,events,andactivities.Statechartdiagramsaddressthedynamicview ofasystem.

7 Activitydiagram

An activity diagram is a special kind of a state chart diagram that shows the flow fromactivitytoactivity withinasystem.Activitydiagramsaddressthedynamicview ofasystem.

8 componentdiagram:

Acomponentdiagramshowstheorganizationsanddependenciesamongasetofcomponents.Componentdiagramsaddressthestaticimplementationviewofasystem.

9 deployment diagram:A deployment diagram shows the configuration of run-time processingnodesandthecomponentsthatliveonthem.Deploymentdiagramsaddressthestaticdeploymentview ofarchitecture.

Rulesof the UML:-

Like any language, the UML has a number of rules that specify what a well-formed model should look like. A well-formed model is one that is semantically self-consistentandinharmonywithallitsrelatedmodels.TheUMLhassemanticrulesfor

- **Names**:Whatyoucancall things,relationships,anddiagrams
- **Scope** :Thecontextthatgivesspecificmeaningtoaname
- **Visibility**:Howthose names canbeseenandusedbyothers
- **Integrity**: Howthingsproperlyandconsistentlyrelatetooneanother
- **Execution**:Whatitmeansto runorsimulateadynamicmodel

Models built during the development of a software-intensive system tend to evolve and may beviewed by many stakeholders in different ways and at different times. For this reason, it is common forthedevelopmentteamtonot only buildmodels thatare well-formed,butalsoto buildmodels thatare

- ElidedCertainelementsarehiddentosimplifytheview
- IncompleteCertainelementsmaybemissing
- Inconsistenttheintegrity of themodelisnotguaranteed.

CommonMechanismsintheUML:

UMLismadesimplerbythepresenceoffourcommonmechanisms thatapplyconsistentlythroughout thelanguage.They are:

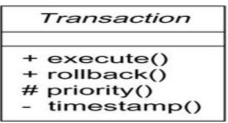
1 Specifications
2 Adornments
3 Commondivisions
4 Extensibilitymechanisms

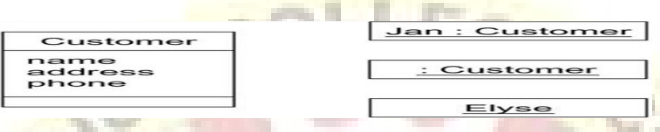
1 Specifications

TheUMLismorethanjustagraphicallanguage.Rather,behindeverypartofitsgraphical notation there is a specification that provides a textual statement of the syntax andsemanticsofthatbuildingblock.Forexample,behindaclassiconisaspecificationthatprovides the full set of attributes, operations (including their full signatures), and behaviors thattheclassembodies;

2 Adornments

Most elements in the UML have a unique and direct graphical notation that provides avisual representation of the most important aspects of the element. A class's specification mayincludeotherdetails,suchaswhetheritisabstractorthevisibilityofitsattributesand operations. Many of these details can be rendered as graphical or textual adornments to theclass'sbasicrectangular notation.





For example, Figure shows a class, adorned to indicate that it is an abstract class withtwo public, one protected, and one private operation.Every element in the UML's notationstarts with a basic symbol, to which can be added a variety of adornments specific to thatsymbol.

CommonDivisions

Inmodelingobject-orientedsystems,theworldoftengetsdividedinatleastacoupleofways.

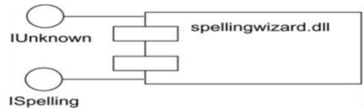
classandobject:

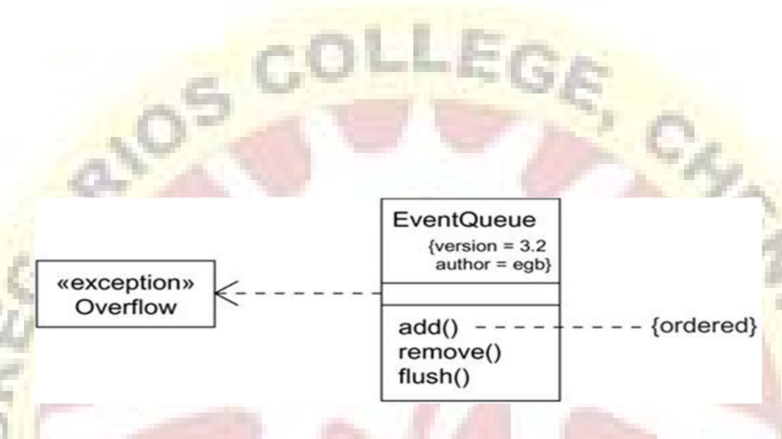
A class is an abstraction; an object is one concrete manifestation of that abstraction.

In this figure, there is one class, named Customer, together with three objects: Jan(which ismarked explicitly as being a Customer object), :Customer (an anonymous Customerobject),and Elyse (which in its specification ismarked as being a kind of Customer object,although it‘snot shown explicitly here).Graphically, the UML distinguishes an object by using thesamesymbolas itsclassandthensimplyunderlyingtheobject's name.

Interfaceandimplementation.:

Aninterfacedeclaresacontract,andanimplementationrepresentsoneconcretereal izationof thatcontract, responsible forfaithfullycarryingouttheinterface'scompletesemantics. In the UML, you can model both interfaces and their implementations, as shown inFigure





In this figure, there is one component named spellingwizard.dll that implements twointerfaces,IUnknownandIspelling.

4.<u>ExtensibilityMechanisms</u>

The UML provides a standard language for writing software blueprints, but it is notpossible for one closed language to ever be sufficient to express all possible nuances of allmodelsacrossalldomainsacrossalltime.TheUML'sextensibilitymechanisms include

1. Stereotypes
2. Taggedvalues
3. Constraints

A *stereotype extends the vocabulary of the UML, allowing you to create new kinds ofbuilding* blocks that are derived from existing ones but that are specific to your problem.Forexample, if you are workingin a programminglanguage, such asJavaor C++, you will oftenwanttomodelexceptions.Intheselanguages,exceptionsarejustclasses,Youcan makeexceptions first class citizens in your models, meaning that they are treated like basic buildingblocks,bymarkingthemwithanappropriatestereotype,asfortheclassOverflowinFigure.

Taggedvalues

A *tagged value extends the properties of a UML building block, allowing you to createnew* information in that element's specification. For example, if you want to specify the versionandauthorofcertaincriticalabstractions.Versionandauthorarenotprimitive UMLconcepts.

- o Forexample,theclassEvent Queueisextendedbymarkingitsversionandauthorexplicitl y.

3.2 Constraints

A *constraint extends the semantics of a UML building block, allowing you to add newrules or* modify existing ones. For example, you mght want to constrain the EventQueue class sothat all additions are done in order. As Figure shows, you can add a constraint that explicitlymarksthese fortheoperationadd.

UNIFIED PROCESS (UP)

The Unified Process has emerged as a popular iterative software development process for building object oriented systems. The Unified Process (UP) combines commonly accepted best practices, such as an iterative lifecycle and risk-driven development, into a cohesive and welldocumented description. The best-known and extensively documented refinement of the Unified Process is the Rational Unified Process(RUP). Reasons to use UP • UP is an iterative process • UP practices provide an example structure to talk about how to do, and how to learn OOA/D. Best Practices and Key Concepts in UP • Tackle high-risk and high-value issues in early iterations • Engage users continuously for evaluation, feedback, and requirements • Build a cohesive, core architecture in early iterations • Apply use cases • Provides visual modeling using UML • Practice change request and configuration management. UP PHASES There are 4 phases in Unified Process, 1. Inception 2. Elaboration 3. Construction 4. Transition INCEPTION Inception is the initial stage of the project. Inception is not a requirements phase but it is a feasibility phase where complete investigation takes place to support a decision to continue or stop .It deals with • Approximate vision • Business case • Scope • Vague estimates.

OBJECT ORIENTED ANALYSIS AND DESIGN

UNIT II-OBJECT ORIENTED ANALYSIS

2.1 USE CASE MODEL

A use-case model is a model of how different types of users interact with the system to solve a problem. As such, it describes the goals of the users, the interactions between the users and the system, and the required behavior of the system in satisfying these goals.

A use-case model consists of a number of model elements. The most important model elements are: use cases, actors and the relationships between them.

A use-case diagram is used to graphically depict a subset of the model to simplify communications. There will typically be several use-case diagrams associated with a given model, each showing a subset of the model elements relevant for a particular purpose. The same model element may be shown on several use-case diagrams, but each instance must be consistent. If tools are used to maintain the use-case model, this consistency constraint is automated so that any changes to the model element (changing the name for example) will be automatically reflected on every use-case diagram that shows that element.

The use-case model may contain packages that are used to structure the model to simplify analysis, communications, navigation, development, maintenance and planning.

Much of the use-case model is in fact textual, with the text captured in the Use-Case Specifications that are associated with each use-case model element. These specifications describe the flow of events of the use case.

The use-case model serves as a unifying thread throughout system development. It is used as the primary specification of the functional requirements for the system, as the basis for analysis and design, as an input to iteration planning, as the basis of defining test cases and as the basis for user documentation

Basic model elements

The use-case model contains, as a minimum, the following basic model elements.

Actor

A model element representing each actor. Properties include the actors name and brief description. See Concept: Actor for more information.

Use Case

A model element representing each use case. Properties include the use case name and use case specification. See Artifact: Use Case and Concept: Use Case for more information.

Associations

Associations are used to describe the relationships between actors and the use cases they participate in. This relationship is commonly known as a “communicates-association”.

Advanced model elements

The use-case model may also contain the following advanced model elements.

Subject

A model element that represents the boundary of the system of interest.

Use-Case Package

A model element used to structure the use case model to simplify analysis, communications, navigation, and planning. If there are many use cases or actors, you can use use-case packages to further structure the use-case model in much the same manner you use folders or directories to structure the information on your hard-disk.

You can partition a use-case model into use-case packages for several reasons, including:

To reflect the order, configuration, or delivery units in the finished system thus supporting iteration planning.

To support parallel development by dividing the problem into bite-sized pieces.

To simplify communication with different stakeholders by creating packages for containing use cases and actors relevant to a particular stakeholder.

Generalizations

A relationship between actors to support re-use of common properties.

Dependencies

A number of dependency types between use cases are defined in UML. In particular, <<extend>> and <<include>>.

<<extend>> is used to include optional behavior from an extending use case in an extended use case.

<<include>> is used to include common behavior from an included use case into a base use case in order to support re-use of common behavior.

The latter is the most widely used dependency and is useful for:

Factoring out behavior from the base use case that is not necessary for the understanding of the primary purpose of the use case to simplify communications.

Factoring out behavior that is in common for two or more use cases to maximize re-use, simplify maintenance and ensure consistency.

2.1.1 USE CASES UNDER THE MICROSCOPE

Use Case Driven

A use case is a sequence of actions, performed by one or more actors (people or non-human entities outside of the system) and by the system itself, that produces one or more results of value to one or more of the actors. One of the key aspects of the Unified Process is its use of use cases as a driving force for development. The phrase use case driven refers to the fact that the project team uses the use cases to drive all development work, from initial gathering and negotiation of requirements through code. (See "Requirements" later in this chapter for more on this subject.)

Use cases are highly suitable for capturing requirements and for driving analysis, design, and implementation for several reasons.

Use cases are expressed from the perspective of the system's users, which translates into a higher comfort level for customers, as they can see themselves reflected in the use case text. It's relatively difficult for a customer to see himself or herself in the context of requirements text.

Use cases are expressed in natural language (English or the native language of the customers). Well-written use cases are also intuitively obvious to the reader.

Use cases offer a considerably greater ability for everyone to understand the real requirements on the system than typical requirements documents, which tend to contain a lot of ambiguous, redundant, and contradictory text. Ideally, the stakeholders should regard use cases as binding contracts between customers and developers, with all parties agreeing on the system that will be built.

Use cases offer the ability to achieve a high degree of traceability of requirements into the models that result from ongoing development. By keeping the use cases close by at all times, the development team is always in touch with the customers' requirements.

Use cases offer a simple way to decompose the requirements into chunks that allow for allocation of work to sub teams and also facilitate project management. (See "Use Case Model" in Chapter 2 for information about breaking use cases up into UML packages.) This is not the same as functional decomposition, though; see Use Case Driven Object Modeling with UML (Rosenberg and Scott, 1999) for an explanation of the difference

2.1.2 EXTENDS AND USES ASSOCIATIONS

Extend relationships

In UML modelling, you can use an extend relationship to specify that one use case (extension) extends the behaviour of another use case (base). This type of relationship reveals details about a system or application that are typically hidden in a use case.

The extend relationship specifies that the incorporation of the extension use case is dependent on what happens when the base use case executes. The extension use case owns the extend relationship. You can specify several extend relationships for a single base use case.

While the base use case is defined independently and is meaningful by itself, the extension use case is not meaningful on its own. The extension use case consists of one or several behaviour sequences (segments) that describe additional behaviour that can incrementally augment the behaviour of the base use case. Each segment can be inserted into the base use case at a different point, called an extension point.

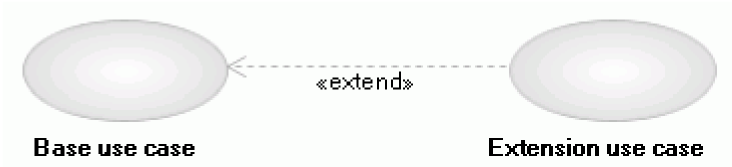
The extension use case can access and modify the attributes of the base use case; however, the base use case is not aware of the extension use case and, therefore, cannot access or modify the attributes and operations of the extension use case.

You can add extend relationships to a model to show the following situations:

- A part of a use case that is optional system behaviour
- A sub flow is executed only under certain conditions
- A set of behaviour segments that may be inserted in a base use case

Extend relationships do not have names.

As the following figure illustrates, an extend relationship is displayed in the diagram editor as a dashed line with an open arrowhead pointing from the extension use case to the base use case. The arrow is labelled with the keyword «extend».



Example

You are developing an e-commerce system in which you have a base use case called Place Online Order that has an extending use case called Specify Shipping Instructions. An extend relationship points from the Specify Shipping Instructions use case to the Place Online Order use case to indicate that the behaviours in the Specify Shipping Instructions use case are optional and only occur in certain circumstances.

2.1.3 IDENTIFYING THE ACTORS

Use Case Analysis: How to Identify Actors?

Identify Candidate Actors for Use Cases

Primary vs Supporting Actors

Actor Identification Process

Actor Role Description

Identifying actors is one of the first steps in use case analysis. Each type of external entities with which the system must interact is represented by an actor. For example, the operating environment of a software system consists of the users, devices, and programs that the system interacts with. These are called actors which has the following characteristics:

An actor in use case modeling specifies a role played by a user or any other system that interacts with the subject.

An Actor models a type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data), but which is external to the subject.

Actors may represent roles played by human users, external hardware, or other subjects.

Actors do not necessarily represent specific physical entities but merely particular facets (i.e., “roles”) of some entities that are relevant to the specification of its associated use cases.

A single physical instance may play the role of several different actors and a given actor may be played by multiple different instances.

Types of actors include:

Users

database systems

clients and servers

cloud platforms

devices

Types of Actors

Identify Candidate Actors for Use Cases

Candidate actors include groups of users who will require help from the system to perform their tasks and run the system’s primary or secondary functions, as well as external hardware, software, and other systems.

Define each candidate actor by naming it and writing a brief description. Includes the actor’s area of responsibility and the goals that the actor will attempt to accomplish when using the system. Eliminate actor candidates who do not have any goals.

These questions are useful in identifying actors:

Which user groups require help from the system to perform their tasks?

Which user groups are needed to execute the system’s most obvious main functions?

Which user groups are required to perform secondary functions, such as system maintenance and administration?

Will the system interact with any external hardware or software system?

Any individual, group or phenomenon that fits one or more of these categories is a candidate for an actor.

Primary vs Supporting Actors

Primary actor of a use case is the stakeholder that calls on the system to deliver one of its services. It has a goal with respect to the system – one that can be satisfied by its operation. The primary actor is often, but not always, the actor who triggers the use case.

Supporting Actors: A supporting actor in a use case in an external actor that provides a service to the system under design.

Note That:

Supporting actors may or may not have goals that they expect to be satisfied by the use case, the primary actor always has a goal, and the use case exists to satisfy the primary actor. It might be an external server or a web service.

A primary actor initiates an interaction with the system.

The system initiates interactions with secondary actors.

Here are a few questions to guide the identification of primary and secondary actors:

Who are the primary actors?

What roles do they play in the application domain?

What are their roles? What tasks must they perform?

How will they interact with the system? GUI? Telephone? Web?

Who are the secondary actors?

What services do they provide?

How will the system communicate with them? Network? Port?

What specific APIs do secondary actors provide for communication and services?

Actor Identification Process

To define what will be handled by the system and what will be handled outside the system (system scope, i.e. manual or automated procedure).

To define who (actors) and what (the functionalities needed) will interact with the system.

To outline the functionality of the system (help to identify use cases)

Actor Role Description

Define each actor by writing a brief description that includes the actor's area of responsibility, and what the actor needs the system for. Because actors represent things outside the system, you need not describe them in detail. You can basically make list all your actors with their role description and their objectives in a tabular format as shown below:

Actor / Role Name	Role Description and Objective

Actor

Briefly describe the role of the actor to the system and how the actor will use the system

What is the actor's goal?

What does the actor need from the system?

What is the expected outcome of the system?

Customer Customers will place food orders and may or may not order juice. When the order is served, the customer will eat his/her meal and pay the check.

Waiter The waiter will receive the food order from the customer and confirm the order with the cook and serve food to the customer.

Use Case Description example:

A user clicks the search button on an application's user interface. The application sends an SQL query to a database system. The database system responds with a result set. The application formats and displays the result set to the user.

In this scenario:

The user is a primary actor because he initiates the interaction with the system (application).

The database system is a secondary actor because the application initiates the interaction by sending an SQL query.

Once you have identified your actors and their goals, you have now created your initial list of high-level use cases. Remember, effective use cases must have understandable actors and goals

2.1.4 GUIDELINES FOR FINDING USE CASES

Before we can produce a use case diagram we must first identify the groups of related scenarios - the use cases. In addition we need to identify the initiators of the use cases - the actors. Recall from the previous sections, actors reside outside of the system and interact with it; use cases describe the functionality that helps actors achieve their goals. There are many approaches to identifying actors and use cases. In this section we present a method for doing this.

To identify use cases we will take the following steps:

Step 1: Identify candidate system actors.

Step 2: Identify the goals of the actors.

Step 3: Identify the candidate use cases.

Step 4: Identify the start point for each use case.

Step 5: Identify the end point for each use case.

Step 6: Refine and scope units of interaction.

Now we discuss each of these steps in more detail.

Step 1: Identify candidate system actors.

Read through the requirements documentation and make a note of all the candidate system actors. Remember an actor is not just a person but may also be an external system such as a credit card verification service. Actors interact with the system and reside outside of it. If you find that multiple terms have been used to describe the same actor, group these terms together and use a generic term. Be aware, it is crucial that you are certain that these terms are describing the same actor.

Activity: Identify the Actors in the Case Study

Learning Objective: This exercise will give you some insight into the process of identifying the system actors that are responsible for initiating uses cases.

Read through the UWEFlix case study and identify the actors. Remember from the previous section, actors are external to the system and not part of it. They are not necessarily human and can be external systems such as credit card verification services. For example, Manager could be an actor. Any others?

Step 2: Identify the goals of the actors.

Use cases describe the functionality required of the system in order for actors to achieve their goals. Therefore you need to identify what the goals of your candidate actors are. At this stage you may find that you have multiple actors requiring the same or similar system functionality, at this stage do not worry about this. For example, the manager needs to decide what films to shown in which screens.

Step 3: Identify the candidate use cases.

Now that you have identified the candidate actors and the goals of these actors we can identify the candidate list of use cases. Remember a use case represents a substantial piece of system functionality, not just a single method in software. Use cases cover a group of related scenarios, for example, a use case called "purchase ticket" will include the scenario where payment is unsuccessful in addition to the typical payment success scenario. Using the actors goals try and identify the text in the requirements document that corresponds to these goals. Choose an initial name to describe the candidate use case and paste the textual commentary from the requirements document underneath the use case name. At this point do not worry about common functionality in multiple use cases.

Activity: Identify the Use Cases in the Case Study

Learning Objective: This exercise will give you some insight into the process of identifying the uses cases that represent the required system functionality.

Read through the case study and identify the use cases. Remember from the previous section that use cases are groups of related scenarios. For example, we may have a use case for deleting film.

Step 4: Identify the start point for each use case.

You may have already started to do this when producing your list of candidate use cases but you need to identify the start point for each use case. To do this, look for an actor and an initial event. You will find this is easier to do with some use cases rather than others. For example, with the "DeleteFilm" use case, the initial event may be to choose a film.

Step 5: Identify the end point for each use case.

In a manner similar to that of step 2, you need to identify the beneficial result of the use case, the end point. The purpose of this step is to allow you to refine the size of use cases, ensuring that a candidate use case is neither too small nor too big.

Step 6: Refine and scope units of interaction.

At this point all of the functionality described in the requirements document needs to be covered by at least one candidate use case. You now need to work through the list of candidate use cases refining size and scope. Remember after completing this step you are not after a fixed final list of actors and use cases, it is still possible to refine this later.

2.1.5 HOW DETAILED USE CASE MUST BE?

Detail the Flow of Events of the Use Case To top of page

You should already have a short, step-by-step description of the use-case flow of events. This is created in the Activity: Find Actors and Use Cases. Use this step-by-step description as a starting point, and gradually make it more detailed.

Describe use cases according to the standards decided for the project (documented in the Artifact: Use-Case Modeling Guidelines). Decide on the following points before describing the use cases so that you are consistent across use cases:

How does the use case start? The start of the use case must clearly describe the signal that activates the use case. Write, for example, "The use case can start when … happens."

How does the use case terminate? You should clearly state whatever happens in the course of the flow to terminate the use case. Write, for example, "When … happens, the use case terminates."

How does the use case interact with actors? To minimize any risk of misunderstanding say exactly what will reside inside the system, and what will reside outside the system. Structure the description as a series of paragraphs, in which each paragraph expresses an action in the format: "When the actor does …, the system does …." You can also emphasize interaction by writing that the use case sends and receives signals from actors, for example: "The use case starts when it receives the signal 'start' from the Operator."

How does the use case exchange data with an actor? If you like, you can refer to the arguments of the signals, but it might be better to write, for example, "The use case starts when the User logs into the system by giving his name and password."

How does the use case repeat some behavior? You should try to express this in natural language. However, in exceptional cases, it might be worthwhile to use code-like constructs, such as "WHILE-END WHILE," "IF-THEN-ELSE," and "LOOP-END LOOP," if the corresponding

natural language terms are difficult to express. In general, however, you should avoid using such code-like constructs in use-case descriptions because they are hard to read and maintain.

Are there any optional situations in a use case's flow of events? Sometimes an actor is presented with several options. Team members should write this in the same way. For example:

"The actor chooses one of the following, one or more times

How should the use case be described so that the customer and the users can understand it? The use of methodology-specific terminology, such as use case, actor, and signal, might make the text unnecessarily hard to grasp. To make the text easier to read, you might enumerate the actions, or adopt some other strategy. Whatever strategy you use should be specified in the general use-case-modeling guidelines so that you keep it in mind during the entire activity of describing use cases.

Concentrate on describing what is done in the use case, not how specific problems internal to the system should be solved. When working with object models, you may have to complement the description with details about how things work, so do not make the description overly detailed at this point. Describe only what you believe will be stable later on.

If a use case's flow of events has become too encompassing or complex, or if it appears to have parts that are independent of one another, split it into two or more use cases.

When you write the descriptive text, refer to the glossary. As fresh terms evolve from new concepts, include them in the glossary. Do not change the definition of a term without informing the appropriate project members.

The Content of a Flow of Events Description

A flow of events description explores:

How and when the use case starts.

Example:

"The use case can start when the function 'Administer Order' is activated by a user."

When the use case interacts with the actors, and what data they exchange.

Example:

"To create a new order, the user activates the function 'New' and then specifies the following mandatory data concerning the order: name, network elements (at least one), and type of measurement function. Optional data can also be specified concerning the order: a comment (a small textual description). The user then activates the function 'Ok,' and a new order is created in the system."

Note: You must be explicit regarding the data exchanged between the actors and the use case; otherwise, the customer and the users will probably not understand the use-case description.

How and when the use case uses data stored in the system, or stores data in the system.

Example:

"The user activates the function 'Modify' to modify an existing order, and specifies an order number (small integer). The system then initializes an order form with the name of the order, its network elements, and its type of measurement function. This data is retrieved from a secondary storage device."

How and when the use case ends.

Example:

"The use case ends when the function 'Exit' is activated by the Orderer."

You should also describe odd or exceptional flows of events. An exceptional flow is a subflow of the use case that does not adhere to the use case's normal or basic behavior. This flow may nevertheless be necessary in any complete description of the use case's behavior. A typical example of an exceptional flow is the one given in the first example. If the use case receives some unexpected data (that the actor is not the one expected in that particular context) it terminates. Having the wrong actor and terminating prematurely are not in the typical flow of events.

Other "do's and don'ts" to consider when you describe a use case include:

Describe the flow of events, not just the use case's functionality or purpose.

Describe only flows that belong to the use case, not what is going on in other use cases that work in parallel with it.

Do not mention actors who do not communicate with the use case in question.

Do not provide too much detail when you describe the use case's interaction with any actor.

If the order of the subflows described for the use case does not have to be fixed, do not describe it as if it does have to be fixed.

Use the terms in the common glossary and consider the following in writing the text:

Use straightforward vocabulary. Don't use a complex term when a simple one will do.

Write short, concise sentences.

Avoid adverbs, such as very, more, rather, and the like.

Use correct punctuation.

Avoid compound sentences.

For more information, see Guidelines: Use Case, the discussions on contents and style of the flow of events.

Structure the Flow of Events of the Use Case To top of page

A use case's flow of events can be divided into several subflows. When the use case is activated the subflows can combine in various ways if the following holds true:

The use case can proceed from one of several possible paths, depending on the input from a given actor, or the values of some attribute or object. For example, an actor can decide, from several options, what to do next, or, the flow of events may differ if a value is less or greater than a certain value.

Example:

Part of the description of the use case Withdraw Money in an automated teller machine system could be "The amount of money the client wants to withdraw from the account is compared to the balance of the account. If the amount of money exceeds the balance, the client is informed and the use case terminates. Otherwise, the money is withdrawn from the account."

The use case can perform some subflows in optional sequences.

The use case can perform several subflows at the same time.

You must describe all these optional or alternative flows. It is recommended that you describe each subflow in a separate supplement to the Flow of Events section, and should be mandatory for the following cases:

Subflows that occupy a large segment of a given flow of events.

Exceptional flows of events. This helps the use case's basic flow of events to stand out more clearly.

Any subflow that can be executed at several intervals in the same flow of events.

If a subflow involves only a minor part of the complete flow of events, it is better to describe it in the body of the text.

Example:

"This use case is activated when the function 'administer order' is called for by either of the actors Orderer or Performance Manager Administrator. If the signal does not come from one of these actors, the use case will terminate the operation and display an appropriate message to the user. However, if the actor is recognized, the use case proceeds by....."

You can illustrate the structure of the flow of events with an activity diagram, see Guidelines: Activity Diagram in the Use-Case Model.

For more information, see Guidelines: Use Case, structure of the flow of events.

Illustrate Relationships with Actors and Other Use Cases To top of page

Create use-case diagrams showing the use case and its relationships to actors and other use cases. A diagram of this type functions as a local diagram of the use case, and should be related to it. Note that this kind of local use-case diagram is typically of little value, unless the use case has use-case relationships that need to be explained, or if there is an unusual complexity among the actors involved.

For more information, see Guidelines: Use-Case Diagram.

Describe the Special Requirements of the Use Case To top of page

Any requirements that can be related to the use case, but that are not taken into consideration in the Flow of Events of the use case, should be described in the Special Requirements of the use case. Such requirements are likely to be nonfunctional.For more information, see Guidelines: Use Case, special requirements.

Describe Communication Protocols To top of page

Develop a communication protocol if the actor is another system or external hardware. The description of the use case should state if some existing protocol (maybe even a standardized one) is to be used. If the protocol is new, you must fully describe it during object-model development.

Describe Preconditions of the Use Case To top of page

A precondition on a use case explains the state the system must be in order for the use case to be possible to start.

Example:

In order for an ATM system to be able to dispense cash, the following preconditions must be satisfied:

The ATM network must be accessible.

The ATM must be in a state ready to accept transactions.

The ATM must have at least some cash on hand that it can dispense.

The ATM must have enough paper to print a receipt for at least one transaction.

These would all be valid preconditions for the use case Dispense Cash.

Take care to describe the system state; avoid describing the detail of other incidental activities that may have taken place prior to this use case.

Preconditions are not used to create a sequence of use cases. There will never be a case where you have to first perform one use case, then another, in order to have a meaningful flow of events. If you feel there is a need to do this, it is likely that you have decomposed the use-case model too much. Correct this problem by combining the sequentially dependent use cases into a single use case. If this makes the resulting use case too complex, consider techniques for structuring use cases, as presented in Structure the Flow of Events of the Use Case above, or in the Activity: Structure the Use-Case Model.

For more information, see Guidelines: Use Case, Preconditions and Postconditions.

Describe Postconditions of the Use Case To top of page

A postcondition on a use case lists possible states the system can be in at the end of the use case. The system must be in one of those states at the end of the execution of the use case. It is also used

to state actions that the system performs at the end of the use case, regardless of what occurred in the use case.

Example:If the ATM always displays the ‘Welcome’ message at the end of a use case, this could be documented in the postcondition of the use case.

Similarly, if the ATM always closes the customer’s transaction at the end of a use case like Withdraw Cash, regardless of the course of events taken, that fact should be recorded as a postcondition for the use case.

Postconditions are used to reduce the complexity and improve the readability of the flow-of-events of the use case.

2.1.6 DIVIDING USE CASES INTO PACKAGES

Guidelines: Use-Case Package

Use-Case Package A use-case package is a collection of use cases, actors, relationships, diagrams, and other packages; it is used to structure the use-case model by dividing it into smaller parts.

A model structured into smaller units is easier to understand. It is easier to show relationships among the model's main parts if you can express them in terms of packages. A package is either the top-level package of the model, or stereotyped as a use-case package. You can also let the customer decide how to structure the main parts of the model.

If there are many use cases or actors, you can use use-case packages to further structure the use-case model. A use-case package contains a number of actors, use cases, their relationships, and other packages; thus, you can have multiple levels of use-case packages (packages within packages).

The top-level package contains all top-level use-case packages, all top-level actors, and all top-level use cases.

Use To top of page

You can partition a use-case model into use-case packages for many reasons:

You can use use-case packages to reflect order, configuration, or delivery units in the finished system.

Allocation of resources and the competence of different development teams may require that the project be divided among different groups at different sites. Some use-case packages are suitable for a group, and some for one person, which makes packages a naturally efficient way to proceed with development. You must be sure, however, to define distinct responsibilities for each package so that development can be performed in parallel.

You can use use-case packages to structure the use-case model in a way that reflects the user types. Many change requirements originate from users. Use-case packages ensure that changes from a particular user type will affect only the parts of the system that correspond to that user type.

In some applications, certain information should be accessible to only a few people. Use-case packages let you preserve secrecy in areas where it is needed.

2.1.7 NAMING A USE CASE

The first step in writing the use cases for a project is to define the scope of the project. One way to do that is to list the use case names that define all of the user goals that are in scope. To do that, you need to know how to write good use case names. Good use case names also serve as a great reference and provide context and understanding throughout the life of the project.

Goals of Use Case Naming

Use case names are also known as use case titles. When creating names, we have a set of goals:

Clearly indicate the user goal represented by the use case.

Avoid specifying the design of the system.

Make people want to read the use case, not dread reading it.

Allow for evolution of use cases across releases.

Define the scope of the project.

Write consistently

Common Use Case Mistakes

We identified the top ten use case mistakes in a couple of articles about a year ago. They still hold true today:

From Top Five Use Case Blunders:

Inconsistency

Incorrectness

Wrong Priorities

Implementation Cues

Broken Traceability

From Top Ten Use Case Mistakes:

Unanticipated Error Conditions

Overlooking System Responses

Undefined Actors

Impractical Use Cases

Out of Scope Use Cases

Writing good use case names will help avoid errors in consistency, implementation cues, scope management, and traceability. They will also help us make people want to read the use cases. Think of the use case name as the headline of a magazine article – does it make you want to read it, or avoid it?

Good use case names also serve as reminders of what a particular use case does. Weeks after we've written a use case, a quick scan of the title will remind us of what the use case represents. On a large project with dozens of use cases, this is invaluable.

Tips For Writing Good Use Case Names

Here are the best practices we've adopted, and some we've collected from around the internet.

Good Use Case Names Reflect User Goals. A good use case name reflects the goal of the user (or external system). A name like "Process Invoices" doesn't tell us what's being done – is it collections, organization, auditing, or some other function? A more insightful name would be "Collect Late Payments From Customers." The goal in this example is to collect payments from delinquent customers. The second name does a much better job of defining what the user is trying to do when they perform the use case.

Good Use Case Names are As Short As Possible. Some people suggest 5 words, or even two words. There are just too many examples that make setting specific word-count limits impractical. In the previous example, "Collect Late Payments From Customers," which words would you remove without losing meaning? This name is as short as we can make it without losing clarity. This short name is better than "Collect Late Payments From Customers Who Are Past-Due."

Good Use Case Names Use Meaningful Verbs. Usually people will suggest that we should prefer strong verbs to weak verbs. That is effective advice for general writing. For writing use cases, we can be more specific. A meaningless verb is one that, while indicating action, does not specify the action with enough detail. "Process the Order" can be improved with a more meaningful verb. "Validate the Ordered Items" makes it much more clear what the user is trying to achieve.

Good Use Case Names Use An Active Voice. A call to action is a hallmark of good writing. Using an active voice will inspire action more than a passive voice. "Calculate Profitability" is more inspiring than "Profitability is Calculated."

Good Use Case Names Use The Present Tense. "Create New Account" is in the present tense. "New Account Was Created" is in the past tense. The present tense implies what the user is trying to do, not something that has already been done.

Good Use Case Names Don't Identify The Actor. Some people prefer to name the actor in the use case, because it is more specific. We like the idea of using evolutionary use cases to manage the delivery of functionality across releases. When we do this, we are often releasing the first version of the use case for one actor, and the next version for another actor. For example, "Rank Employee Performance" might be our use case. In the first release, we want to enable the functionality for supervisors – who can rank their direct employees. In the second release, we want to add the ability for managers to rank the employees that report to multiple supervisors. We prefer having two versions of the same use case over having two use cases (Rank Direct/Indirect Employee Performance).

Good Use Case Names Are Consistent. We should always apply the same set of rules across all of our use case names. Inconsistent application of the names will create a sense of discord for our

readers. Consistent names will make it more comfortable for readers, and provide a sense of cohesion for the overall project

2.2 OBJECT ANALYSIS: CLASSIFICATION

2.2.1 INTRODUCTION

Classification

Classification is the means whereby we order knowledge. In object-oriented design, recognizing the sameness among things allows us to expose the commonality within key abstractions and mechanisms and eventually leads us to smaller applications and simpler architectures. Unfortunately, there is no golden path to classification. To the reader accustomed to finding cookbook answers, we unequivocally state that there are no simple recipes for identifying classes and objects. There is no such thing as the "perfect" class structure, nor the "right" set of objects. As in any engineering discipline, our design choices are a compromise shaped by many competing factors

2.2.2 CLASSIFICATIONS THEORY

The Theory of Classification

Object-oriented programming languages were originally developed in advance of any formal theory of classification and, as a result, their treatment of class was muddled and ill-founded. Strongly-typed languages assumed that the notion of class was equated with type, and behavioural compatibility could be described using a terminology of types and subtyping, which strictly did not apply. The type systems of languages affected by such misunderstandings were sometimes incorrect. Other languages assumed that objects have class and type independently, relegating the notion of class to a mere implementation construct.

To counter this confusion, a mathematical theory of classification was developed, which encompassed other approaches to type abstraction, such as "type constructors", "generic parameters", "classes", "inheritance" and "polymorphism". The theory extended earlier work on F-bounded polymorphism by providing second- and higher-order typed record combination for objects modelled as records of functions. The theory captures notions such as single, multiple and mixin inheritance and unifies the treatment of generic polymorphism with inheritance-based polymorphism. Object-oriented languages such as Smalltalk, C++, Eiffel and Java may be explained within the theory, which also shows how the syntactic treatment of type polymorphism may be simplified in future object-oriented languages.

Project History

This project grew out of a personal dissatisfaction with the various ad-hoc typing mechanisms used to express polymorphism in object-oriented programming languages. Inspired by Cook's account of how a strongly-typed object-oriented language violated mathematical subtyping, by permitting covariant specialisation of method argument types [1], work started on an object-oriented language, which was to support strict subtyping (Simons, 1991) and whose subtyping rules also regulated the permissible refinements of pre- and postconditions (Simons, 1994a).

Subsequently, the type system of this language, Brunel, was completely redesigned to accommodate Cook's new theory of F-bounded polymorphism, which treated a class as a second-order construct in the function-bounded polymorphic λ-calculus [2], [3]. From this, an initial approach was developed to unify the object-oriented notion of inclusion polymorphism with more traditional

notions of parametric polymorphism and type constructors (Simons and Cowling, 1992). The unified theory of classification was summarised in (Simons, 1996a) and was presented as tutorials at ECOOP and OOPSLA (Simons, 1993a, 1993b, 1994b), in which the consequences of adopting an F-bounded type system were explored. A class is a function-bounded second-order construct in the λ-calculus, whereas an object type is the least fixed point, a first-order construct. Different parametric type checking rules apply, which permit the covariant specialisation of method argument types, as object-oriented languages intuitively seem to require.

Distinguishing the notion of class from type

The unified theory of classification was the subject of the PhD thesis (Simons, 1995a), which also described the Brunel 2.0 language as an exemplar of the new type system. This presented a unified treatment of inheritance and genericity (templates), showing how the interactions of different kinds of extension and specialisation by inheritance and by parametric specialisation and substitution are confluent. This eventually extended Cook's model from second-order to higher-order, but could be approximated by dependent second-order types some of the time, such as the treatment of mixins in (Simons, 1995b). The impact of the theory upon another object-oriented language, Eiffel, was described in (Simons, 1995c, 1995d). This provided an alternative second-order, F-bounded solution to the type-failure problem, which Cook had addressed earlier using first-order subtyping [1]. The revised solution permitted covariant method argument specialisation, as found in Eiffel, but also replaced three different syntactic mechanisms for expressing polymorphism (conformance, anchored types, constrained genericity) by a single mechanism. The popular impact of the theory of classification upon language terminology and design notation was discussed in (Simons, 1996b).

Most recently, the Theory of Classification has reached a much wider audience, through the publication of an invited series of articles in the Journal of Object Technology. This is a popular monograph, serialised in 20 parts (Simons, 2002 - 2005). This material has over 100 citations and is collated on a number of specialist websites, such as the community weblog for programming language research, Lambda the Ultimate. The material has formed the basis for a number of courses on object-oriented type theory at the Universities of Berne (Switzerland), Sheffield (UK) and others.

2.2.3 APPROACHES FOR IDENTIFYING CLASSES

1 Identification of Classes

2 Object Oriented Analysis (OOA) OOA is process by which we identify classes that play role in achieving system goals & requirements. Classification is the process of checking to see if an object belongs to a category or a class which guides us in making decisions about modularization.

3 Approaches for identifying classes & their behaviours in problem domain Noun phrase approach Common class patterns Use–case driven approach Classes, Responsibilities and Collaboration (CRC) approach

4 Noun Phrase Approach Read through the requirements or use cases looking for noun phrases. Nouns in textual description are considered to be classes & verbs to be methods of classes. As a whole, classes are grouped in to three categories: Relevant classes, Fuzzy classes and Irrelevant classes.

5 The series of steps in this approach are as follows: 1.Identifying Tentative Classes: Following are guidelines for selecting classes in an application Look for nouns, and noun phrases in use cases Some classes are implicit or taken from general knowledge All classes must make sense in application domain Avoid computer implementation classes – defer them to the design stage

6 Selecting Classes form Relevant & Fuzzy Categories: Following guidelines help in selecting candidate classes from relevant & fuzzy categories of classes in problem domain Redundant classes: If more than one word is being used to describe same idea, select one that is most meaningful in the context of system. This part of building a common vocabulary for the system as a whole. Adjectives classes: Adjectives can be in many ways. An adjective can suggest a different kind of object, different use of the same object or it could be utterly irrelevant Attribute classes: Tentative objects that are used only as values should be defined or restated as attributes and not as a class Irrelevant classes: Each class must have a purpose and every class should be clearly defined and necessary. Classes which cannot be given statement of purpose are eliminated

7 Elimination& Refining: The process of eliminating redundant classes & refining remaining classes is not sequential. It can be done forth & back among steps as of wish.

8 Example Case study (ATM) The following section provides a description of the ATM system requirements. The bank client must be able to deposit an amount to and withdraw an amount from his or her accounts using the touch screen at the bank atm. Each transaction must be recorded and the client must be able to review all transactions performed against a given account. In this bank client can have two types of accounts: a checking account and savings account. The bank client will interact with the bank system by going through the approval process. After the approval process, the client can perform the transaction. The steps involved in transaction are: a)Insert ATM card b)Perform approval process c)Ask type of transaction d)Enter type e)Perform transaction f)Eject card g)Take card

9 Initially the client enters the PIN code that consists of 4 digits. If PIN is valid the clients account will be available. Else an appropriate message is displayed on the screen. Neither a checking nor savings account can have a negative balance. If the balance in saving account is less than the withdrawal amount requested, the transaction will stop and the client will be notified.

10 Solution: List of nouns identified are: Account Account balance Amount Atm card Bank Bank client Card Checking account Client Clients account Four digits Invalid PIN Message PIN PIN code Savings Savings account Transaction Transaction history Touch Screen

11 Eliminate irrelevant classes Account Account balance Amount Atm card Bank Bank client Card Checking account Client Clients account Four digits Invalid PIN Message PIN PIN code Savings Savings account Transaction Transaction history Touch Screen

12 Reviewing the redundant classes and building a common vocabulary Client, Bank client = Bank client Account, Client's Account = Account PIN, PIN code = PIN Checking, Checking Account= Checking Account Savings, Savings Account =Savings Account

13 Here is the revised list of classes Account Account balance Amount Atm card Bank Bank client Card Checking account Client Clients account Four digits Invalid PIN Message PIN PIN code Savings Savings account Transaction Transaction history

14 Reviewing the classes containing adjectives In this example, we have no classes containing adjectives that we can eliminate.

15 Reviewing the Possible Attribute Amount -A value not a class Account Balance -An attribute of the Account class. Invalid PIN -only a value, not class Transaction history - attribute of transaction class PIN -attribute of Bank client class Password: An attribute of Bank Client class

16 Revised list after eliminating the attributes Account Account balance Amount Atm card Bank Bank client Card Checking account Client Clients account Four digits Invalid PIN Message PIN PIN code Savings Savings account Transaction Transaction history

17 Review the class purpose: ATM card class Bank client class: Details about client Account class: abstract class Savings and checking account: inherits account class. Transaction class: keeps track of the records.

18 Common Class Patterns Approach It is based on a knowledge base of common classes that have been proposed by various researchers. They have compiled & listed the following patterns for finding the candidate class & object Concept class: It encompasses principles that are not tangible but used to organize or keep track of business activities or communications. Eg: Performance is an concept class object Events class: These are points in time that must be recorded. Associated with things remembered are attributes such as who, what, when, where, how or why. Eg: Landing Organization class: It is collection of people, resources, facilities or groups to which users belong; their capabilities have a defined mission, whose existence is independent of individuals People class: It represent different roles users play in interacting with application. It is also known as person, roles and roles played class divided into 2 – users & non-users information Places class: Places are physical locations that system must keep information about. Eg: Stores Tangible things & devices class: This class includes physical objects or groups of objects that are tangible & devices with which application interacts. Eg: cars, pressure sensors

19 CRC Approach The Classes, Responsibilities and Collaborators process consists of three steps 1. Identify classes' responsibilities (and identify classes) 2. Assign responsibilities 3. Identify collaborators Classes are identified & grouped by common attributes, which also provides candidates for super classes. Responsibilities are distributed; they should be as general as possible & placed as high as possible in inheritance hierarchy. The idea in locating collaborators is to identify how classes interact.

2.2.4 NOUN PHRASE APPROACH

Noun phrase approach:

Nouns in the textual description are considered to be classes and verbs to be methods of the classes. All plurals are changed to singular, the nouns are listed and the list divided into relevant classes, fuzzy classes and irrelevant class.

Look for the noun phrases through the use cases.

Three categories:

Relevant classes.

Fuzzy classes.

Irrelevant classes.

Identifying tentative classes.

Look for noun phrases and nouns in the use cases.

Some classes are implicit or taken from general knowledge.

All classes must make sense in the application domain.

Carefully choose and define class names.

Selecting classes from the relevant and fuzzy classes.

Redundant classes.

Adjective classes.

Attribute classes.

Irrelevant classes.

Guide lines for selecting classes in all application:

Carefully choose and define class names.

All classes must make sense in the application domain, avoid computer implementation classes – defer there to the design stage.

Look for nouns and noun-phrases in the use cases.

Some classes are implicit or taken from general knowledge.

Example: Bank ATM system: Identifying classes by using noun phrase approach:

Initial list of Noun phrases candidate classes

Account

Account Balance

Amount

Approval process

ATM card

ATM machine

Bank

Bank client

Card

Cash

Check

Checking

Checking Account

Client

Client's Account

Currency

Dollar

Envelope

Four digits

Fund

Savings

Savings Account

Step

System

Transaction

Transaction history

Invalid PIN

Message

Money

Password

PIN

Pin Code

Record

The following irrelevant classes are removed from the above list:

Envelope

Four Digits

Step

Reviewing the Redundant classes and Building a common vocabulary:

Client, Bank client à Bank client

Account, Client's Account à Account

PIN, PIN code à PIN

Checking, Checking Accountà Checking Account

Savings, Savings Account à Savings Account

Fund, Money à Fund

ATM card, card à ATM card

Reviewing the classes containing adjectives

In this example, we have no classes containing adjectives that we can eliminate.

Reviewing the Possible Attribute

Amount à A value not a class

Account Balance à An attribute of the Account class

Invalid PIN à It is only a value, not class

Password à An attribute for Bank client class

Transaction history à An attribute of transaction class

PIN à An attribute of Bank client class

Reviewing the class purpose

The final candidate classes are:

ATM machine class

ATM card class

Bank client

Bank class

Account class

Checking Account class

Savings Account class

Transaction class

2.2.4.1 IDENTIFYING TENTATIVE CLASSES

Identifying tentative classes.

1.Look for noun phrases and nouns in the use cases.

2.Some classes are implicit or taken from general knowledge.

3.All classes must make sense in the application domain.

4.Carefully choose and define class names

2.2.4.2. THE VIANET BANK ATM SYSTEM

The ViaNet Bank ATM System: Identifying Classes by Using Classes, Responsibilities, and Collaborators We already identified the initial classes of the bank system. The objective of this example is to identify objects' responsibilities such as attributes and methods in that system. Account and Transaction provide the banking model. Note that Transaction assumes an active role while money is being dispensed and a passive role thereafter. The class Account is responsible mostly to the BankClient class and it collaborates with several objects to fulfill its responsibilities. Among the responsibilities of the Account class to the BankClient class is to keep track of the BankClient balance, account number, and other data that need to be remembered. These are the attributes of the Account class. Furthermore, the Account class provides certain services or methods, such as means for BankClient to deposit or withdraw an amount and display the account's Balance (see Figure). Classes, Responsibilities, and Collaborators encourages team members to pick up the card and assume a role while "executing" a scenario. It is not unusual to see a designer with a card in each hand, waving them about, making a strong identification with the objects while describing their collaboration. Ward Cunningham writes: Classes, Responsibilities,andCollaboratorscards work by taking people through programmingepisodestogether.

150 As cardsare written for familiar objects, all participants pick up the same context and ready themselves for decision making. Then, by waving cards and pointing fingers and yelling statements like, "no, this guy should do that," decisions are made. Finally, the group starts to relax as consensus has been reached and the issue becomes simply finding the right words to record a decision as a responsibility on a card. In similar fashion other cards for the classes that have been identified earlier in this chapter must be created, with the list of their responsibilities and their collaborators. As you can see from Figure , this process is iterative. Start with few cards (classes) then proceed to play "what if." If the situation calls for a responsibility not already covered by one of the objects, either add the responsibility to an object or create a new object to address that responsibility.

2.2.4.3 INITIAL LIST OF NOUN PHRASES

Initial list of Noun phrases candidate classes

Account

Account Balance

Amount

Approval process

ATM card

ATM machine

Bank

Bank client

Card

Cash

Check

Checking

Checking Account

Client

Client’s Account

Currency

Dollar

Envelope

Four digits

Fund

Savings

Savings Account

Step

System

Transaction

Transaction history

Invalid PIN

Message

Money

Password

PIN

Pin Code

Record

2.2.4.4 REVIEWING THE CLASSES CONTAINING ADJECTIVES

Analysis is an attempt to build a model that describes the application domain -- developers do this

Takes place after (or during) requirements specification

The analysis model will typically consist of all three types of models discussed before:

Functional model (denoted with use cases)

Analysis object model (class and object idagrams)

Dynamic model

At this level, note that we are still looking at the application domain.

This is not yet system design

However, many things discovered in analysis could translate closely into the system design

Goal is to completely understand the application domain (the problem at hand, any constraints that must be adhered to, etc.)

New insights gained during analysis might cause requirements to be updated.

Analysis activities include:

Identifying objects (often from use cases as a starting point)

Identfying associations between objects

Identifying general attributes and responsibilities of objects

Modeling interactions between objects

Modeling how individual objects change state -- helps identify operations

Checking the model against requirements, making adjustments, iterating through the process more than once

Finding the objects

We often think of objects in code as mapping to some object we want to represent in the real world. Although this isn't always the case.

Here are some categories of objects to look for:

Entity objects -- these represent persistent information tracked by a system. This is the closest parallel to "real world" objects.

Boundary objects -- these represent interactions between user and system. (For instance, a button, a form, a display)

Control objects -- usually set up to manage a given usage of the system. Often represent the control of some activity performed by a system

UML diagrams can include a label known as a stereotype, above the class name in a class diagram. This would be placed inside <<>> marks, like this:

<<entity>>

<<boundary>>

<<control>>

Note: Different sources and/or "experts" will give other categorizations of types of objects

There are some different popular techniques for identifying objects. Two traditional and popular ones that we will discuss are:

natural language analysis (i.e. parts of speech)

CRC cards

It also helps to interact with domain experts -- these are people who are already well-versed in the realm being studied.

Note that the goal in the analysis phase is NOT to find implementation specific objects, like HashTable or Stack.

This stage still models the application domain

Using natural language analysis

Pioneered by Russell Abbott (1983), popularized by Grady Booch

Not perfect, but coupled with other techniques, it's a good start

This can be done from a general problem description, or better, from a use case or scenario

Map parts of speech to object model components.

nouns usually map to classes, objects, or attributes

verbs usually map to operations or associations

Part of speech	model component	Examples
Proper noun	Instance (object)	Alice, Ace of Hearts
Common noun	Class (or attribute)	Field Officer, PlayingCard, value
Doing verb	Operation	Creates, submits, shuffles
Being verb	Inheritance	Is a kind of, is one of either
Having verb	Aggregation/Composition	Has, consists of, includes
Modal verb	Constraint	Must be
Adjective	Helps identify an attribute	a yellow ball (i.e. color)

Identifying different object types

Finding Entity Objects

Some things to look for. These may be candidates for objects, or they may help identify objects:

Terms that are domain-specific in use cases

Recurring nouns

Real-world entities and activities tracked by system

Use good naming conventions. Good to use names from the application domain -- they understand their own terminology best

Example: In a ReportEmergency use case -- "A field officer sumits information to the system by filling out a form and pressing the 'Send Report' button"

FieldOfficer is a real world entity that interacts with the system

This is also likely an actor from the use case

As an actor, FieldOfficer is an external entity

But we see that the field officer submits information -- here's data to be tracked

We'll create the entity object type EmergencyReport, as that's the more common name for the information the officer submits (according to client)

Finding Boundary Objects

Identify general user interface controls that initiate a use case

Note: Don't bother with the visual details here. This will evolve later

Identify forms or windows for entering data into a system

Identify messages used by system to respond to a user

Finding Control Objects

Control objects can help manage communication and interaction of other objects

If a use case is conplex and involves many objects, create a control object to manage the use case

Identify one control object per actor involved in a use case

Life span of control object should last through the use case

2.2.4.5 REVIEWING THE POSSIBLE ATTRIBUTES

A set of attributes for the objects that are to be instantiated from the class. Generally, different objects of a class have some difference in the values of the attributes. Attributes are often referred as class data. A set of operations that portray the behavior of the objects of the class

According to some research on object-oriented analysis, there are usually four types of attributes

([2], [7], [10]): (a) descriptive, (b) naming, (c)

state information, and (d) referential:

• Descriptive Attributes: Descriptive

attributes are facts that are intrinsic to

each entity. If the value of a descriptive

attribute changes, it only means that

some aspects of an entity (instance) have

changed. From a problem-domain

perspective, it is still the same entity. For

example, if Hassan gains one pound,

from nearly all problem-domain

perspectives, Hassan is still a person.

More importantly, Hassan is still the

same person as before when he gained

one pound.

• Naming Attributes: These are used to name or label an entity/object. Typically, they are somewhat arbitrary and frequently used as identifiers or as part of an identifier. If the value of a naming attribute changes, it only means that a new name has been given to the same entity/object. In fact, naming attributes do not have to be unique. For example, if Hassan changes his name to Ali, all that changes; yet, his weight, height, etc. are still the same.

• State-Information Attributes: They are used to keep a history of the entity. These are usually needed to capture the states of the finite state machines used to implement the dynamic aspect of the behavior of objects. For example, the attribute 'speed' of a "Car" is used to control the different states of the object. The State-Information attributes are important to build simulation software [26].

• Referential-Information Attributes: They are some facts that connect one

object to another so as to capture

relationships. For example, assume that

“Driver” and “Car” object. Then, we can

define the attribute ‘driver’ for “Car”

object.

Coad and Yourdon (1991) put it very well when

they said [5]: "Make each attribute capture an

atomic concept". The atomic concept means that

an attribute will contain a single value or a

tightly-related grouping of values that some

applications treat as a whole. In this respect,

attributes of objects are divided into individual

attributes and composite attributes:

 Single-Value Attribute: They are

Downloaded from ijiepr.iust.ac.ir at 15:45 IRDT on Wednesday April 21st 2021

individual attributes such as ‘age’,

‘salary’, and ‘weight’ of an object

“Person”.

 Group-Values Attribute: They are

composite data items such as legal

‘name’, ‘address’, and ‘birth date’ for an

object “Person”.

Some attributes of an object are dependent on other attributes. In fact, the attributes can be calculated from other attributes and are generally used to increase the performance of an application. In this aspect, attributes can be classified into the following categories:

 Basic Attribute: It is a simple attribute that adheres to an object such as ‘birthday’, ‘salary’, and ‘weight’ of a “Person”.

 Performance Attribute: It is calculable based on the basic attributes such as ‘age’ that can be calculated from the current year and the ‘birthday’ of the object “Person”.

3-3. Eliminating incorrect attributes

Attributes are rarely fully described in a requirements document. Fortunately, they seldom affect the basic structure of the model of objectorientation. Analysts must draw upon their knowledge of the application domain and the real world to find them. Because most guidelines for identifying attributes do not help differentiate between incorrect attributes and real attributes, the following suggestions help eliminate incorrect attributes ([8],[9], [15]):

 Objects: If the independent existence of the attribute is more important than its value, then the attribute is an object and there needs to be a link to it. For example, consider a “Person” object, an instance of the class person, in an application of Staff Administration. The ‘address’ or ‘city’ in which the Person lives is an attribute. In this application, if analysts do not manipulate the ‘address’ without knowing to which person the address belongs, then it is an attribute. However, if analysts manipulate the ‘address’ as an entity itself, like Military Service application, then the ‘address’ or ‘city’ should be an object with a link between it and Person.

 Qualifiers: If the value of an attribute depends on a particular context such as Sport or Police, then we must consider it as a qualifier. For example, the badgeNumber of Player/Police is not really his/her attribute. It really qualifies as a link "plays/works" between the object Player/Police and the object Club/Police-Center.

 Names: It is noted that a name is an attribute when it does not depend on the context. For example, a ‘person’ name is an attribute of “Person”. Note that an attribute, as in a person's name, does not have to be unique. However, names are usually qualifiers and not attributes. They usually either define a role in an association or define a subclass or superclass abstraction. For example, ‘parent’ and ‘teacher’ are not attributes of the object “Person”. Both are probably roles for associations. Another example is a male person and a female person. There are two ways to capture this: we must (a) consider ‘gender’ as an attribute of the object “Person” or (b) make two subclasses.

 Identifiers: We must not prepare a list of the unique identifiers that object-oriented languages need for making an unambiguous reference to an object. This is implicitly assumed to be part of the model; however, the application domain identifiers are listed. For example, in most accounting applications, an

‘account code’ is an attribute of an object

“Account”, whereas a transaction

identifier is probably not an attribute of

the “Account”.

 Link attributes: If an attribute of an

object depends on the presence of a link,

then it is an attribute of the link and not

of the objects in the link. We must

consider the link as an associative object

and make the proposed attribute as one of

its attributes. For example, assume that

Ali is married to Maryam. The date of

their marriage is an attribute of the

“is_married” association and not an

attribute of Ali or Maryam.

 Discordant: We must omit minor

attributes that do not affect the methods.

For example, in the application of student

registration in university, the number of

brothers/sister of a student must be

removed from the list of attributes of the

student.

2.2.5 COMMON CLASS PATTERNS APPROACH

2.2.5.1 VIANET BANK ATM SYSTEM: IDENTIFYING CLASSES BY USING COMMON CLASS PATTERNS

The ViaNet Bank ATM System: Identifying Classes by Using Classes, Responsibilities, and Collaborators We already identified the initial classes of the bank system. The objective of this example is to identify objects' responsibilities such as attributes and methods in that system.

Account and Transaction provide the banking model. Note that Transaction assumes an active role while money is being dispensed and a passive role thereafter. The class Account is responsible mostly to the BankClient class and it collaborates with several objects to fulfill its responsibilities. Among the responsibilities of the Account class to the BankClient class is to keep track of the BankClient balance, account number, and other data that need to be remembered. These are the attributes of the Account class. Furthermore, the Account class provides certain services or methods, such as means for BankClient to deposit or withdraw an amount and display the account's Balance (see Figure). Classes, Responsibilities, and Collaborators encourages team members to pick up the card and assume a role while "executing" a scenario. It is not unusual to see a designer with a card in each hand, waving them about, making a strong identification with the objects while describing their collaboration. Ward Cunningham writes: Classes, Responsibilities,andCollaboratorscards work by taking people through programmingepisodestogether.

150 As cardsare written for familiar objects, all participants pick up the same context and ready themselves for decision making. Then, by waving cards and pointing fingers and yelling statements like, "no, this guy should do that," decisions are made. Finally, the group starts to relax as consensus has been reached and the issue becomes simply finding the right words to record a decision as a responsibility on a card. In similar fashion other cards for the classes that have been identified earlier in this chapter must be created, with the list of their responsibilities and their collaborators. As you can see from Figure , this process is iterative. Start with few cards (classes) then proceed to play "what if." If the situation calls for a responsibility not already covered by one of the objects, either add the responsibility to an object or create a new object to address that responsibility.

2.2.6 USE CASE DRIVEN APPROACH

Use Case Driven

A **use case** is a sequence of actions, performed by one or more **actors** (people or non-human entities outside of the system) and by the system itself, that produces one or more results of value to one or more of the actors. One of the key aspects of the Unified Process is its use of use cases as a driving force for development. The phrase *use case driven* refers to the fact that the project team uses the use cases to drive all development work, from initial gathering and negotiation of requirements through code. (See "Requirements" later in this chapter for more on this subject.)

Use cases are highly suitable for capturing requirements and for driving analysis, design, and implementation for several reasons.

- Use cases are expressed from the perspective of the system's users, which translates into a higher comfort level for customers, as they can see themselves reflected in the use case text. It's relatively difficult for a customer to see himself or herself in the context of requirements text.
- Use cases are expressed in natural language (English or the native language of the customers). Well-written use cases are also intuitively obvious to the reader.
- Use cases offer a considerably greater ability for everyone to understand the real requirements on the system than typical requirements documents, which tend to contain a lot of ambiguous, redundant, and contradictory text. Ideally, the stakeholders should regard use cases as binding contracts between customers and developers, with all parties agreeing on the system that will be built.

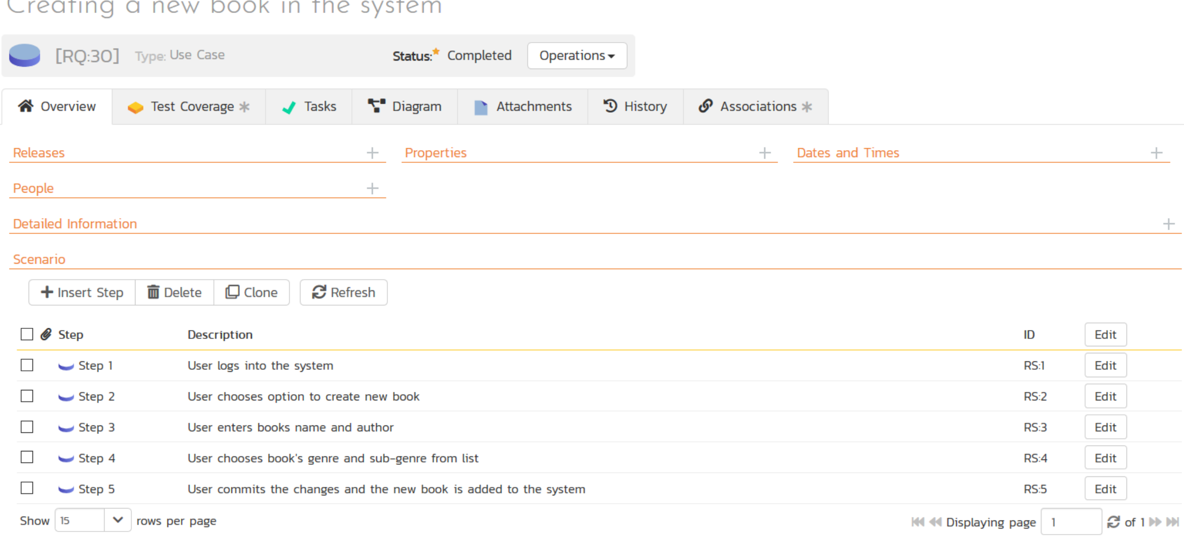
- Use cases offer the ability to achieve a high degree of traceability of requirements into the models that result from ongoing development. By keeping the use cases close by at all times, the development team is always in touch with the customers' requirements.
- Use cases offer a simple way to decompose the requirements into chunks that allow for allocation of work to subteams and also facilitate project management. (See "Use Case Model" in Chapter 2 for information about breaking use cases up into UML packages.) This is *not* the same as functional decomposition, though; see *Use Case Driven Object Modeling with UML* (Rosenberg and Scott, 1999) for an explanation of the difference.

2.2.6.1 IMPLEMENTATION OF SCENARIOS

Use Cases and Scenarios

Once you have developed an initial set of **Functional Requirements** during the Requirements Gathering phase you will have a good understanding of the intended behavior of the system. You will understand what functionality is desired, what constraints are imposed, and what business objectives will be satisfied. However, one shortcoming of a traditional 'laundry-list' of requirements is that they are static and don't concern themselves with the different business processes that need be supported by one feature.

For example, in our fictitious online library system, the functionality for managing returns would need to handle the separate situations where a borrower returns a book early and when he/she returns it late. Although the same functionality is involved, they are different situations and the system would need to handle the separate conditions in each **use case**. Therefore use-cases are a valuable way of uncovering implied functionality that occurs due to different ways in which the system will be used. Also use-cases provide a great starting point for the test cases that will be used to test the system.

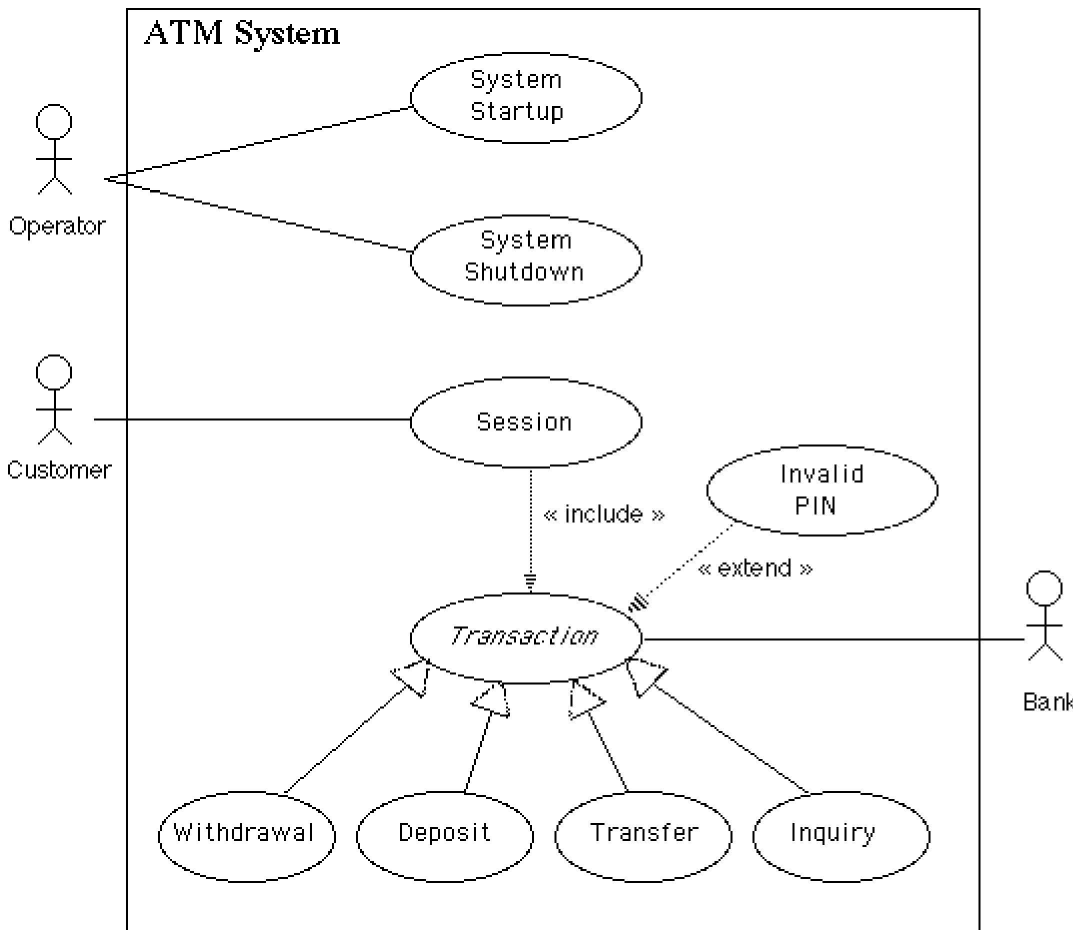


A **use case** is a definition of a specific business objective that the system needs to accomplish. A use-case will define this process by describing the various external actors (or entities) that exist outside of the system, together with the specific interactions they have with the system in the accomplishment of the business objective

2.2.6.2 THE VIANET BANK ATM SYSTEMS

An automated teller **machine** (**ATM**) or the automatic **banking machine** (ABM) is a **banking** subsystem (subject) that provides **bank** customers with access to financial transactions

in a public space without the need for a cashier, clerk, or **bank** teller. ... Customer may need some help from the **ATM**.



(Click on a use case above to go to the flow of events for that use case)

Flows of Events for Individual Use Cases

System Startup Use Case

The system is started up when the operator turns the operator switch to the "on" position. The operator will be asked to enter the amount of money currently in the cash dispenser, and a connection to the bank will be established. Then the servicing of customers can begin.

[Interaction Diagram]

System Shutdown Use Case

The system is shut down when the operator makes sure that no customer is using the machine, and then turns the operator switch to the "off" position. The connection to the bank will be shut down. Then the operator is free to remove deposited envelopes, replenish cash and paper, etc.

[Interaction Diagram]

Session Use Case

A session is started when a customer inserts an ATM card into the card reader slot of the machine. The ATM pulls the card into the machine and reads it. (If the reader cannot read the card due to improper insertion or a damaged stripe, the card is ejected, an error screen is displayed, and the session is aborted.) The customer is asked to enter his/her PIN, and is then allowed to perform one or more transactions, choosing from a menu of possible types of transaction in each case. After each transaction, the customer is asked whether he/she would like to perform another. When the customer is through performing transactions, the card is ejected from the machine and the session ends. If a transaction is aborted due to too many invalid PIN entries, the session is also aborted, with the card being retained in the machine.

The customer may abort the session by pressing the Cancel key when entering a PIN or choosing a transaction type.

[Interaction Diagram]

Transaction Use Case

Note: Transaction is an abstract generalization. Each specific concrete type of transaction implements certain operations in the appropriate way. The flow of events given here describes the behavior common to all types of transaction. The flows of events for the individual types of transaction (withdrawal, deposit, transfer, inquiry) give the features that are specific to that type of transaction.

A transaction use case is started within a session when the customer chooses a transaction type from a menu of options. The customer will be asked to furnish appropriate details (e.g. account(s) involved, amount). The transaction will then be sent to the bank, along with information from the customer's card and the PIN the customer entered.

If the bank approves the transaction, any steps needed to complete the transaction (e.g. dispensing cash or accepting an envelope) will be performed, and then a receipt will be printed. Then the customer will be asked whether he/she wishes to do another transaction.

If the bank reports that the customer's PIN is invalid, the Invalid PIN extension will be performed and then an attempt will be made to continue the transaction. If the customer's card is retained due to too many invalid PINs, the transaction will be aborted, and the customer will not be offered the option of doing another.

If a transaction is cancelled by the customer, or fails for any reason other than repeated entries of an invalid PIN, a screen will be displayed informing the customer of the reason for the failure of the transaction, and then the customer will be offered the opportunity to do another.

The customer may cancel a transaction by pressing the Cancel key as described for each individual type of transaction below.

All messages to the bank and responses back are recorded in the ATM's log.

[Interaction Diagram]

Withdrawal Transaction Use Case

A withdrawal transaction asks the customer to choose a type of account to withdraw from (e.g. checking) from a menu of possible accounts, and to choose a dollar amount from a menu of possible amounts. The system verifies that it has sufficient money on hand to satisfy the request before sending the transaction to the bank. (If not, the customer is informed and asked to enter a different amount.) If the transaction is approved by the bank, the appropriate amount of cash is dispensed by the machine before it issues a receipt. (The dispensing of cash is also recorded in the ATM's log.)

A withdrawal transaction can be cancelled by the customer pressing the Cancel key any time prior to choosing the dollar amount.

[Interaction Diagram]

Deposit Transaction Use Case

A deposit transaction asks the customer to choose a type of account to deposit to (e.g. checking) from a menu of possible accounts, and to type in a dollar amount on the keyboard. The transaction is initially sent to the bank to verify that the ATM can accept a deposit from this customer to this account. If the transaction is approved, the machine accepts an envelope from the customer containing cash and/or checks before it issues a receipt. Once the envelope has been received, a second message is sent to the bank, to confirm that the bank can credit the customer's account - contingent on manual verification of the deposit envelope contents by an operator later. (The receipt of an envelope is also recorded in the ATM's log.)

A deposit transaction can be cancelled by the customer pressing the Cancel key any time prior to inserting the envelope containing the deposit. The transaction is automatically cancelled if the customer fails to insert the envelope containing the deposit within a reasonable period of time after being asked to do so.

[Interaction Diagram]

Transfer Transaction Use Case

A transfer transaction asks the customer to choose a type of account to transfer from (e.g. checking) from a menu of possible accounts, to choose a different account to transfer to, and to type in a dollar amount on the keyboard. No further action is required once the transaction is approved by the bank before printing the receipt.

A transfer transaction can be cancelled by the customer pressing the Cancel key any time prior to entering a dollar amount.

[Interaction Diagram]

Inquiry Transaction Use Case

An inquiry transaction asks the customer to choose a type of account to inquire about from a menu of possible accounts. No further action is required once the transaction is approved by the bank before printing the receipt.

An inquiry transaction can be cancelled by the customer pressing the Cancel key any time prior to choosing the account to inquire about.

[Interaction Diagram]

Invalid PIN Extension

An invalid PIN extension is started from within a transaction when the bank reports that the customer's transaction is disapproved due to an invalid PIN. The customer is required to re-enter the PIN and the original request is sent to the bank again. If the bank now approves the transaction, or disapproves it for some other reason, the original use case is continued; otherwise the process of re-entering the PIN is repeated. Once the PIN is successfully re-entered, it is used for both the current transaction and all subsequent transactions in the session. If the customer fails three times to enter the correct PIN, the card is permanently retained, a screen is displayed informing the customer of this and suggesting he/she contact the bank, and the entire customer session is aborted. If the customer presses Cancel instead of re-entering a PIN, the original transaction is cancelled.

2.2.7 CLASSES, RESPONSIBILITIES AND COLLABORATORS

Class-responsibility-collaboration (**CRC**) **cards** are a brainstorming tool used in the design of object-oriented software. They were originally proposed by Ward Cunningham and Kent Beck as a teaching tool,[1] but are also popular among expert designers[2] and recommended by extreme programming supporters.[3] Martin Fowler has described in his book about UML that if you want to explore multiple alternative interactions quickly, you may be better off with CRC cards, as that avoids a lot of drawing and erasing. It's often handy to have a CRC card session to explore design alternatives and then use sequence diagram to capture any interactions that you want to refer to later. CRC cards are part of the design phase within system/software development and gives a good overview if you go from use case descriptions to CRC cards and then to class diagrams. This allows a smoother transition with a greater overview and allows the developer to easier implement a system with low binding and high cohesion. CRC cards are used after use case descriptions and before class diagrams within software development but can be skipped for smaller projects.

CRC cards are usually created from index cards. Members of a brainstorming session will write up one CRC card for each relevant class/object of their design. The card is partitioned into three areas:[1][2]

1. On top of the card, the **class** name
2. On the left, the **responsibilities** of the class
3. On the right, **collaborators** (other classes) with which this class interacts to fulfill its responsibilities

2.2.7.1 CRC PROCESS

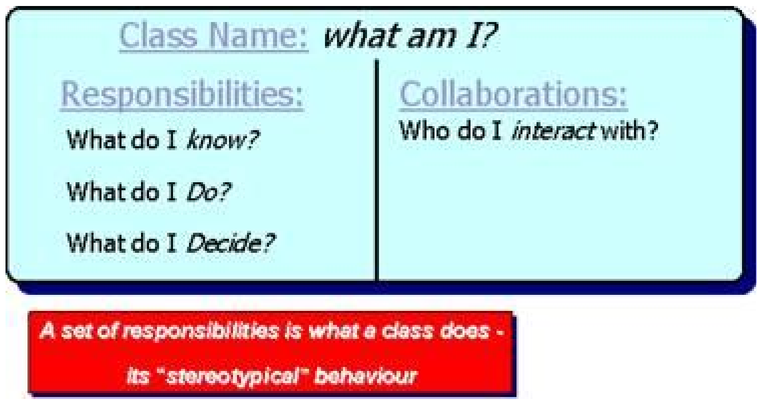
CRC Process

One of the most valuable techniques in coming up with a good OO design is to explore object interactions, because it focuses on behaviour rather than data.

Once you have a reasonable list of candidate classes in your OO design you can further evaluate their place in a particular system by identifying their responsibilities - what they do, and who they need to work with to do this - their collaborations. The process is referred to as the Classes Responsibilities and Collaborations process or more commonly as the CRC process.

Put simply, the idea is that by establishing what the responsibilities of a particular class are and who that class will collaborate with it is possible to justify the existence of that class in the system. Classes that have no responsibilities can be removed because they do not add value to the system, they are superfluous. By concentrating on the responsibilities of a class, the encapsulation of the candidate class is reinforced. The set of responsibilities of the class yields the class role - also known as its "stereotypical" behaviour.

The idea of responsibility driven analysis of classes originated with Rebecca Wirfs-Brock in the book "Designing Object-Oriented Software" (1994). Rebecca also originated the simple but practical, highly effective idea of CRC cards, where each candidate class is captured on its own CRC card (or piece of paper) as shown below.



CRC cards are used as part of a role-play that will help you validate class selection. The idea is that following the initial phases of class elimination you write the names of all remaining candidate classes on a series of cards. Then you work through the textual narrative of the system requirements, i.e. the use case descriptions assigning responsibilities to classes. Responsibilities include determining doing something, knowing something and decision making.

Wherever possible you should work in a group to carry out this process as you will typically find you will get better results if you have to justify decisions to peers. You should divide up the cards equally between members of the group and work through the requirements of the system, annotating cards during this process. As with the other parts of class elimination you will find the CRC process to be recursive

2.2.7.2 THE VIANET BANK ATM SYSTEMS

THE VIANET BANK ATM SYSTEM: Scenario with a Sequence Diagram: Object behavior Analysis A sequence diagram represents the sequence and interactions of a given use case or scenario. Sequence diagrams are among the most popular UML diagrams and, if used with an object model or class diagram, can capture most of the information about a system . Most object-to-object interactions and operations are considered events, and events include signals, inputs, decisions, interrupts, transitions, and actions to or from users or external devices. An event also is considered to be any action by an object that sends information. The event line represents a message sent from one object to another, in which the "from" object is requesting an operation be performed by the "to" object. The "to" object performs the operation using a method that its class contains. Developing sequence or collaboration diagrams requires us to think about objects that generate these events and therefore will help us in identifying classes. To identify objects of a system, we further analyze the lowest level use cases with a sequence and collaboration diagram pair (actually, most CASE tools such as SA/Object allow you to create only one, either a sequence or a collaboration diagram, and the system generates the other one). Sequence and collaboration diagrams represent the order in which things occur and how the objects in the system send messages to one another. These diagrams provide a macro-level analysis of the dynamics of a system. Once you start creating these diagrams, you may find that objects may need to be added to satisfy the particular sequence of events for the given use case.

You can draw sequence diagrams to model each scenario that exists when a BankClient withdraws, deposits, or needs information on an account. By walking through the steps, you can determine what objects are necessary for those steps to take place. Therefore, the process of creating sequence or collaboration diagrams can assist you in identifying classes or objects of the system. This approach can be combined with noun phrase and class categorization for the best results. We identified the use cases for the bank system. The following are the low level (executable) use cases: Deposit Checking Deposit Savings Invalid PIN Withdraw Checking Withdraw More from Checking Withdraw Savings Withdraw Savings Denied Checking Transaction History Savings Transaction History Let us create a sequence/collaboration diagram for the following use cases: .Invalid PIN use case .Withdraw Checking use case .Withdraw More from Checking use case Sequence/collaboration diagrams are associated with a use case. For example, to model the sequence/collaboration diagrams in SA/Object, you must first select a use case, such as the Invalid PIN use case, then associate a sequence or collaboration child process.

You can draw sequence diagrams to model each scenario that exists when a BankClient withdraws, deposits, or needs information on an account. By walking through the steps, you can determine what objects are necessary for those steps to take place. Therefore, the process of creating sequence or collaboration diagrams can assist you in identifying classes or objects of the system. This approach can be combined with noun phrase and class categorization for the best results. We identified the use cases for the bank system. The following are the low level (executable) use cases: Deposit Checking Deposit Savings Invalid PIN Withdraw Checking Withdraw More from Checking Withdraw Savings Withdraw Savings Denied Checking Transaction History Savings Transaction History

2.2.8 NAMING CLASSES

Why Naming Matters

Here are some of the benefits of proper class naming and naming conventions:

- You know what to expect from a certain class without looking at code or documentation, even if you aren't the person who created it or if it was written a long time ago.
- It's easy to search and navigate a codebase.
- It's easier to talk to your team when discussing problems/improvements.

- It makes onboarding newcomers easier, quicker, and less confusing.

A properly used MVP with an established naming convention is usually a good example of working naming expectations. If the class is named UseCase/Interactor, you'd expect it to contain business logic. But let's say that we have a convention to use UseCase for a single piece of logic and Interactor to put in similar UseCases. Let's also pretend we have a catalog application about all different kinds of monsters. You can have your own collection of monsters you have encountered, add them to favorites, and keep track of

limited edition monsters. By combining those two knowledge aspects (MVP + the monster app), you can tell, without even checking inside that:

GetAllMonstersUseCase — returns a collection of all monsters. GetMonsterByIdUseCase — requires that you pass an ID to get a Monster.

And if you find MonsterFilteringInteractor, you shouldn't be surprised to find methods such as getMonstersForArea(area) or getMonstersLargerThen(size) inside of it.

Now imagine you are looking for a method, which, for example, returns all carnivorous monsters. To do this, check in the filtering interactor, and if this method doesn't exist, you will at least have a nice idea about where the proper place to add it is.

Enough explanation for now; let's get to the naming itself, along with some important tips to keep in mind.

2.3 IDENTIFYING OBJECT RELATIONSHIPS, ATTRIBUTES AND METHODS

2.3.1 INTRODUCTION

In an object-oriented environment, objects take on an active role in a system.

Of course, objects do not exist in isolation but interact with each other.

Indeed, these interactions and relationships are the application.

Relationships in UML are used to represent a connection between structural, behavioral, or grouping things. It is also called a link that describes how two or more things can relate to each other during the execution of a system.

an **attribute** is a specification that defines a property of an object, element, or file. It may also refer to or set the specific value for a given instance of such. For clarity, **attributes** should more correctly be considered metadata

A **method** in object-oriented programming (OOP) is a procedure associated with a message and an object. ... This allows the sending objects to invoke behaviors and to delegate the implementation of those behaviors to the receiving object. A **method** in Java programming sets the behavior of a class object

2.3.2 ASSOCIATIONS

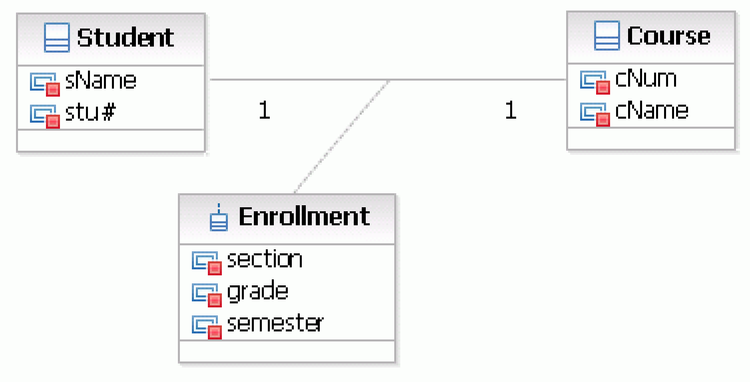
In **UML diagrams**, an **association** class is a class that is part of an **association** relationship between two other classes. ... For example, a class called Student represents a student and has an **association** with a class called Course, which represents an educational course

In UML diagrams, an association class is a class that is part of an association relationship between two other classes. You can attach an association class to an association relationship to provide additional information about the relationship. An association

class is identical to other classes and can contain operations, attributes, as well as other associations.

For example, a class called Student represents a student and has an association with a class called Course, which represents an educational course. The Student class can enroll in a course. An association class called Enrollment further defines the relationship between the Student and Course classes by providing section, grade, and semester information related to the association relationship.

As the following figure illustrates, an association class is connected to an association by a dotted line.



2.3.3 GUIDELINES FOR IDENTIFYING ASSOCIATION

Guidelines: Association

Association	An **association** models a bi-directional semantic connection among instances

Associations

Associations represent structural relationships between objects of different classes; they represent connections between instances of two or more classes that exist for some duration. Contrast this with transient links that, for example, exist only for the duration of an operation. These latter situations can instead be modeled using collaborations, in which the links exist only in particular limited contexts.

You can use associations to show that objects know about another objects. Sometimes, objects must hold references to each other to be able to interact, for example send messages to each other; thus, in some cases associations may follow from interaction patterns in sequence diagrams or collaboration diagrams.

Association Names

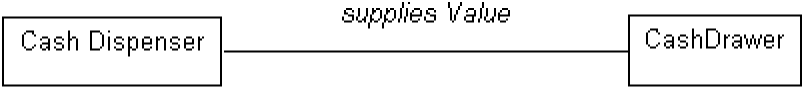
Most associations are binary (exist between exactly two classes), and are drawn as solid paths connecting pairs of class symbols. An association may have either a name or the association roles may have names. Role names are preferable, as they convey more information. In

cases where only one of the roles can be named, roles are still preferable to association names so long as the association is expected to be uni-directional, starting from the object to which the role name is associated.

Associations are most often named during analysis, before sufficient information exists to properly name the roles. Where used, association names should reflect the purpose of the relationship and be a verb phrase. The name of the association is placed on, or adjacent to the association path.

Example

In an ATM, the **Cash Drawer** provides the money that the **Cash Dispenser** dispenses. In order for the **Cash Dispenser** to be able to dispense funds, it must keep a reference to the **Cash Drawer** object; similarly, if the **Cash Drawer** runs out of funds, the **Cash Dispenser** object must be notified, so the **Cash Drawer** must keep a reference to the **Cash Dispenser.** An association models this reference.



An association between the **Cash Dispenser** and the **Cash Drawer**, named **supplies Value**.

Association names, if poorly chosen, can be confusing and misleading. The following example illustrates good and bad naming. In the first diagram, association names are used, and while they are syntactically correct (using verb phrases), they do not convey much information about the relationship. In the second diagram, role names are used, and these convey much more about the nature of the participation in the association.

Examples of good and bad usage of association and role names

Roles

Each end of an association is a **role** specifying the face that a class plays in the association**.** Each role must have a name, and the role names opposite a class must be unique. The role name should be a noun indicating the associated object's role in relation to the associating object. A suitable role name for a **Teacher** in an association with a **Course Section** would, for instance, be **lecturer**; avoid names like "**has**" and "**contains**", as they add no information about what the relationships are between the classes.

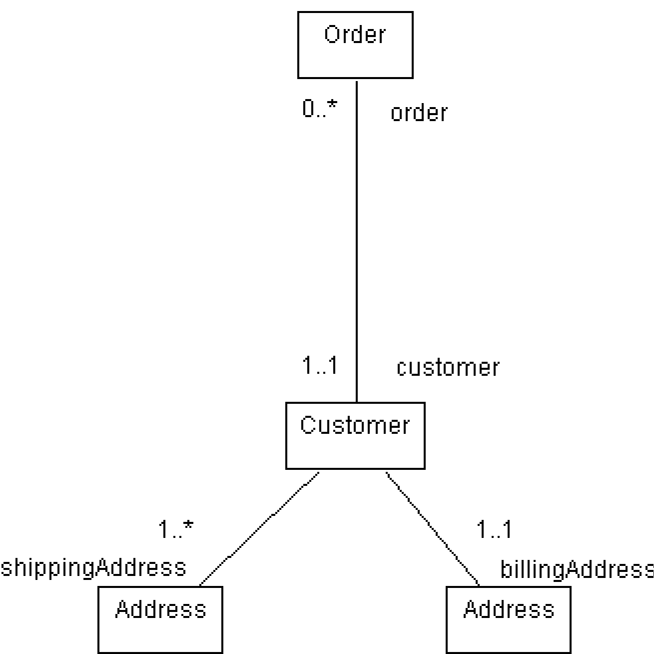
Note that the use of association names and role names is mutually exclusive: one would not use both an association name **and** a role name. Role names are preferable to

association names except in cases where insufficient information exists to name the role appropriately (as is often the case in analysis; in design role names should always be used). Lack of a good role name suggests an incomplete or ill-formed model.

The role name is placed next to the end of the association line.

Example

Consider the relationships between classes in an order entry system. A Customer can have two different kinds of Addresses: an address to which bills are sent, and a number of addresses to which orders may be sent. As a result, we have **two** associations between Customer and Address, as shown below. The associations are labeled with the **role** the associated address plays for the Customer.



2.3.4 COMMON ASSOCIATION PATTERNS

Association patterns provide guidance for modeling the associations that occur among objects within both the real world and the solution domains of computer applications. The patterns help the designer better understand and more precisely define the semantics of these associations, which allows them to be more easily and properly implemented. This paper describes a number of association patterns using Object Relationship Notation (ORN) and by doing so provides evidence for the effectiveness of this notation. It also shows how the development of database systems can be improved by an approach that uses association patterns to build a database model and then implements the model by mapping it to an ORN-extended database definition that is supported by a DBMS. The feasibility of this approach and the applicability of our association patterns have been validated by DBMS research prototypes and by the modeling, implementing, and testing of numerous associations. The **common** class **patterns** approach is based on a knowledge base of the **common** classes that have been proposed researchers.

2.3.5 SUPER – SUB CLASS RELATIONSHIPS

Super class–**subclass relationship** also known as generalization hierarchy, allow objects to be built from other objects. ... The **super-sub class** hierarchy is a **relationship** between classes, where one class is the parent (**super** or ancestor) class of another (derived) class

The other aspect of classification is identification of super-sub relations among classes.

For the most part, a class is part of a hierarchy of classes, where the top class is the most general one and from it descend all other, more specialized classes.

The super-sub class relationship represents the inheritance relationships between related classes, and the class hierarchy determines the lines of inheritance between class.

Superclass-subclass relationships, also known as generalization hierarchy, allow objects to be built from other objects.

Such relationships allow us to implicitly take advantage of the commonality of objects when constructing new classes.

The super-sub class hierarchy is a relationship between classes, where one class is the parent class of another (derived) class that the parent class also is known as the base or super class or ancestor.

The real advantage of using this technique is that we can build on what we already have and, more important, reuse what we already have.

Inheritance allows classes to share and reuse behaviors and attributes.

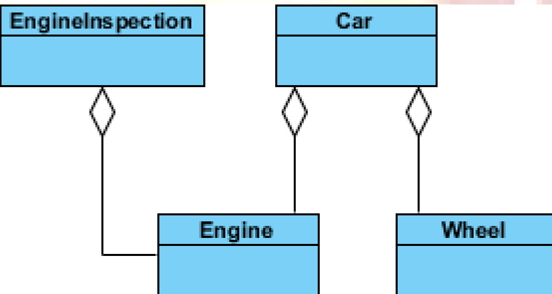
2.3.6 A- PART-OF RELATIONSHIP-AGGREGATION

Aggregation or composition is a relationship among classes by which a class can be made up of any combination of objects of other classes. It allows objects to be placed directly within the body of other classes.

Aggregation is a special form of association. It is a relationship between two classes like association, however **its** a directional association, which means it is strictly a one way association. It represents a HAS-A relationship.

Aggregation Example:

It's important to note that the **aggregation link doesn't state in any way that Class A owns Class B nor that there's a parent-child relationship** (when parent deleted all its child's are being deleted as a result) between the two. Actually, quite the opposite! The aggregation link is usually used to stress the point that Class A instance is not the exclusive container of Class B instance, as in fact the same Class B instance has another container/s.



An **aggregation** is a collection, or the gathering of things together. This relationship is represented by a “has a” relationship. In other words, aggregation is a group, body, or mass composed of many distinct parts or individuals For example, phone number list is an example of aggregation.

2.3.7 CASE STUDY

2.3.7.1 IDENTIFYING CLASSES RELATIONSHIPS

In object-oriented software design (OOD), classes are templates for defining the characteristics and operations of an object. Often, classes and objects are used interchangeably, one synonymous with the other. In actuality, a class is a specification that an object implements.

Identifying classes can be challenging. Poorly chosen classes can complicate the application's logical structure, reduce reusability, and hinder maintenance. This article provides a brief overview of object-oriented classes and offers tips and suggestions to identify cohesive classes.

Note: The following class diagrams were modeled using Enterprise Architect. Many other modeling tools exist. Use the one that is best suited for your purpose and project.

Object-oriented classes support the object-oriented principles of abstraction, encapsulation, polymorphism and reusability. They do so by providing a template, or blueprint, that defines the variables and the methods common to all objects that are based on it. Classes specify knowledge (attributes) - *they know things* - and behavior (methods) - *they do things*.

Classes are specifications for objects.

Derived from the Use Cases, classes provide an abstraction of the requirements and provide the internal view of the application.

Identifying classes

Identifying object-oriented classes is both a skill and an art. It's a process that one gets better at over time. For example, it's not unusual for inexperienced designers to identify too many classes. Modeling too many classes results in poor performance, unnecessary complexity and increased maintenance. On the other hand, too few classes tend to increase couplings, and make classes larger and unwieldy. In general, strive for class cohesiveness where behavior is shared between multiple, related classes rather than one very large class.

Cohesive classes reduce coupling, enable extensibility and increase maintainability.

Moreover, classes that seem obvious wind up being poor choices and classes that are initially hidden or that rely on problem domain knowledge wind up as the best choices.

Therefore, begin class modeling by identifying candidate classes - an initial list of classes from which the actual design classes will emerge. Design classes are the classes that are modeled. Candidate classes exist for the sole purpose of deriving the design classes. Initially, there will be a lot of candidate classes – that's good. However, through analysis, their number will be reduced as they are dropped, combined and merged.

Candidate classes provide the initial impetus to produce cohesive classes.

Candidate classes can be discovered in a variety of ways. Here are three:

1 Noun and noun phrases: Identify the noun and noun phrases, verbs (actions) and adjectives (attributes) from the Use Cases, Actor-Goal List, Application Narrative and Problem Description.

2 CRC cards: an informal, group approach to object modeling.

3 GRASP: A formal set of principles that assign responsibilities.

Each of these methods will yield a list candidate classes. The list won't be complete nor will every class be appropriate and there will likely be a mix of business and system oriented classes; i.e.; Student and StudentRecord, for example. That's fine. The goal is to identify the major classes - the obvious ones. Other classes will become apparent as the design process continues.

Once the list has been created, analyze the candidate classes for associations with other classes. Look for collaborating classes. How does each relate to each other and to the business process?

Sometimes, it's helpful to ask, "Why keep this class?" In other words, assume the class is redundant or unnecessary. Keep it only if it plays a collaborating role. Often you'll find the class' functionality is accomplished by another class or within the context of another class.

2.3.7.2 DEVELOPING A UML DIAGRAM BASED ON THE USE CASE ANALYSIS UML is a way of visualizing a software program using a collection of diagrams. The notation has evolved from the work of Grady Booch, James Rumbaugh, Ivar Jacobson, and the Rational Software Corporation to be used for object-oriented design, but it has since been extended to cover a wider variety of software engineering projects. Today, UML is accepted by the Object Management Group (OMG) as the standard for modeling software deTypes of UML Diagrams

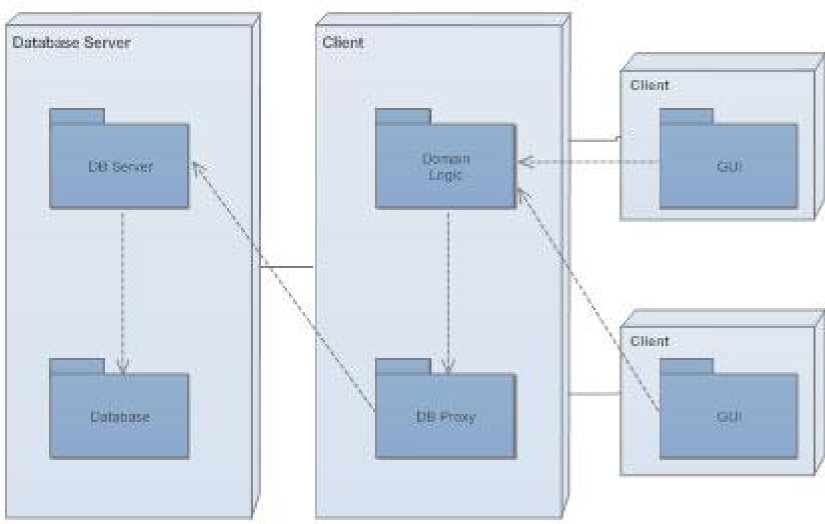
The current UML standards call for 13 different types of diagrams: class, activity, object, use case, sequence, package, state, component, communication, composite structure, interaction overview, timing, and deployment.

These diagrams are organized into two distinct groups: structural diagrams and behavioral or interaction diagrams.

Structural UML diagrams

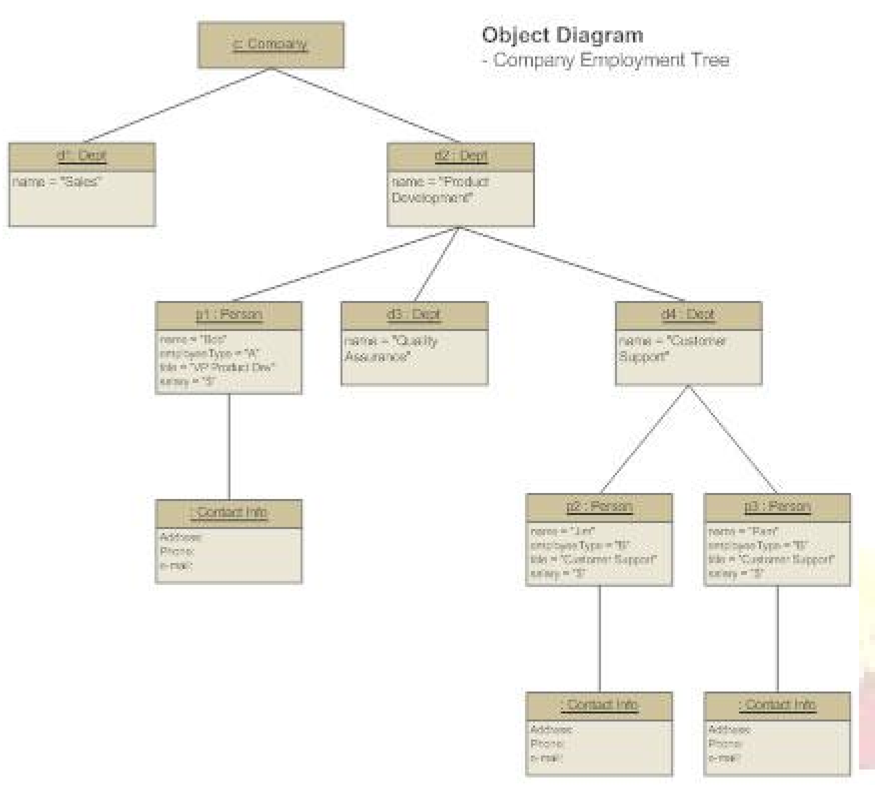
- Class diagram
- Package diagram
- Object diagram
- Component diagram
- Composite structure diagram
- Deployment diagram

Package diagrams organize elements of a system into related groups to minimize dependencies between packages.



Object Diagram

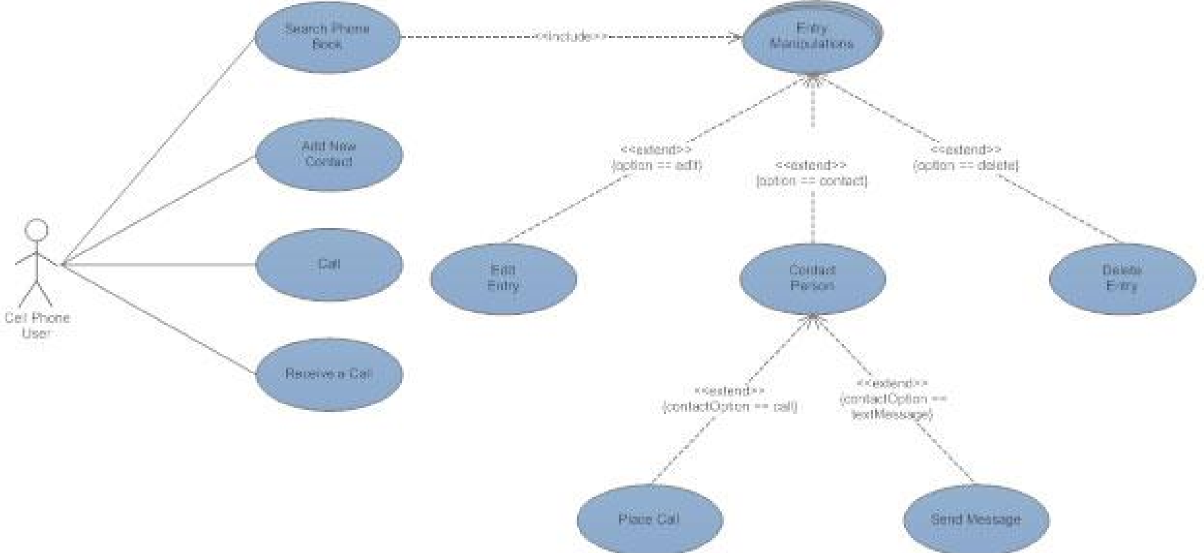
Object diagrams describe the static structure of a system at a particular time. They can be used to test class diagrams for accuracy.



Composite Structure Diagram: Composite structure diagrams show the internal part of a class.

Use Case Diagram

Use case diagrams model the functionality of a system using actors and use cases.



2.3.7.3 DEFINING ASSOCIATION RELATIONSHIPS

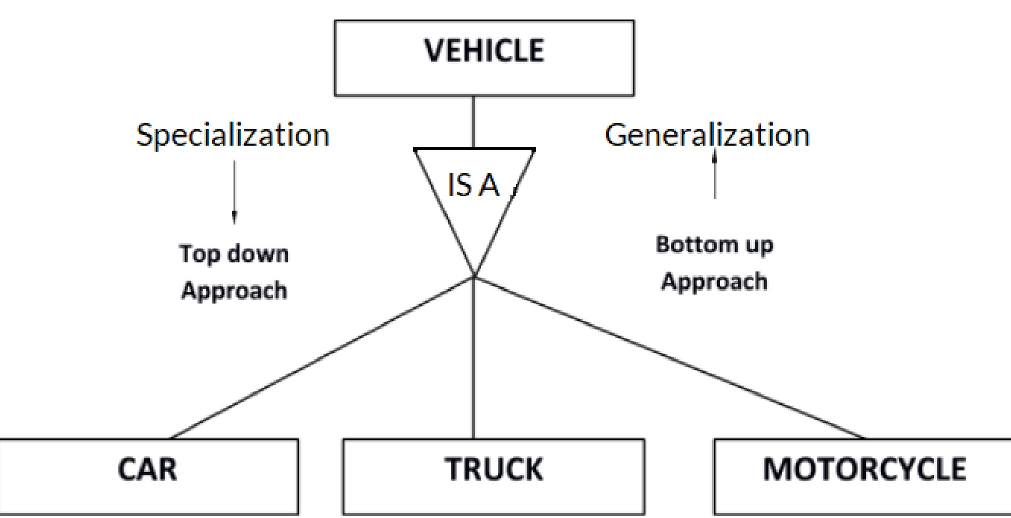
An **association relationship** can be represented as one-to-one, one-to-many, or many-to-many (also known as cardinality). Essentially, an **association relationship** between two or more objects denotes a path of communication (also called a link) between them so that one object can send a message to another

Association is a relationship between two objects. In other words, association defines the multiplicity between objects. You may be aware of one-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects

2.3.7.4 DEFINING SUPER-SUB RELATIONSHIPS

A subclass is a class derived from the superclass. It inherits the properties of the superclass and also contains attributes of its own. An example is:

Car, Truck and Motorcycle are all subclasses of the superclass Vehicle. They all inherit common attributes from vehicle such as speed, colour etc. while they have different attributes also i.e Number of wheels in Car is 4 while in Motorcycle is 2.



Car, Truck and Motorcycle are all subclasses of the superclass Vehicle. They all inherit common attributes from vehicle such as speed, colour etc. while they have different attributes also i.e Number of wheels in Car is 4 while in Motorcycle is 2.

Superclasses

A superclass is the class from which many subclasses can be created. The subclasses inherit the characteristics of a superclass. The superclass is also known as the parent class or base class.

In the above example, Vehicle is the Superclass and its subclasses are Car, Truck and Motorcycle.

2.3.7.5 IDENTIFYING THE AGGREGATION/a-PART OF RELATIONSHIP

In UML models, an aggregation relationship shows a classifier as a part of or subordinate to another classifier.

An aggregation is a special type of association in which objects are assembled or configured together to create a more complex object. An aggregation describes a group of objects and how you interact with them. Aggregation protects the integrity of an assembly of objects by defining a single point of control, called the aggregate, in the object that represents the assembly. Aggregation also uses the control object to decide how the assembled objects respond to changes or instructions that might affect the collection.

Data flows from the whole classifier, or aggregate, to the part. A part classifier can belong to more than one aggregate classifier and it can exist independently of the aggregate. For example, a Department class can have an aggregation relationship with a Company class, which indicates that the department is part of the company. Aggregations are closely related to compositions.

You can name an association to describe the nature of the relationship between two classifiers; however, names are unnecessary if you use association end names. As the following figure

illustrates, an aggregation association appears as a solid line with an unfilled diamond at the association end, which is connected to the classifier that represents the aggregate. Aggregation relationships do not have to be unidirectional.

Aggregation is referred as a "part–of" or "has–a" relationship, with the ability to navigate from the whole to its parts. An aggregate object is an object that is composed of one or more other objects.

.

Aggregation is a part of an association relationship.
Aggregation is considered as a weak type of association.
In an aggregation relationship, objects that are associated with each other can remain in the scope of a system without each other.
Linked objects are not dependent upon the other object.
In UML Aggregation, deleting one element does not affect another associated element.
Example: A car needs a wheel, but it doesn't always require the same wheel. A car can function adequately with another wheel as well.

2.4 CLASS RESPONSIBILITY

Class responsibilities are the **class's** attributes and methods. Clearly, they represent the **class's** state and behaviour. Collaborators represent the associations the **class** has with other **classes**. ...

Class-Responsibility-Collaborator cards

- Identify the **classes**.
- List **responsibilities**.
- List collaborators

Class responsibilities are the class's attributes and methods. Clearly, they represent the class's state and behaviour. Collaborators represent the associations the class has with other classes.

2.4.1 GUIDELINES FOR DEFINING ATTRIBUTES

an **attribute** is a specification that defines a property of an object, element, or file. It may also refer to or set the specific value for a given instance of such. For clarity, **attributes** should more correctly be considered metadata. An attribute describes a range of values for that data definition. A classifier can have any number of attributes or none at all. Attributes describe the structure and value of an instance of a class.

Entities contain **attributes** , which are characteristics or modifiers, qualities, amounts, or features. An **attribute** is a fact or nondecomposable piece of information about an entity. Later, when you represent an entity as a table, its **attributes** are added to the model as new columns.

To identify attributes:

· choose a class from the evolving object model and look for the properties associated with it,

· select a property candidate (e.g., from a problem statement or a requirements document) and look for the class that it describes.

Use the <u>Parse Requirements technique</u> to help identify attributes. They are likely adjectives, or nouns that describe a property of the class, such as date of car reservation, or enumerated lists of nouns and adjectives provided for illustration or detail. For example, in the expression “types of car rentals include day, weekend, and long term” the terms day, weekend, and long term may be attributes of a class CarRental.

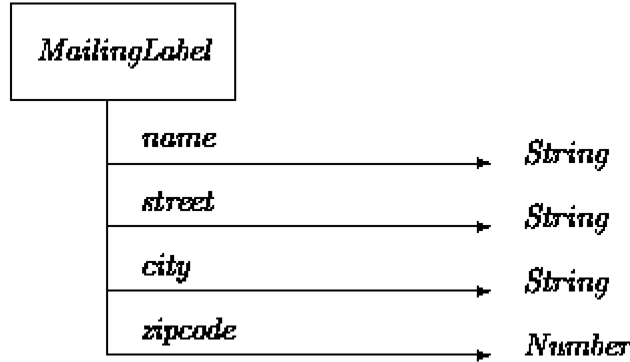
Because attributes are less often described in the problem statement or requirements documents, they are more difficult to discover than classes. Apply knowledge of the problem domain, and other analysis techniques for Data Gathering, such as, interviewing and workshops, to identify attributes.

Attributes in concrete classes must be defined via either *binding* or *computation*. The choice is made in *ODL* by either listing an attribute as `` <>", meaning that the value must be bound at construction, or defining it in `` {...}" brackets, meaning that it is computed.

Bound (or ``stored") attributes differ from computed ones in that they may be *rebound* (if non-fixed) and/or *unbound* (if opt). For simplicity and conformance to most implementation languages, we require that computationally defined attributes and operations not have their definitions rebound, unbound, or otherwise dynamically modified. The only way in which their values may change over time is by internally accessing properties of one or more mutable *objects*. The effects of rebinding may be had in this way, but the logistics are a bit harder.

Examples

Objects such as MailingLabels simply maintain several loosely related attributes. The classes consist of set/get interfaces, with a value reporter and a value replacer operation for each property listed in the analysis model:



2.5 OBJECT RESPONSIBILITY

The Object-Oriented Thought Process, is intended for someone just learning an object-oriented language and wants to understand the basic concepts before jumping into the code or someone who wants to understand the infrastructure behind an OOP language they are already using. Click here to start at the beginning of the series.

In keeping with the code examples used in the previous articles, Java will be the language used to implement the concepts in code. One of the reasons that I like to use Java is because you can download the Java compiler for personal use at the Sun Microsystems Web site http://java.sun.com/. You can download the J2SE 1.4.2 SDK (software development kit) to compile and execute these applications and I will provide the code listings for all examples in this article. I have the SDK 1.4.0 loaded on my machine. I will also provide figures and the output (when appropriate) for these examples. See the previous articles in this series for detailed descriptions for compiling and running all the code examples in this series.

In the last column, you explored the history and evolution of object-oriented languages—to provide a basic understanding of how object-oriented languages developed. In earlier columns, you explored a number of issues relating to the development of object-oriented systems and provided a basis for underlying fundamental object-oriented concepts. In this article, you will explore one of the fundamental characteristics that make for good class design. Remember that it is the class that provides the blueprint for all object-oriented design.

An Object Must be Responsible for Itself

One of the most important mantras used in object-oriented design is that of "*an object must be responsible for itself*". I first heard this phrase while taking my first class in object-oriented programming class in Smalltalk. Because I had never programmed in an object-oriented language before, I was curious as to what this really meant.

While there are many ways to approach this point, it really boils down to the concept of encapsulation, just as everything does when dealing with properly designed classes. The whole concept of an object is that it contains both attributes and behaviors. In structured programming, there is code and there is data, and the two are thought of as separate entities. The power of an object is that these two separate entities are encapsulated together to form a single, atomic entity. It is in this fundamental concept of an encapsulated, atomic entity that you come to realize how and why an object is responsible for itself.

A straightforward definition for object-responsibility is this: *An object must contain the data (attributes) and code (methods) necessary to perform any and all services that are required by the object.* This means that the object must have the capability to perform required services itself or at least know how to find and invoke these services. Rather than attempt to further refine the definition, take a look at an example that illustrates this responsibility concept

The UML defines a **responsibility** as "a contract or obligation of a classifier" [OMG01]. Responsibilities are related to the obligations of an object in terms of its behavior. Basically, these responsibilities are of the following two types:

- knowing
- doing

Doing responsibilities of an object include:

- doing something itself, such as creating an object or doing a calculation
- initiating action in other objects
- controlling and coordinating activities in other objects

Knowing responsibilities of an object include:

- knowing about private encapsulated data
- knowing about related objects
- knowing about things it can derive or calculate

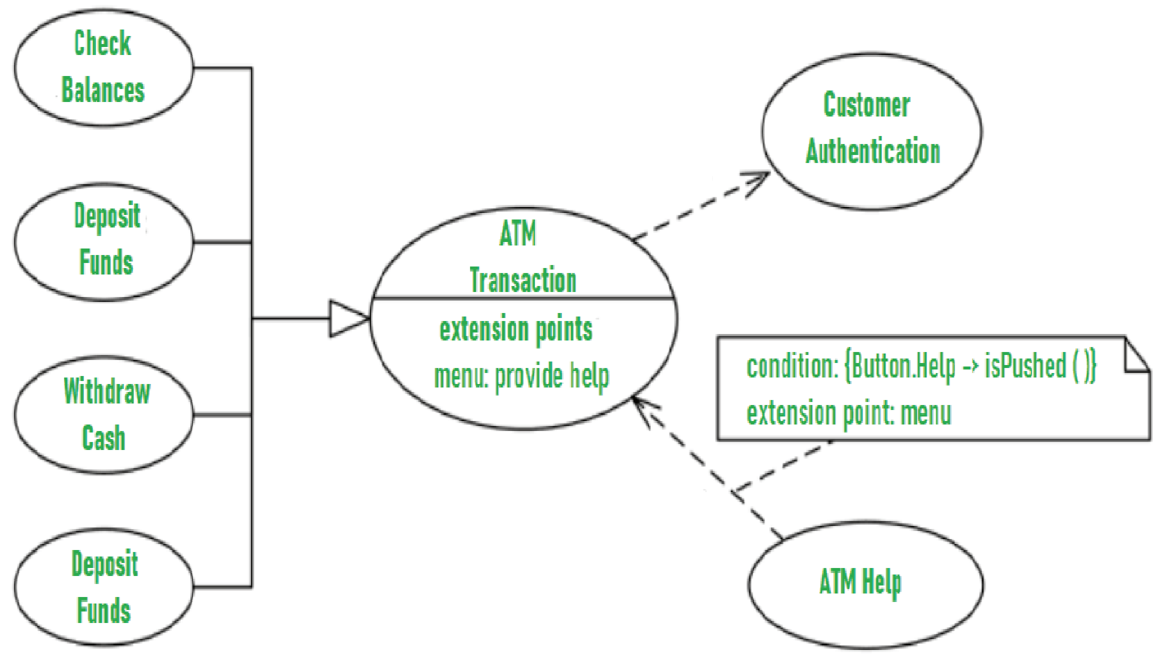
2.6 CASE STUDY

2.6.1 DEFINING ATTRIBUTES FOR VIA NET BANK OBJECTS

Automated Teller Machine (ATM) also known as ABM (Automated Banking Machine) is a banking system. This baking system allows customers or users to have access to financial transactions. These transactions can be done in public space without any need for a clerk, cashier, or bank teller. Working and description of the ATM can be explained with the help of the **Use Case Diagram**.

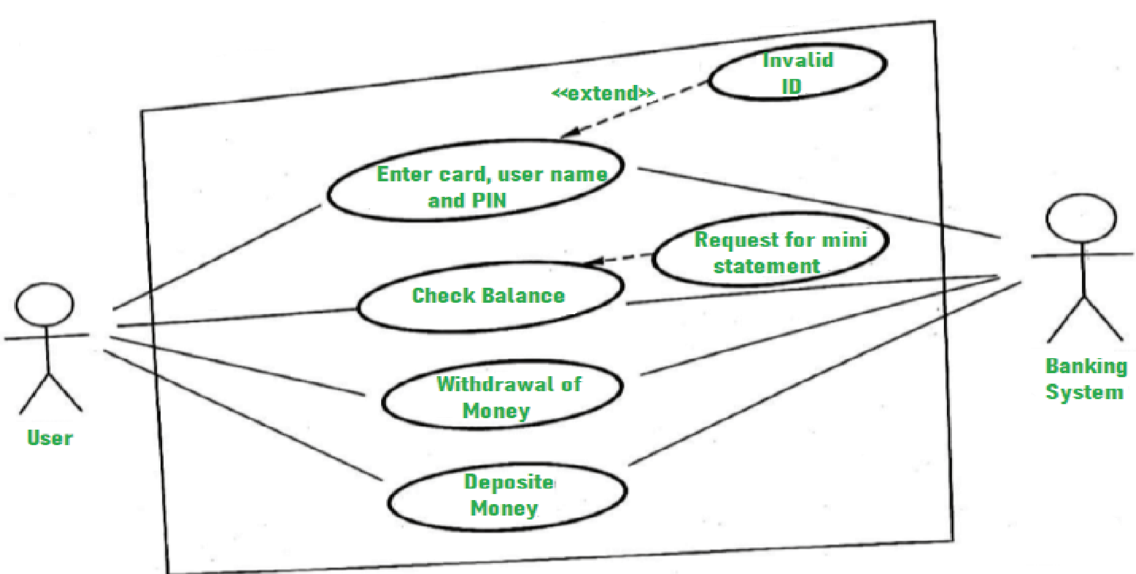
We will understand about designing the use case diagram for the ATM system. Some scenarios of the system are as follows.

- **Step-1:**
  The user is authenticated when enters the plastic ATM card in a Bank ATM. Then enters the user name and PIN (Personal Identification Number). For every ATM transaction, a Customer Authentication use case is required and essential. So, it is shown as include relationship.
  Example of use case diagram for Customer Authentication is shown below:



Use Case Diagram for Customer Authentication

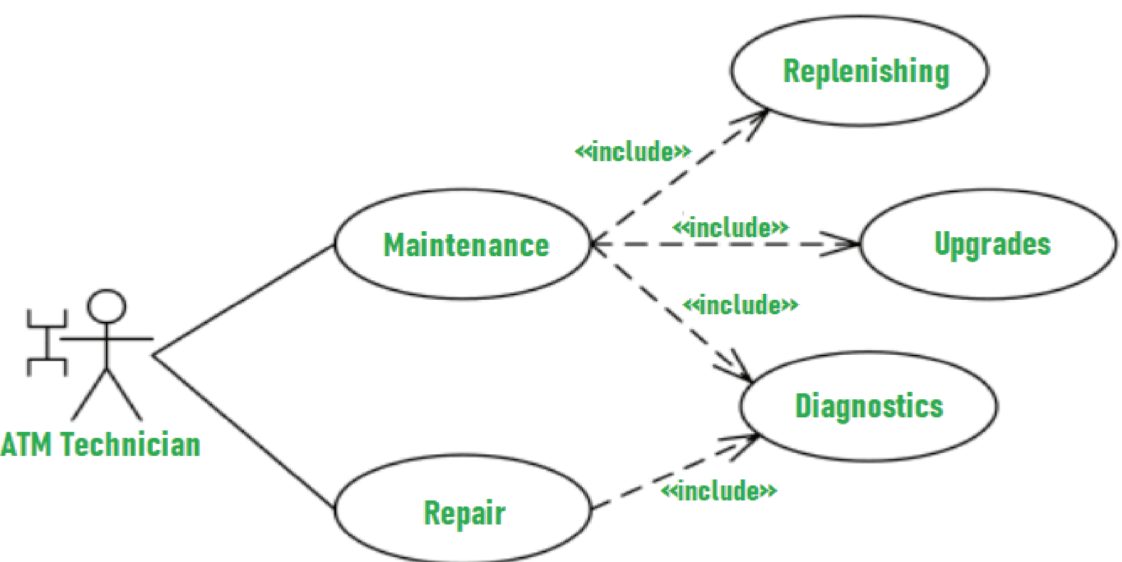
- **Step-2:**
  User checks the bank balance as well as also demands the mini statement about the bank balance if they want. Then the user withdraws the money as per their need. If they want to deposit some money, they can do it. After complete action, the user closes the session. Example of the use case diagram for Bank ATM system is shown below:



Use Case Diagram for Bank ATM System

- **Step-3:**
  If there is any error or repair needed in Bank ATM, it is done by an ATM technician. ATM technician is responsible for the maintenance of the Bank ATM, upgrades for hardware,

firmware or software, and on-site diagnosis. Example of use case diagram for working of ATM technician is shown below:



Use Case Diagram for Working of ATM Technician

Attention reader! Don't stop learning now. Get hold of all the important CS Theory concepts for SDE interviews with the **CS Theory Course** at a student-friendly price and become industry ready

2.6.1.1 DEFINING ATTRIBUTES FOR ACCOUNT AND TRANSCATION CLASS

The **accounts attribute** contains a list of objects for each **account** linked to the Identity Manager user. Each **account** object contains the values of the **account attributes** retrieved from the resource. The name of each **account** object is typically the name of the associated resource.

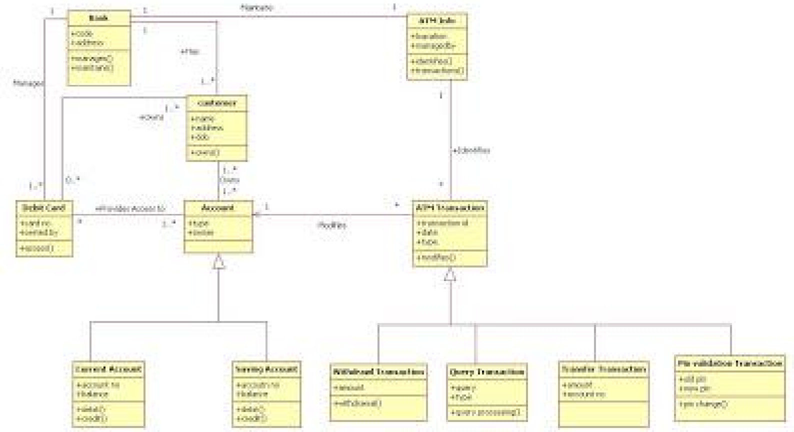
Account and Transaction provide the banking model. Note that Transaction assumes an active role while money is being dispensed and a passive role thereafter. The class Account is responsible mostly to the BankClient class and it collaborates with several objects to fulfill its responsibilities. Among the responsibilities of the Account class to the BankClient class is to keep track of the BankClient balance, account number, and other data that need to be remembered. These are the attributes of the Account class. Furthermore, the Account class provides certain services or methods, such as means for BankClient to deposit or withdraw an amount and display the account's Balance (see Figure). Classes, Responsibilities, and Collaborators encourages team members to pick up the card and assume a role while "executing" a scenario. It is not unusual to see a designer with a card in each hand, waving them about, making a strong identification with the objects while describing their collaboration. Ward Cunningham writes: Classes, Responsibilities,andCollaboratorscards work by taking people through programmingepisodestogether.

2.6.1.2 DEFINING ATTRIBUTES FOR ATM MACHINE CLASS

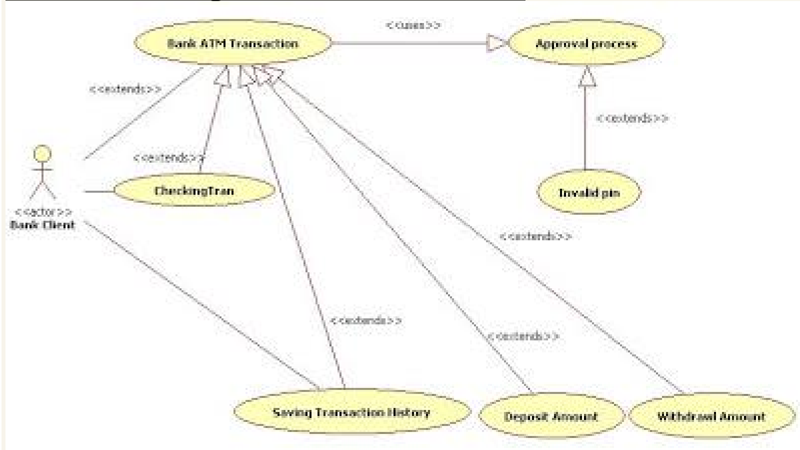
The ATM is given the utmost security in terms of technology because its a stand alone system and easily prone to malicious attacks.

The Below diagrams are has come up in Mumbai University MCA exams..

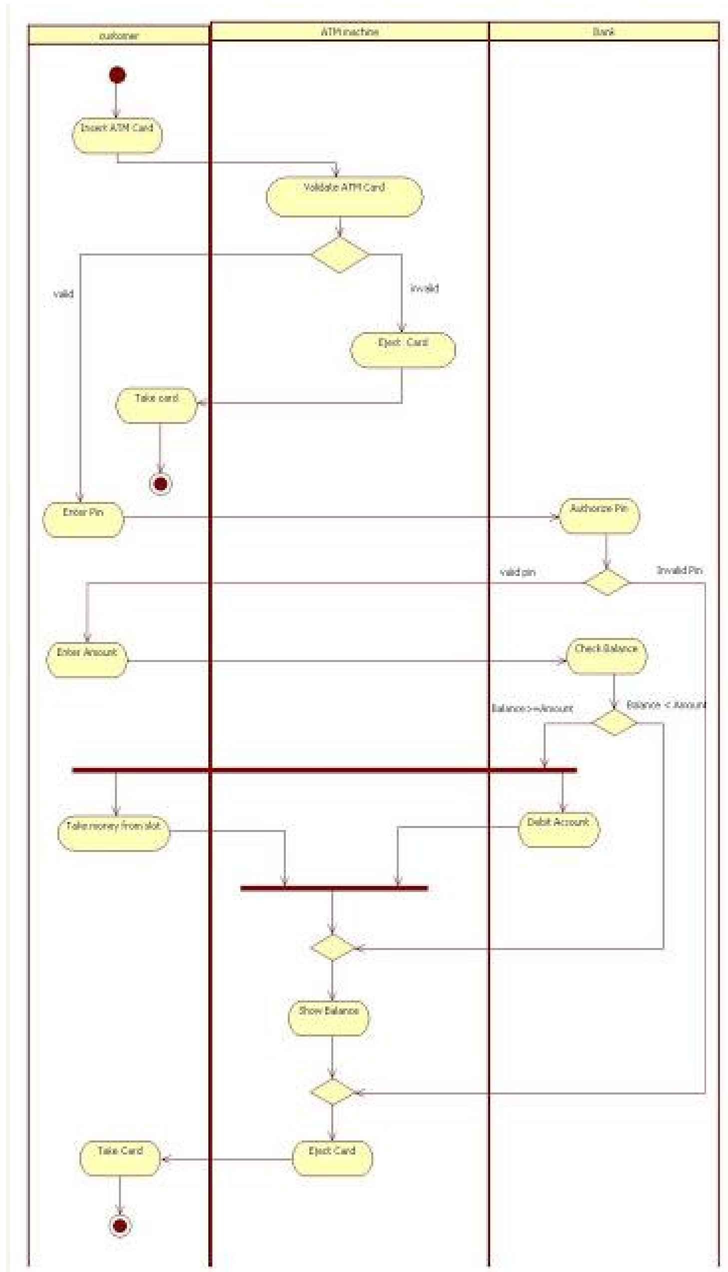
Class Diagram for ATM Machine



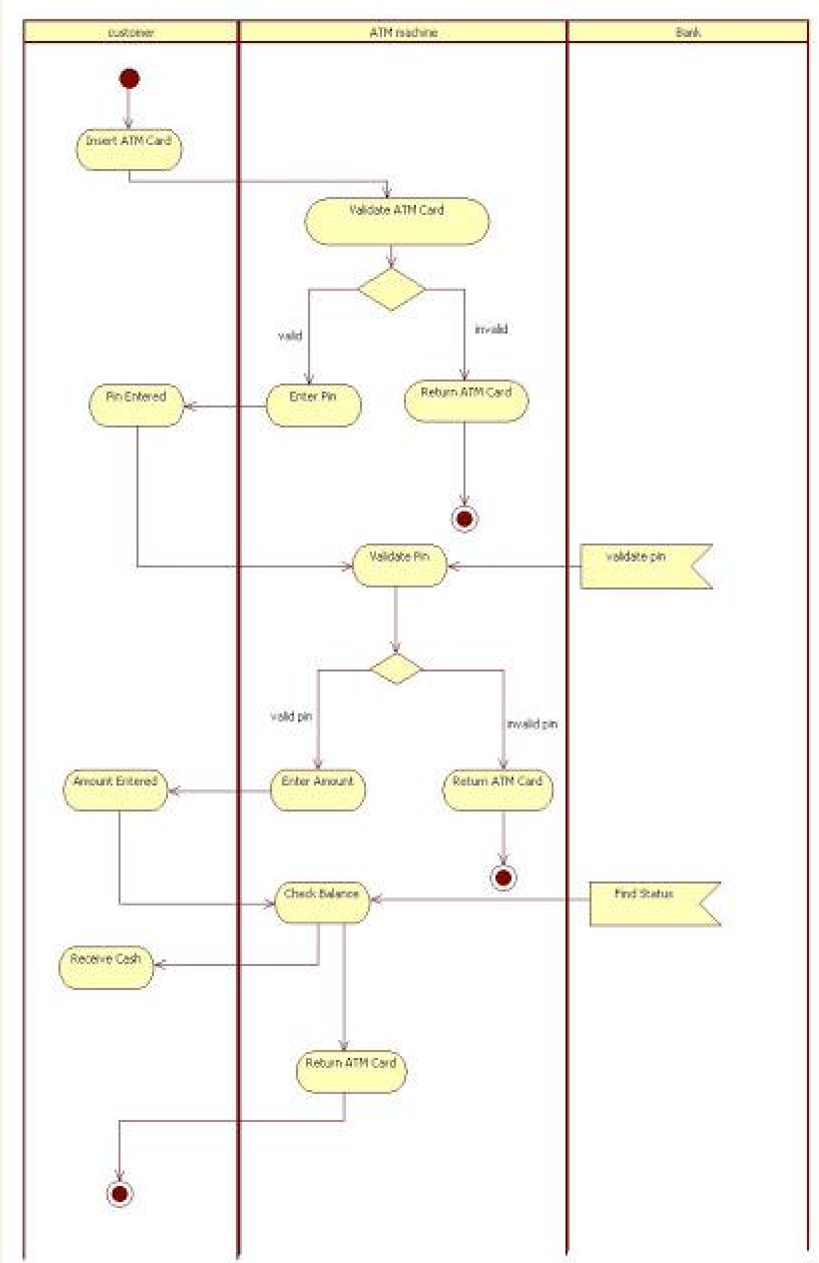
Use Case Diagram ATM Machine



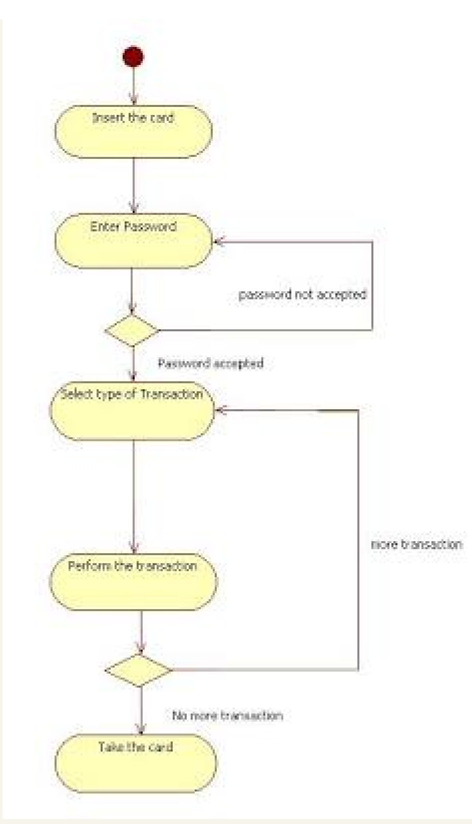
Activity Diagram for ATM Machine 1



Activity Diagram for ATM Machine 2



<u>Activity Diagram for Overall ATM Machine:-</u>


Insert the card
Enter Password
password not accepted
Password accepted
Select type of Transaction
more transaction
Perform the transaction
No more transaction
Take the card

UNIT III- OBJECT ORIENTED DESIGN

3.1 DESIGN PROCESS

3.1.1 INTRODUCTION

Object Oriented Design (OOD) serves as part of the object oriented programming (OOP) process of lifestyle. ... This technique enables the implementation of a software based on the concepts of objects. Additionally, it is a concept that forces programmers to plan out their code in order to have a better flowing program

the conceptual model is developed further into an object-oriented model using object-oriented design (OOD). In OOD, the technology-independent concepts in the analysis model are mapped onto implementing classes, constraints are identified, and interfaces are designed, resulting in a model for the solution domain. In a nutshell, a detailed description is constructed specifying how the system is to be built on concrete technologies

The stages for object–oriented design can be identified as −

Definition of the context of the system

Designing system architecture

Identification of the objects in the system

Construction of design models

Specification of object interfaces

3.1.2 THE OBJECT ORIENTED DESIGN PROCESS

Process of Object Oriented Design:

Understanding the process of any type of software related activity simplifies its development for the software developer, programmer and tester. Whether you are executing functional testing, or making a test report, each and every action has a process that needs to be followed by the members of the team. Similarly, Object Oriented Design (OOD) too has a defined process, which if not followed rigorously, can affect the performance as well as the quality of the software. Therefore, to assist the team of software developers and programmers, here is the process of Object Oriented Design (OOD):

1. To design classes and their attributes, methods, associations, structure, and even protocol, design axiom is applied.

- The static UML class diagram is redefined and completed by adding details.
- Attributes are refined.
- Protocols and methods are designed by utilizing a UML activity diagram to represent the methods algorithm.
- If required, redefine associations between classes, and refine class hierarchy and design with inheritance.
- Iterate and refine again.

2. Design the access layer.

o Create mirror classes i.e., for every business class identified and created, create one access class.

3. Identify access layer class relationship.

4. Simplify classes and their relationships. The main objective here is to eliminate redundant classes and structures.

o **Redundant Classes:** Programmers should remember to not put two classes that perform similar translate requests and translate results activities. They should simply select one and eliminate the other.

o **Method Classes:** Revisit the classes that consist of only one or two methods, to see if they can be eliminated or combined with the existing classes.

5. Iterate and refine again.

6. Design the view layer classes.

o Design the macro level user interface, while identifying the view layer objects.

o Design the micro level user interface.

o Test usability and user satisfaction.

o Iterate and refine.

7. At the end of the process, iterate the whole design. Re-apply the design axioms, and if re-quired repeat the preceding steps again

3.2 DESIGN AXIOMS

The **design axioms** give designers, engineers, product managers, and anyone else who influences the creation of software **design**, a simple-but-essential core set of rules for building effective interfaces

3.2.1 OBJECT ORIENTED DESIGN AXIOMS

Axiom:

An axiom is a fundamental truth that always is observed to be valid and for which there is no counterexample or exception. The axioms cannot be proven or derived but they cannot be invalidated by counterexamples or exceptions. There are two design axioms applied to object-oriented design. Axiom 1 deals with relationships between system components and Axiom 2 deals with the complexity of design.

Axiom 1: The independence axiom. Maintain the independence of components.

Axiom 2: The information axiom. Minimize the information content of the design.

Axiom 1 States that, during the design process, as we go from requirement and use - case to a system component, each component must satisfy that requirement, without affecting other requirements Axiom 2 • Concerned with simplicity. Rely on a general rule known as Occam‟s razor. •Occam‟s razor rule of simplicity in OO terms : The best designs usually involve the

least complex code but not necessarily the fewest number of classes or methods. Minimizing complexity should be the goal, because that produces the most easily maintained and enhanced application. In an object-oriented system, the best way to minimize complexity is to use inheritance and the system‟s built-in classes and to add as little as possible to what already is there.

3.2.2 COROLLARIES

Corollaries May be called Design rules , and all are derived from the two basic axioms :The origin of collaries as shown in figure 2. Corollaries 1,2 and 3 are from both axioms, whereas corollary 4 is from axiom 1 and corollaries 5 & 6 are from axiom 2

Origin of corollaries

Corollary 1:

Uncoupled design with less information content.

 Coupling is a measure of the strength of association established by a connection from

one object or software component to another.

 Coupling is a binary relationship.

 It is important for design because a change in one component should have a minimal

impact on the other components.

 Degree or strength of coupling between two components is measured by the amount &

complexity of information transmitted between them.

 OO design has 2 types of coupling :

o Interaction coupling and Inheritance coupling.

Interaction coupling

 The amount & complexity of messages between components.

 Desirable to have a little interaction.

 Minimize the number of messages sent & received by an object

 Types of coupling among objects or components

Inheritance coupling

coupling between super-and subclasses

 A subclass is coupled to its superclass in terms of attributes & methods

 High inheritance coupling is desirable

 Each specialization class should not inherit lots of unrelated & unneeded

methods & attributes

 Need to consider interaction within a single object or sw component► Cohesion

o Cohesion ► reflects the‟single-purposeness‟ of an object (see corollaries 2&3)

o Method cohesion ► a method should carry only one function.

o A method carries multiple functions is undesirable

Corollary 2 : Single purpose .

 Each class must have a purpose & clearly defined

 Each method must provide only one service

Corollary 3 : Large number of simple classes.

 Keeping the classes simple allows reusability

 A class that easily can be understood and reused (or inherited) contributes to the overall system

 Complex & poorly designed class usually cannot be reused.

 Guideline ► The smaller are your classes, the better are your chances of reusing them inother projects. Large & complex classes are too specialized to be reused.

 The emphasis OOD places on encapsulation, modularization, and polymorphism suggests reuse rather than building anew

 Primary benefit of sw reusability ► Higher prpductivity

Corollary 4 : Strong mapping.

 There must be a strong association between the analysis's object and design's object.

 OOA and OOD are based on the same model.

 As the model progresses from analysis to implementation, more detailed is added.

Corollary 5 : Standardization.

 Promote standardization by designing interchangeable components and reusing existing classes or components

 The concept of design patterns might provide a way to capture the design knowledge, document it, and store it in a repository that can be shared and reused in different applications

Corollary 6 : Design with inheritance.

 Common behavior (methods) must be moved to super classes.

 The superclass-subclass structure must make logical sense.

3.3 CLASSES DESIGN

3.3.1 INTRODUCTION

In general class design is a process to add details to the class

diagrams defined in the analysis phase and making fine decisions

You choose how to implement your classes considering

Minimization of execution time, memory and other cost measures

Choice of algorithms implementing methods

Braking complex operation ito simpler operations

OO design is an iterative process

When one level of abstraction is complete you should design the nextlower level of abstraction

For each level you may

add new operations, attributes, and classes

Revise relations between classes

The Essence of Design is…

6 System modelling – Fabrizio Maria Maggi

To build a bridge across the gap between:

Desired features

Use cases

Application commands

System operations

System services

Available resources

Operating system infrastructure

Class libraries

Previous applications

3.3.2 THE OBJECT ORIENTED DESIGN PHILOSOPHY

As a part of an overall strategy of agile and adaptive programming, a number of object-oriented design principles were proposed for the design and programming of computer software system that is easy to maintain and extend over time. These principles are guidelines intended for programmers to apply while working on software to remove "code smells" (potentially buggy code) by refactorizing the source code until it is both legible and extensible. In this page, we introduce the **SOLID** principles, that is, Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion. The following information was integrated from various sources on the Web.

Single Responsibility Principle (SRP)

The SRP requires that a class should have only a single responsibility.

Example: If a class SalesOrder keeps information about a sales order, and in addition has a method saveOrder() that saves the SaleOrder in a database and a method exportXML() that exports the SalesOrder in XML format, this design will violate the SRP because there will be different types of users of this class and different reasons for making changes to this class. A change made for one type of user, say change the type of database, may require the re-test, recompilation, and re-linking of the class for the other type of users.

A better design will be to have the SalesOrder class only keeps the information about a sales order, and have different classes to save order and to export order, respectively. Such a design will confirm to SRP.

Open-Closed Principle (OCP)

The OCP requires that each software entity should be open for extension, but closed for modification.

Example: Suppose an OrderValidation class has a method validate(Order order) that is programmed to validate an order based on a set of hard-coded rules. This design violates the OCP because if the rules change, the OrderValidation class has to be modified, tested, and compiled.

A better design will be to let the OrderValidation class contain a collection of ValidationRule objects each of which has a validate(Order order) method (perhaps defined in a Validation interface) to validate an Order using a specific rule, and the validate(Order order) method of OrderValidation class can simply iterate through those ValidationRule objects to validate the order. The new design will satisfy the OCP, because if the rules change, we can just create a new ValidationRule object and add it to an OrderValidation instance at run time (rather than to the class definition itself).

This is can also be achieved by using subclasses of a base class AbstractValidationRule that has an override-able function validate(Order order). Subclasses can implement the method differently without changing the base class functionality.

Liskov Substitution Principle (LSP)

The LSP requires that objects in a program should be replaceable with instances of their subclasses without altering the correctness of that program.

The users must be able to use objects of subclasses via references to base classes without noticing any difference. When using an object through its base class interface, the object of a subclass must not expect the user to obey preconditions that are stronger than those required by the base class.

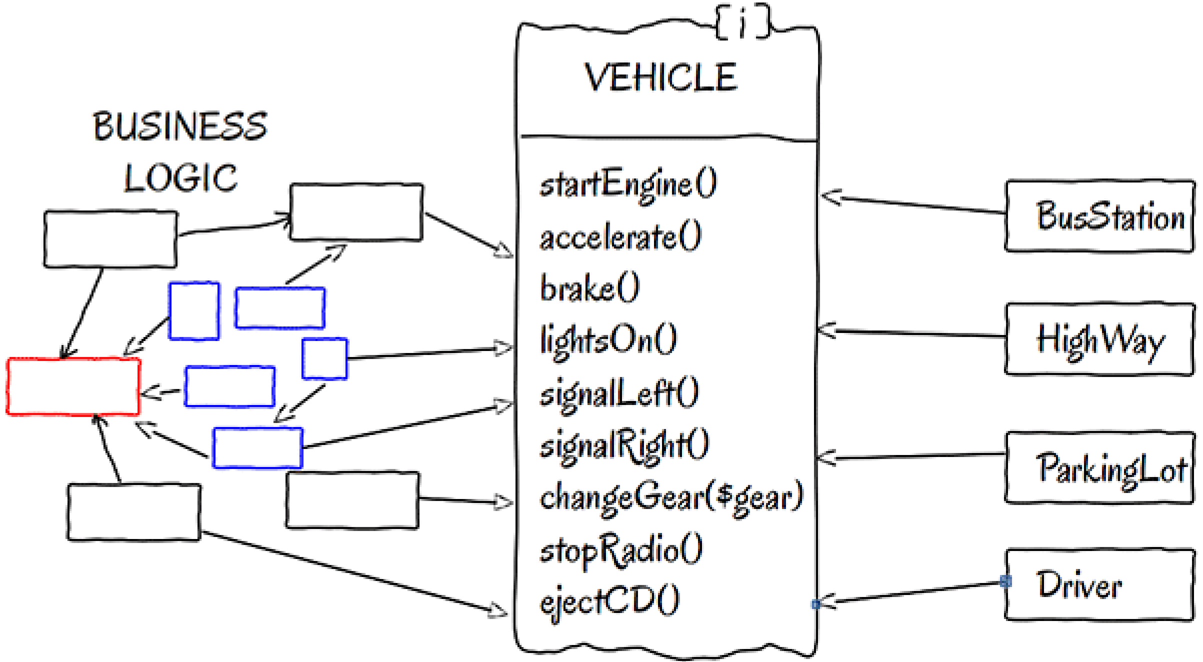
Example: Suppose a Rectangle class has two instance variables height and width, and a method setSize(int a, int b), which set height to a and width to b. Suppose Square is a subclass of Rectangle and it overrides the inherited method by setting both height and width to a. This design will violate LSP. To see this, consider a client uses a reference variable of type Rectangle to call the setSize() method to assign different values of a and b, and then immediately verify if the sizes were set correctly or the area is correctly computed. The results will be different if the variable references to a Rectangle object than to a Square object.

It turns out that in OO programming, a Square is not a Rectangle at all because it behaves differently from a Rectangle.

Interface Segregation Principle (ISP)

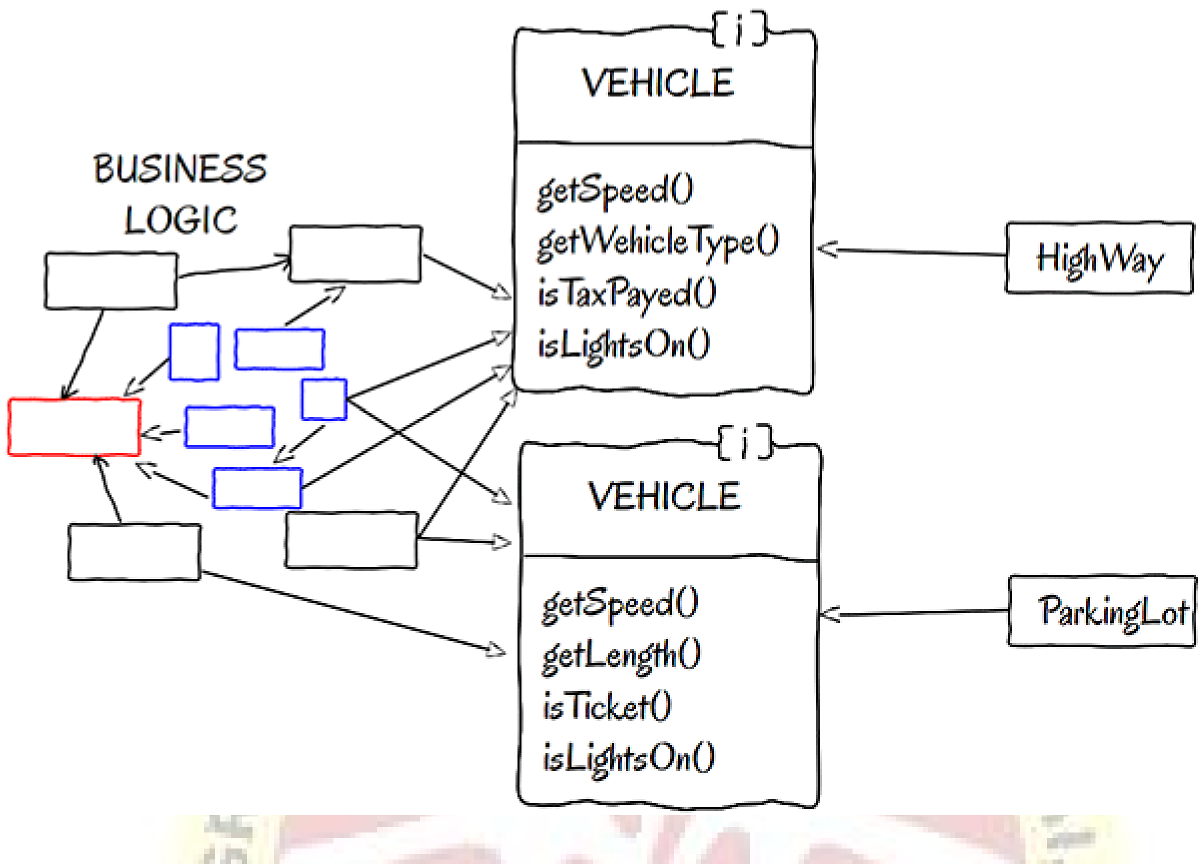
The ISP requires that clients should not be forced to depend on interfaces that they do not use.

Example: Suppose a Vehicle interface shown in the figure is designed for clients to use



This violates ISP because clients are forced to depend on methods they do not use: HighWay does not use stopRadio() or ejectCD(), and ParkingLot does not need accelerate() or ejectCD().

A better design is to design smaller interfaces for different types of clients as shown in the following figure



Dependency Inversion Principle (DIP)

The DIP requires that high level modules should not depend on low level modules, both should depend on abstraction. Also, abstraction should not depend on details, details should depend on abstractions.

Example: Making a class Button associate to another class Lamp (because a Lamp has a Button) is a violation of DIP. A better design will be associate an AbstractButton with an AbstractButtonClient, and define Button as a subclass of the AbstractButton and a Lamp a subclass of the AbstractButtonClient.

3.3.3 DESIGNING CLASSES: THE PROCESS

1. Designing Classes1. Designing classes2. Designing protocols and class visibility3. The UML object Constraint language4. Designing methods

2. Introduction• Designer has to know – Specification of the class – interaction of that class with other classes

3. Object oriented design philosophy• Designing class with reusability in mind -> more gain in productivity and reduce the time for developing new application

4. UML Object Constraint Language(OCL)• The rules and semantics of UML are expressed using OCL• OCL – Specification language – Uses simple logic to specify the properties of the system• UML modeling constructs requires expression. some example as follow – Item.selector • Item -> object • Selector -> attribute • Ex : kathir.regno – Item.selextor[qualifier-value] •

Qualifier is uesd to select related values • Ex: – Kathir.phone[2] – Set->select (Boolean expression) • Company.employee->salary>20000

5. Designing classes : process• Apply design axioms to design calsses , their atribute, methods and association , structures, amd protocols – Refine and complete the static uml class diagrams by adding details to that design • Refine attribute • Design methods and protocols – Uses uml activity diagram to represent algorithm • Refine the association bw classes • Refine class hierarchy and design with inheritance – Iterate and refine

6. Class visibility: desiging well defined public , private and protected protocols• 2 problems in designing methods or attributes of class – Protocol or interface to the class operation and its visibility – How it is implemented• Protocol layers – Private protocol – Protected protocol – Public protocol

7. Internal layer• Defines the implementation of the object• Apply axioms and corollaries (corollary 1) to decide what to private• Private protocol – Includes messages that should not be sent from other objects – Accessible to only operations of that class• Protected protocol – Methods and attributes can be used by class itself or its subclass

8. External layer• Design the functionality of the object• Public protocol – Defined to pass message bw associated classes – Interpretation and implementation of each message is up to the individual classes

9. Encapsulation leakage• lack of well designed protocol leads to this problem• Encapsulation leakage occurs when details about classes internal implementation are disclosed through the interface

10. Refining attribute• Attribute – Represents the state of the object• Analysis phase -> name of the attribute is sufficient• Design phase -> detailed information must be added• Attribute types – Single valued attribute • Ex: dob of the student – Multiplicity or multivalue attribute • Subjects handled by the staff – Reference to another object or instance connection • Person hold the account

11. UML attribute presentation• OCL is used in the design phase to define the class attributes• Attribute presentation – Visibility name : type-expression = initial-value – Visibility • Public visibility : + • Protected visibility : # • Private visibility : - – Type expression • Language dependent specification of the type – Initial value • Language dependent expression for the initial value

12. Designing methods and protocols• Specifying the algorithm for methods – By using formal structure (ex uml activity diag) with OCL• Types of methods – Constructor – Destructor – Conversion method – Copy method – Attribute set – Attribute get – IO methods – Domain specific

13. Design issues : avoiding pitfalls• Better to have Large set of simple classes than few large , complex classes – Initially class might be too big – Apply design axioms and corollary • to reduce the size • Improve reusability

3.3.4 DESIGN ISSUES

Using object-oriented methodologies, discuss design solutions for the following problem. Describe what classes you would need and define the relationships between these classes

create both “is a” and “has a” relationships. State any assumptions you make concerning functional requirements.

1. Flight Simulator

Design a flight simulator. Consider what must be displayed to have a view from the cockpit of a small airplane, periodically updating this view to reflect motion. The world in which flights take place must show mountains, rivers, lakes, roads, bridges, a radio tower and of course a runway. Control inputs are form two joysticks. The left joystick operators the rudder and engine. The right one controls ailerons and elevator. Make the simulator as realistic as possible without making it too complex.

2. Make Simulator

Prepare a design for a system that automatically executes actions needed to build a software system from its components, similar to the UNIX Make facility. The system reads a file which describes what must be done in the form of dependency rules. Each rule has one or more targets, one or more sources, and an optional action. Targets and sources are names of files. If any of the sources of a rule are newer than any of its targets, the action of the rule is executed by the system to rebuild the targets from the sources.

3. Tic Tac Toe Player

Design a system that plays tic-tac-toe. Inputs and outputs are provided through a dedicated hardware interface. The user indicates moves by pressing membrane switches, one for each of the nine squares. X’s and O’s are displayed by a liquid crystal display. The user may select a level of skill and who is to go first.

4. Pilot Training

Design a system that would allow for training of a glider pilot. The simulator is for one glider with wings and rudder only. Effects of the wind and forces generated by the body of the

glider must be considered. Details of the user interface to the simulator are should also be part of this design.

Gliders have several associated lifting surfaces, in this case two wings and a rudder. The wings provide lift and the rudder is used to steer. Methods would be provided to perform simulation – for example, the force on each surface would be calculated from its attribute values and the orientation, velocity and rotational rate of the glider. The force on the rudder would also depend on its deflection. The translational acceleration would be computed by retrieving the results of force calculations and masses of associated surfaces. The accelerations would be numerically integrated to update position, orientation, velocity, and rotational rate.

5. Integrated Circuits

Design a system for a portable tester for integrated circuits. The tester will have several different types of sockets. An integrated circuit will be tested by placing it in the socket that matches its pin configuration and identifying the type of circuit. The tester will then run through a series of tests, applying power and signals to the appropriate pins and measuring the response of the circuit. The same signal can be applied to more than one pin. Each pin may receive signals from several test cases.

6. Parser

Design a parser for the C programming language that has data types, operators, conditional expressions, loops, and functions. Take this design and apply it the next step to C++'s class constructs, inheritance, and if possible add to this dynamic binding.

7. Airline Reservation System

Design an airline reservation system. There could be a ternary relationship between the flight, seats, and passengers. Seats are assigned zero or one passengers. A passenger may travel on many flights but must have exactly one seat on a traveled flight and must be sitting in it

during the flight. How would you change your design if you allowed passengers to reserve (and pay for) multiple seats (maybe they are want that extra elbow room?).

8. Automated Home

Design a system that would allow our homes to be automated by a central processor that controls lights, heat, oven, alarm, coffee maker, vcr, etc. (use your imagination). Allow users to access the controls directly from a keypad or remotely from the phone (given a passcode), or from a wireless remote control unit.

9. Drilling!

Design a system that automates drilling of holes in rectangular metal plates. The size and location of the holes is described interactively using a graphical editor. When the user is satisfied with a particular drawing, a peripheral device on the personal computer punches a numerical control tape which can be used by commercially available drilling machines which have moving drill heads and that can change drill sizes. You are concerned only with the editing of the drawings and the punching of the tapes. The tapes contain a sequence of instructions to move the drill head, change drills, and drill. Since it takes some time to move the drill between holes, and even longer to change drills, the system should determine a reasonably efficient drilling sequence. It is not necessary to achieve the absolute minimum time, the system should not be grossly inefficient either. The drill head is controlled independently in the x and y directions, so the time it takes to move between holes is proportional to the larger of the required displacements in the x and the y directions.

3.3.5 UML OPERATION PRESENTATION

The Unified Modeling Language (UML) is a graphical language for OOAD that gives a standard way to write a software system's blueprint. It helps to visualize, specify, construct, and document the artifacts of an object-oriented system. It is used to depict the structures and the relationships in a complex system.

Brief History

It was developed in 1990s as an amalgamation of several techniques, prominently OOAD technique by Grady Booch, OMT (Object Modeling Technique) by James Rumbaugh, and OOSE (Object Oriented Software Engineering) by Ivar Jacobson. UML attempted to standardize semantic models, syntactic notations, and diagrams of OOAD.

Systems and Models in UML

System − A set of elements organized to achieve certain objectives form a system. Systems are often divided into subsystems and described by a set of models.

Model − Model is a simplified, complete, and consistent abstraction of a system, created for better understanding of the system.

View − A view is a projection of a system's model from a specific perspective.

Conceptual Model of UML

The Conceptual Model of UML encompasses three major elements −

- Basic building blocks
- Rules
- Common mechanisms

Basic Building Blocks

The three building blocks of UML are −

- Things
- Relationships
- Diagrams

Things

There are four kinds of things in UML, namely −

- **Structural Things** − These are the nouns of the UML models representing the static elements that may be either physical or conceptual. The structural things are class, interface, collaboration, use case, active class, components, and nodes.
- **Behavioral Things** − These are the verbs of the UML models representing the dynamic behavior over time and space. The two types of behavioral things are interaction and state machine.
- **Grouping Things** − They comprise the organizational parts of the UML models. There is only one kind of grouping thing, i.e., package.
- **Annotational Things** − These are the explanations in the UML models representing the comments applied to describe elements.

Relationships

Relationships are the connection between things. The four types of relationships that can be represented in UML are −

- **Dependency** − This is a semantic relationship between two things such that a change in one thing brings a change in the other. The former is the independent thing, while the latter is the dependent thing.
- **Association** − This is a structural relationship that represents a group of links having common structure and common behavior.

- **Generalization** − This represents a generalization/specialization relationship in which sub-classes inherit structure and behavior from super-classes.
- **Realization** − This is a semantic relationship between two or more classifiers such that one classifier lays down a contract that the other classifiers ensure to abide by.

Diagrams

A diagram is a graphical representation of a system. It comprises of a group of elements generally in the form of a graph. UML includes nine diagrams in all, namely −

- Class Diagram
- Object Diagram
- Use Case Diagram
- Sequence Diagram
- Collaboration Diagram
- State Chart Diagram
- Activity Diagram
- Component Diagram
- Deployment Diagram

Rules

UML has a number of rules so that the models are semantically self-consistent and related to other models in the system harmoniously. UML has semantic rules for the following −

- Names
- Scope
- Visibility
- Integrity
- Execution

Common Mechanisms

UML has four common mechanisms −

- Specifications
- Adornments
- Common Divisions
- Extensibility Mechanisms

Specifications

In UML, behind each graphical notation, there is a textual statement denoting the syntax and semantics. These are the specifications. The specifications provide a semantic backplane that contains all the parts of a system and the relationship among the different paths.

Adornments

Each element in UML has a unique graphical notation. Besides, there are notations to represent the important aspects of an element like name, scope, visibility, etc.

Common Divisions

Object-oriented systems can be divided in many ways. The two common ways of division are −

- **Division of classes and objects** − A class is an abstraction of a group of similar objects. An object is the concrete instance that has actual existence in the system.
- **Division of Interface and Implementation** − An interface defines the rules for interaction. Implementation is the concrete realization of the rules defined in the interface.

Extensibility Mechanisms

UML is an open-ended language. It is possible to extend the capabilities of UML in a controlled manner to suit the requirements of a system. The extensibility mechanisms are −

- **Stereotypes** − It extends the vocabulary of the UML, through which new building blocks can be created out of existing ones.
- **Tagged Values** − It extends the properties of UML building blocks.
- **Constraints** − It extends the semantics of UML building blocks.

3.4 OBJECT STORAGE AND OBJECT INTEROPERABILITY

3.4.1 INTRODUCTION

Interoperability quality enables a software system to work with other software systems [13]. Interoperability of things and services in a WoT system is important to facilitate their composition and provisioning. Interoperability is also critical from infrastructure perspective that is used to host the services in a WoT system. For example, if heterogeneous IaaS cloud infrastructures are used to host the services, their capability to host the IoT subsystems constituting a WoT system is important so that an appropriate IaaS cloud, which matches the security and reliability constraints can be chosen. A number of architecture tactics can be adopted in the IoT subsystems architectures and the corresponding WoT system architecture to support interoperability. For example, proxies and services' facades can hide the internal details of how the subsystems are deployed and migrated among IaaS clouds during their life-cycle [14]. Autonomous conversion of information associated with an IoT service semantics into service syntax can facilitate services' interoperability [15]. Adopting a layered architecture approach can be useful to compartmentalise the IoT subsystems' services and to provide

interoperability among the services belonging to similar layers of the federated clouds [16]. A strategy to delegate tasks to the optimal services configuration and dynamically allocating hosting IaaS cloud resources can facilitate the process of achieving interoperability

Interoperability is the ability of equipment and systems from different vendors to operate together. Interoperability is a must as smart objects emerge as a large-scale technology. Interoperability is essential both between smart objects from different manufacturers and between smart objects and existing infrastructures.

For smart objects, interoperability is as multifaceted as standardization. Smart objects must interoperate from the physical layer up to the application or integration layer. Physical layer interoperability occurs when equipment from different vendors physically communicates with each other. At the physical level, smart objects must agree on matters such as the physical frequencies at which communication takes place, what type of modulation the physical signals should carry, and the rate at which information is transferred. At the network level, nodes must agree on the format of the information that is sent and received over the physical channel and how nodes are addressed, as well as how messages should be transported through a network of smart objects. At the application or integration level, smart objects must share a common view on how data should be entered or extracted from a smart object network, as well as how the smart objects should be reached from outside systems.

The challenges of interoperability are in the technical definition of smart objects as well as the standardization and implementation and testing processes. To achieve interoperability, it is imperative that the technical architecture of smart objects is defined to ease interoperability. If the architecture either disallows interoperability or makes interoperability cumbersome, it is very difficult to achieve interoperability later. Likewise, the standardization process must make interoperability a primary concern. To do this, smart object standards cannot be tied to any particular hardware or communication technology. After standardization is complete, a testing or certification procedure helps to achieve and ensure interoperability between different devices and vendors.

As with standardization, interoperability poses several challenges for smart objects. First, the technical architecture for smart objects is still an open issue. In this book, we choose one such architecture for smart objects: the IP architecture. Second, although some of the standards for smart objects are still under development, those standards that already exist can be reused. We return to this ongoing standardization process in Part II. Third, interoperability test suites and conformance tests are still an open issue. Ideally, such interoperability test suites should test many levels of interoperability such as physical, networking, and application levels.

3.4.2 DATABASE MANAGEMENT SYSTEM

An object-oriented database management system (OODBMS) is a database management system that supports the creation and modeling of data as objects. OODBMS also includes support for classes of objects and the inheritance of class properties, and incorporates methods, subclasses and their objects. Most of the object databases also offer some kind of query language, permitting objects to be found through a declarative programming approach.

An object-oriented database management system represents information in the form of objects as used in object-oriented programming. OODBMS allows object-oriented programmers to develop products, store them as objects and replicate or modify existing objects to produce new

ones within OODBMS. OODBMS allows programmers to enjoy the consistency that comes with one programming environment because the database is integrated with the programming language and uses the same representation model. Certain object-oriented databases are designed to work with object-oriented programming languages such as Delphi, Python, Java, Perl, Objective C and Visual Basic .NET.

An object-oriented database management system (OODBMS), sometimes shortened to ODBMS for *object database management system*, is a database management system (DBMS) that supports the modelling and creation of data as objects. This includes some kind of support for classes of objects and the inheritance of class properties and methods by subclasses and their objects.

There is no widely agreed-upon standard for what constitutes an OODBMS, although the Object Data Management Group (ODMG) published The Object Data Standard ODMG 3.0 in 2001 which describes an object model as well as standards for defining and querying objects. The group has since disbanded.

Currently, Not Only SQL (NoSQL) document database systems are a popular alternative to the object database. Although they lack all the capabilities of a true ODBMS, NoSQL document databases provide key-based access to semi-structured data as documents, typically using JavaScript Object Notation (JSON).

Features of an ODBMS

In their influential paper, The Object-Oriented Database Manifesto, Malcolm Atkinson and others define an OODBMS as follows:

An object-oriented database system must satisfy two criteria: it should be a DBMS, and it should be an object-oriented system, i.e., to the extent possible, it should be consistent with the current crop of object-oriented programming languages.

The first criterion translates into five features: persistence, secondary storage management, concurrency, recovery and an ad hoc query facility.

The second one translates into eight features: complex objects, object identity, encapsulation, types or classes, inheritance, overriding combined with late binding, extensibility and computational completeness.

3.4.3 DATABASE MODELS

Types of database models

There are many kinds of data models. Some of the most common ones include:

- Hierarchical database model
- Relational model
- Network model
- Object-oriented database model
- Entity-relationship model

• Document model

• Entity-attribute-value model

• Star schema

• The object-relational model, which combines the two that make up its name

You may choose to describe a database with any one of these depending on several factors. The biggest factor is whether the database management system you are using supports a particular model. Most database management systems are built with a particular data model in mind and require their users to adopt that model, although some do support multiple models.

In addition, different models apply to different stages of the database design process. High-level conceptual data models are best for mapping out relationships between data in ways that people perceive that data. Record-based logical models, on the other hand, more closely reflect ways that the data is stored on the server.

Selecting a data model is also a matter of aligning your priorities for the database with the strengths of a particular model, whether those priorities include speed, cost reduction, usability, or something else.

Let's take a closer look at some of the most common database models.

Relational model

The most common model, the relational model sorts data into tables, also known as relations, each of which consists of columns and rows. Each column lists an attribute of the entity in question, such as price, zip code, or birth date. Together, the attributes in a relation are called a domain. A particular attribute or combination of attributes is chosen as a primary key that can be referred to in other tables, when it's called a foreign key.

Each row, also called a tuple, includes data about a specific instance of the entity in question, such as a particular employee.

The model also accounts for the types of relationships between those tables, including one-to-one, one-to-many, and many-to-many relationships. Here's an example:

Within the database, tables can be normalized, or brought to comply with normalization rules that make the database flexible, adaptable, and scalable. When normalized, each piece of data is atomic, or broken into the smallest useful pieces.

Relational databases are typically written in Structured Query Language (SQL). The model was introduced by E.F. Codd in 1970.

Hierarchical model

The hierarchical model organizes data into a tree-like structure, where each record has a single parent or root. Sibling records are sorted in a particular order. That order is used as the physical order for storing the database. This model is good for describing many real-world relationships.

This model was primarily used by IBM's Information Management Systems in the 60s and 70s, but they are rarely seen today due to certain operational inefficiencies.

Network model

The network model builds on the hierarchical model by allowing many-to-many relationships between linked records, implying multiple parent records. Based on mathematical set theory, the model is constructed with sets of related records. Each set consists of one owner or parent record and one or more member or child records. A record can be a member or child in multiple sets, allowing this model to convey complex relationships.

It was most popular in the 70s after it was formally defined by the Conference on Data Systems Languages (CODASYL).

Object-oriented database model

This model defines a database as a collection of objects, or reusable software elements, with associated features and methods. There are several kinds of object-oriented databases:

A **multimedia database** incorporates media, such as images, that could not be stored in a relational database.

A **hypertext database** allows any object to link to any other object. It's useful for organizing lots of disparate data, but it's not ideal for numerical analysis.

The object-oriented database model is the best known post-relational database model, since it incorporates tables, but isn't limited to tables. Such models are also known as hybrid database models.

Object-relational model

This hybrid database model combines the simplicity of the relational model with some of the advanced functionality of the object-oriented database model. In essence, it allows designers to incorporate objects into the familiar table structure.

Languages and call interfaces include SQL3, vendor languages, ODBC, JDBC, and proprietary call interfaces that are extensions of the languages and interfaces used by the relational model.

Entity-relationship model

This model captures the relationships between real-world entities much like the network model, but it isn't as directly tied to the physical structure of the database. Instead, it's often used for designing a database conceptually.

Here, the people, places, and things about which data points are stored are referred to as entities, each of which has certain attributes that together make up their domain. The cardinality, or relationships between entities, are mapped as well.

A common form of the ER diagram is the star schema, in which a central fact table connects to multiple dimensional tables.

Other database models

A variety of other database models have been or are still used today.

Inverted file model

A database built with the inverted file structure is designed to facilitate fast full text searches. In this model, data content is indexed as a series of keys in a lookup table, with the values pointing to the location of the associated files. This structure can provide nearly instantaneous reporting in big data and analytics, for instance.

This model has been used by the ADABAS database management system of Software AG since 1970, and it is still supported today.

Flat model

The flat model is the earliest, simplest data model. It simply lists all the data in a single table, consisting of columns and rows. In order to access or manipulate the data, the computer has to read the entire flat file into memory, which makes this model inefficient for all but the smallest data sets.

Multidimensional model

This is a variation of the relational model designed to facilitate improved analytical processing. While the relational model is optimized for online transaction processing (OLTP), this model is designed for online analytical processing (OLAP).

Each cell in a dimensional database contains data about the dimensions tracked by the database. Visually, it's like a collection of cubes, rather than two-dimensional tables.

Semistructured model

In this model, the structural data usually contained in the database schema is embedded with the data itself. Here the distinction between data and schema is vague at best. This model is useful for describing systems, such as certain Web-based data sources, which we treat as databases but cannot constrain with a schema. It's also useful for describing interactions between databases that don't adhere to the same schema.

Context model

This model can incorporate elements from other database models as needed. It cobbles together elements from object-oriented, semistructured, and network models.

Associative model

This model divides all the data points based on whether they describe an entity or an association. In this model, an entity is anything that exists independently, whereas an association is something that only exists in relation to something else.

The associative model structures the data into two sets:

- A set of items, each with a unique identifier, a name, and a type
- A set of links, each with a unique identifier and the unique identifiers of a source, verb, and target. The stored fact has to do with the source, and each of the three identifiers may refer either to a link or an item.

Other, less common database models include:

- Semantic model, which includes information about how the stored data relates to the real world
- XML database, which allows data to be specified and even stored in XML format
- Named graph
- Triplestore

NoSQL database models

In addition to the object database model, other non-SQL models have emerged in contrast to the relational model:

The **graph database model**, which is even more flexible than a network model, allowing any node to connect with any other.

The **multivalue model**, which breaks from the relational model by allowing attributes to contain a list of data rather than a single data point.

The **document model**, which is designed for storing and managing documents or semi-structured data, rather than atomic data.

Databases on the Web

Most websites rely on some kind of database to organize and present data to users. Whenever someone uses the search functions on these sites, their search terms are converted into queries for a database server to process. Typically, middleware connects the web server with the database.

The broad presence of databases allows them to be used in almost any field, from online shopping to micro-targeting a voter segment as part of a political campaign. Various industries have developed their own norms for database design, from air transport to vehicle manufacturing.

3.4.4 DATABASE INTERFACE

User-friendly interfaces provide by DBMS may include the following:

1. **Menu-Based Interfaces for Web Clients or Browsing –**
These interfaces present the user with lists of options (called menus) that lead the user through the formation of a request. Basic advantage of using menus is that they removes the tension of remembering specific commands and syntax of any query language, rather than query is basically composed step by step by collecting or picking options from a menu that is basically shown by the system. Pull-down menus are a very popular technique in *Web based interfaces*.

They are also often used in *browsing interface* which allow a user to look through the contents of a database in an exploratory and unstructured manner.

2. **Forms-Based Interfaces –**

A forms-based interface displays a form to each user. Users can fill out all of the form entries to insert a new data, or they can fill out only certain entries, in which case the DBMS will re-deem same type of data for other remaining entries. This type of forms are usually designed or created and programmed for the users that have no expertise in operating system. Many DBMSs have *forms specification languages* which are special languages that help specify such forms.

Example: SQL* Forms is a form-based language that specifies queries using a form designed in conjunction with the relational database schema.b>

3. **Graphical User Interface –**

A GUI typically displays a schema to the user in diagrammatic form.The user then can specify a query by manipulating the diagram. In many cases, GUI's utilize both menus and forms. Most GUIs use a pointing device such as mouse, to pick certain part of the displayed schema dia-gram.

4. **Natural language Interfaces –**

These interfaces accept request written in English or some other language and attempt to under-stand them. A Natural language interface has its own schema, which is similar to the database conceptual schema as well as a dictionary of important words.

The natural language interface refers to the words in its schema as well as to the set of standard words in a dictionary to interpret the request.If the interpretation is successful, the interface generates a high-level query corresponding to the natural language and submits it to the DBMS for processing, otherwise a dialogue is started with the user to clarify any provided condition or request. The main disadvantage with this is that the capabilities of this type of interfaces are not that much advance.

5. **Speech Input and Output –**

There is an limited use of speech say it for a query or an answer to a question or being a result of a request it is becoming commonplace Applications with limited vocabularies such as inquir-ies for telephone directory, flight arrival/departure, and bank account information are allowed speech for input and output to enable ordinary folks to access this information.

The Speech input is detected using a predefined words and used to set up the parameters that are supplied to the queries. For output, a similar conversion from text or numbers into speech take place.

6. **Interfaces for DBA –**

Most database system contains privileged commands that can be used only by the DBA's staff. These include commands for creating accounts, setting system parameters, granting account au-thorization, changing a schema, reorganizing the storage structures of a databases

3.4.5 DATABASE SCHEMA AND DATA DEFINITION LANGUAGE

A database schema represents the logical configuration of all or part of a relational database. It can exist both as a visual representation and as a set of formulas known as integrity constraints that govern a database. These formulas are expressed in a data definition language, such as SQL. As part of a data dictionary, a database schema indicates how the entities that make up the database relate to one another, including tables, views, stored procedures, and more.

Typically, a database designer creates a database schema to help programmers whose software will interact with the database. The process of creating a database schema is called data modeling. When following the three-schema approach to database design, this step would follow the creation of a conceptual schema. Conceptual schemas focus on an organization's informational needs rather than the structure of a database.

There are two main kinds of database schema:

1. A logical database schema conveys the logical constraints that apply to the stored data. It may define integrity constraints, views, and tables.

2. A physical database schema lays out how data is stored physically on a storage system in terms of files and indices.

At the most basic level, a database schema indicates which tables or relations make up the database, as well as the fields included on each table. Thus, the terms *schema diagram* and *entity-relationship diagram* are often interchangeable.

A data definition language (DDL) is a computer language used to create and modify the structure of database objects in a database. These database objects include views, schemas, tables, indexes, etc.

This term is also known as data description language in some contexts, as it describes the fields and records in a database table.

A DDL is a language used to define data structures and modify data. For example, DDL commands can be used to add, remove, or modify tables within in a database. DDLs used in database applications are considered a subset of SQL, the Structured Query Language. However, a DDL may also define other types of data, such as XML.

A Data Definition Language has a pre-defined syntax for describing data. For example, to build a new table using SQL syntax, the CREATE command is used, followed by parameters for the table name and column definitions. The DDL can also define the name of each column

3.4.6 DATA MANIPULATION LANGUAGE(DML)

A data manipulation language (DML) is a family of computer languages including commands permitting users to manipulate data in a database. This manipulation involves inserting data into database tables, retrieving existing data, deleting data from existing tables and modifying existing data. DML is mostly incorporated in SQL databases.

DML resembles simple English language and enhances efficient user interaction with the system. The functional capability of DML is organized in manipulation commands like SELECT, UPDATE, INSERT INTO and DELETE FROM, as described below:

SELECT: This command is used to retrieve rows from a table. The syntax is SELECT [column name(s)] from [table name] where [conditions]. SELECT is the most widely used DML command in SQL.

UPDATE: This command modifies data of one or more records. An update command syntax is UPDATE [table name] SET [column name = value] where [condition].

INSERT: This command adds one or more records to a database table. The insert command syntax is INSERT INTO [table name] [column(s)] VALUES [value(s)].

DELETE: This command removes one or more records from a table according to specified conditions. Delete command syntax is DELETE FROM [table name] where [condition]

3.4.7 CONCURRENCY POLICY

The Concept of Concurrency

Concurrency is a property of a system in which several behaviors can overlap in time – the ability to

perform two or more tasks at once. In the sequential paradigm, the next step in a process can be

performed only after the previous has completed; in a concurrent system some steps are executed in

parallel.

UML and Concurrency

UML supports concurrency, and makes it possible to represent the concept in different kinds of

diagrams. This article covers the three most commonly used – the activity diagram, sequence

diagram, and state machine diagram. Note that the OCUP 2 Foundation level examination covers

concurrency only in the activity diagram; concurrency in sequence and state machine diagrams is

covered at the Intermediate and Advanced levels.

Activity diagram

In activity diagrams, concurrent execution can be shown implicitly or explicitly. If there are two or

more outgoing edges from an action it is considered an implicit split. Two or more incoming edges

signify an implicit join.

The action at an implicit join will not execute until at least one token is offered on every incoming

control flow. When the action begins execution, it will consume all tokens offered on all incoming

control flows.

Concurrent execution can also be drawn explicitly using fork and join nodes:

Explicit concurrency using fork and join nodes

Sequence diagram

Concurrency can be shown in a sequence diagram using a combined fragment with the par operator

or using a coregion area. A coregion can be used if the exact order of event occurrences on one

lifeline is irrelevant or unknown. Coregion is shorthand for parallel combined fragment within a single

lifeline.

3.4.8 OBJECT RELATIONAL SYSTEMS

Object Relational System

A large portion of the information State manipulates lies inside of a PostgreSQL database. Anybody that's used a database knows that saving and pulling data is largely a repetitive process. In State, this task is much easier because much of time we're only pulling data from a single row of information, rarely using data from joins in our actual work. In order to make State easier to debug and operate I've created a simple Object Relational System to handle saving and loading data from the database. For each of object types we define a field mapping with all of the fields we're pulling into the object from our queries and their various properties. There are then a few methods which takes queries and these field mappings and create or save objects to the database.

Peers

A Peer is just a table in the database. If you've studied any Object Relational Systems you know'll recognize the term. Peer's manage objects of a specific type. It's the peer that stores the field mapping and other details that help us manage the objects in a specific table.

SQL Backend

In a few areas we allow the ORS to generate the SQL we need, but only in simple situations where the SQL is very simple. Most of the time we have to write the SQL ourselves.

Fortunately, the ORS helps us in the larger, more common case. Maybe someday we'll expand this, but right now things seem fine.

You may have noticed that there is a single library, libsos where all of the SQL lies. This is by design, to make any future changes easier to make maintaining the code easier. Any changes to the database will require changes to this library, and everything else should work just fine. Unless of course you change a field name or add new fields, then you'll have to use those fields and that will almost always happen outside of the sos library.

3.4.9 MULTIDATABASE SYSTEM

In a multidatabase system, multiple users simultaneously access various component systems. Heterogeneous transactio~ management deals with the problem of maintaining the consistency of each component system individually and ofthemultidatabase system as a whole

A multidatabase system (MDBS) is a facility that allows users access to data located in multiple autonomous database management systems (DBMSs). In such a system,global transactions are executed under the control of the MDBS. ... Each local DBMS integrated by the MDBS may employ a different transaction management scheme.

A multidatabase system (MDBS) is a facility that allows users access to data located in multiple autonomous database management systems (DBMSs). In such a system,global transactions are executed under the control of the MDBS. Independently,local transactions are executed under the control of the local DBMSs. Each local DBMS integrated by the MDBS may employ a different transaction management scheme. In addition, each local DBMS has complete control over all transactions (global and local) executing at its site, including the ability to abort at any point any of the transactions executing at its site. Typically, no design or internal DBMS structure changes are allowed in order to accommodate the MDBS. Furthermore, the local DBMSs may not be aware of each other and, as a consequence, cannot coordinate their actions. Thus, traditional techniques for ensuring transaction atomicity and consistency in homogeneous distributed database systems may not be appropriate for an MDBS environment. The objective of this article is to provide a brief review of the most current work in the area of multidatabase transaction management. We first define the problem and argue that the multidatabase research will become increasingly important in the coming years. We then outline basic research issues in multidatabase transaction management and review recent results in the area. We conclude with a discussion of open problems and practical implications of this research.

3.4.10 OPEN DATABASE CONNECTIVITY (ODBC)

Open Database Connectivity—or ODBC—is an application programming interface (API) that lets software connect with database management systems while remaining independent of them. This is important, because it allows applications to interact with multiple databases simultaneously using SQL (Structured Query Language).

For organizations that have multiple data streams and must store them on separate databases, ODBC offers a solution that lets them use the software they need without having to worry about which database management system they have to use.

It's useful to think of ODBC as a peripheral driver which lets specific tools connect to a program. Much like printers require the specific instructions to allow them to connect with multiple different computers and devices, ODBC is a bridge between applications and the databases they require. Additionally ODBC allows organizations to centralize their data streams into a single application or dashboard more efficiently.

For organizations that use multiple database management systems and streams, ODBC is one of the easiest ways to centralize and manage data without having to use multiple systems at the same time. One of the clearest use cases for ODBC is for creating dashboards.

For most organizations, dashboards—even specific ones—tend to draw data from multiple internal and sometimes external sources. As such, using an ODBC connector can improve several areas of the analytics process.

Open Database Connectivity lets developers connect their existing data visualization tools to any database, greatly increasing their accuracy and depth. Companies that must constantly interact with multiple databases simultaneously for analytics can also optimize their querying ability and draw information from a broader range of sources, as well as create more granular reports.

Additionally, ODBC allows companies to sort and store their data more efficiently. Instead of forcing a single database management system to handle data it may not be optimally suited for, organizations can instead mash up multiple sources without worrying about their compatibility or accessibility.

Open Database Connectivity (ODBC) is a specification for an application programming interface (API) that enables applications to access multiple database management systems using Structured Query Language (SQL). Using the ODBC interface in your COBOL applications, you can dynamically access databases and file systems that support the ODBC interface.

ODBC permits maximum interoperability: a single application can access many different database management systems. This interoperability enables you to develop, compile, and ship an application without targeting a specific type of data source. Users can then add the database drivers, which link the application to the database management systems of their choice.

When you use the ODBC interface, your application makes calls through a driver manager. The driver manager dynamically loads the necessary driver for the database server to which the application connects. The driver, in turn, accepts the call, sends the SQL to the specified data source (database), and returns any result.

Your COBOL applications that use embedded SQL for database access must be processed by a precompiler or coprocessor for a particular database and need to be recompiled if the target database changes. Because ODBC is a call interface, there is no compile-time designation of the target database as there is with embedded SQL. Not only can you avoid having multiple versions of your application for multiple databases, but your application can dynamically determine which database to target.

Some of the advantages of ODBC are:

ODBC provides a consistent interface regardless of the kind of database server used.

You can have more than one concurrent connection.

Applications do not have to be bound to each database on which they will run. Although COBOL for AIX does this bind for you automatically, it binds automatically to only one database. If you want to choose which database to connect to dynamically at run time, you must take extra steps to bind to a different database.

MAR GREGORIOS COLLEGE, CHENNAI, INDIA
LET YOUR LIGHT SHINE

Unit4

Userinterfacedesign

The visual part of a computer application or operatingsystemthroughwhichaclientinteractswithacomputeror software. It determines how commands are given tothe computer or the program and how data is displayedonthescreen

User interface is the front-end application view to which user interactsinordertousethesoftware.Usercanmanipulateandcontrolthesoftware as well as hardware by means of user interface. Today, userinterface is found at almost every place where digital technology exists,right from computers, mobile phones, cars, music players, airplanes,ships etc.

User interface is part of software and is designed such a way that it isexpectedtoprovidetheuserinsightofthesoftware.UIprovidesfundamentalplatform forhuman-computerinteraction.

UIcanbegraphical,text-based,audio-videobased,dependinguponthe underlying hardware and software combination. UI can be hardware orsoftwareoracombinationofboth.

Thesoftwarebecomesmorepopular ifitsuserinterfaceis:

- Attractive
- Simpletouse
- Responsiveinshorttime
- Cleartounderstand
- Consistent on all interfacing screensUIisbroadly dividedintotwocategories:
- Command Line InterfaceGraphical

UserInterface

User interface is the front-end application view to which user interactsin order to use the software. The software becomes more popular if itsuserinterfaceis:

- Attractive
- Simpletouse
- Responsiveinshorttime
- Cleartounderstand
- Consistentonallinterfacescreens

User,task,environmentalanalysis,andmodeling: Initially,thefocusisbasedontheprofileofuserswhowillinteractwiththesystem,i.e.understanding,skillandknowledge,typeofuser,etc,basedontheuser'sprofileusersaremadeintocategories.Fromeachcategoryrequirementsaregathered.Basedontherequirementsdeveloperunderstandhowtodeveloptheinterface.Onceallthe requirementsaregatheredadetailedanalysisisconducted. Intheanalysispart,thetasksthattheuserperformstoestablishthegoalsofthesystemareidentified,describedandelaborated.Theanalysisoftheuserenvironmentfocusesontheph

ysicalworkenvironment.Amongthequestionstobeaskedare:

1 *InterfaceDesign:*
2 Thegoalofthisphaseistodefinethesetofinterfaceobjectsandactions i.e.Controlmechanismsthatenabletheusertoperformdesiredtasks.Indicatehowthesec ontrolmechanismsaffectthesystem.Specifytheactionsequenceoftasksandsubtasks,al socalledauserscenario.Indicatethestateofthesystemwhentheuserperformsaparticula rtask.AlwaysfollowthethreegoldenrulesstatedbyTheoMandel.Designissuessuchasr esponsetime,commandandactionstructure,errorhandling,andhelpfacilitiesareconsid eredasthedesignmodelisrefined.Thisphaseservesasthefoundationfortheimplementa tionphase.

DesigningUserInterfacesforUsers

User interfaces are the access points where users interact with designs.They come in three formats:

Graphicaluserinterfaces(GUIs)—Usersinteractwithvisual representationsondigitalcontrolpanels.Acomputer'sdesktopisa GUI.

Voice-controlled interfaces (VUIs)—Users interact with thesethrough their voices. Most smart assistants—e.g., Siri on iPhone andAlexaonAmazon devices—are VUIs.

Gesture-based interfaces—Users engage with 3D design spacesthroughbodily motions:e.g.,ingames.

TodesignUIsbest,youshouldconsider:

Aim**Userinterface (UI)design**istheprocess**designers** usetbuild **interfaces** in software or computerized devices, focusing on looksor style. **Designers**to create **interfaces** which **users** find easy to useand pleasurable.

- Allowtheusertoputthecurrenttaskintoameaningfulcontext:Manyinterfaceshaved ozensofscreens.Soitisimportanttoprovideindicatorsconsistentlysothattheuserknowaboutt hedoingwork.Theusershouldalsoknowfromwhichpagehasnavigatedtothecurrentpageandf romthecurrentpagewherecannavigate.
- Maintainconsistencyacrossafamilyofapplications:Thedevelopmentofsomesetof applicationsallshouldfollowandimplementthesamedesign,rulessothatconsistencyismainta inedamongapplications.
- Ifpastinteractivemodelshavecreateduserexpectationsdonotmakechangesunlesst hereisacompellingreason.
- User interface is a design for softwares and machinessuchas

computers,mobiledevices etc.

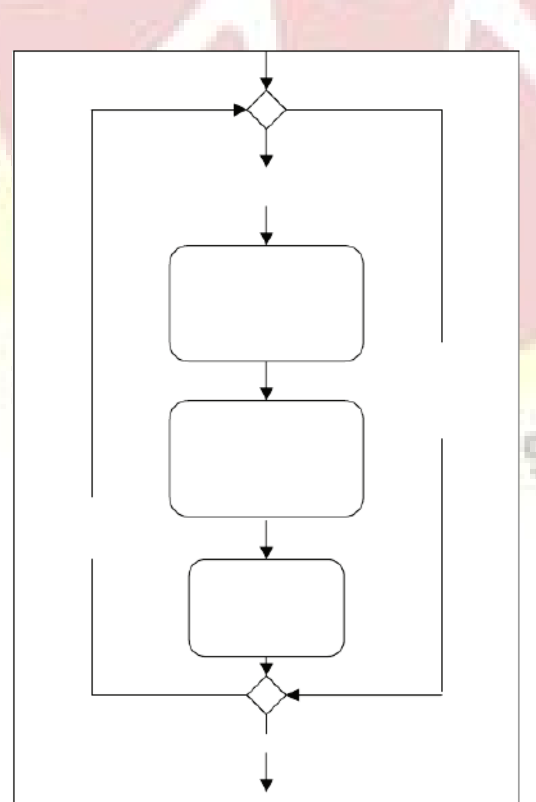
- Itisthefirstimpressionofasoftwarewhereuserinteractswithacomputerorasof twaresystem.
- Asoftwaremustfulfilltherequirementofauser.
- Userinterfacedetermineshowtheinformationisdisplayedon thescreen.
- Poor user interface design causes a user to make fatalerrorsand asoftware systemnever used.

TherearethreetypesofUserInterfaces:

1 Commandlanguage
2 Menus
3 GraphicalUserInterface(GUI)

VIEWLAYERCLASSES

View layer is the only exposed objects of an application with whichusers can interact. Represents the set of operations in the business thatusers must perform to complete their tasks. **View layer** objects areresponsiblefortwomajoraspestsofapplication



Designing **view layer** classes• UI **layer** consists of – objects with whichuser interacts – Objects needed to manage or control UI• Responsibilityof **view layer** objects – Input :responding to user interaction •Translatinguser'sactionintoanappropriateresponse–Output :displayingorprintingbusinessobjects

Main goal of UI is to display and obtain needed information in anaccessible,efficientmannerUIdesign is acreativeprocess

UI layer consists ofobjects with which user interactsObjects needed tomanageorcontrolUIResponsibilityofview layerobjectsInput :responding to user interactionTranslating user's action into anappropriateresponseOutput:displayingorprintingbusinessobjects

It has 4 major activitiesMacro level UI design process: (identifying viewlayer object)Takes place during analysis phaseIdentifying classes thatinteract with human actors by analysing use casessequence andcollaboration diagram helps to identify UI classesMicro level UI designactivitiesDesigningtheviewlayerobjectsbyapplyingdesignaxiomsandcorollariesDecide how to use and extend the components so they bestsupport application specific functions and provide the most usableinterfacePrototyping the view layer interfaceUseful in the early designprocessTesting usability and user satisfactionRefining and iterating thedesign

Presentation **layer** - this is the **User Interface layer**. The screens areusing information both from components and from data stored in legacysystem.

The micro **process** is concerned with specific analysis and designtechniques—the techniques you use to get from requirements toimplementation, and the choice of analysis and design techniques (e.g.,structured,object-oriented,andsoon)affectsthemicro **process**..

. To control what data users see, the owner of a hosted feature **layerview**, or an administrator, can **define** what fields or features areavailable in the **view**. You can also limit the hosted feature **layer view** toa specificareaby**defining**aspatialextent.

Hosted feature **layer views** allow you create multiple uniquewindows into your data, and customize them to fit youraudience. For a while now you've been able to create a copy ofa **layer** and modify its symbology and other basic properties,but copied **layers** are only used to customize the presentationofdata

Tocontrol whatdatauserssee,theownerofahostedfeaturelayerview, or an administrator, can define what fields or features are availablein the view. You can also limit the hosted feature layer view to a specificareabydefiningaspatialextent.Thesedefinitionsaresavedwiththehosted feature layer view and allow you more control over what contentpeoplesee.

For example, you might create multiple different views of a hostedfeature layer containing customer information, and set differentdefinitions for each view depending on the intended users. For a viewyou share with a group that will perform spatial analysis, you might hidethefieldsthatstorecustomernames,astheanalysts don'tneedtoknowthis information. For another view that you share with a group concernedwith routing

deliveries, you would define the view to show only featuresrepresenting customers who purchased a product that had not yet beendelivered.

MACROLEVELPROCESS

Themacrodevelopmentprocessconsistsofthefollowingsteps:

1 Conceptualization. Establish the core requirements and develop aprototype.
2 Analysisanddevelopmentofthemodel....
3 Designorcreatethesystemarchitecture....
4 Evolution or implementation-5.

Software **Development Macro Process** (SDMaP): this is the overallsoftware **development** lifecycle. In this **process** model we define ourrequirements, we execute analysis, design, implementation, test and wedeploythesoftwareintoproduction.

Themacrodevelopmentprocessconsistsofthefollowingsteps:

Conceptualization

Establishthecorerequirementsanddevelopaprototype.

Analysisanddevelopmentofthemodel

Use the class diagram to describe the roles and responsibilities of objects. Usetheobjectdiagramto describethedesiredbehaviorofthesystem.

Designorcreatethesystemarchitecture.

Use the class diagram to decide what classes exist and how they relate to eachother, the object diagram to decide what mechanisms are used, the modulediagram to map out where each class and object should be declared, and theprocessdiagramto determineto whichprocessor toallocateaprocess.

Evolutionorimplementation-

Refinethesystemthroughmuchiteration.

Themacrodevelopmentprocessconsistsofthefollowingsteps:

Conceptualization

Establishthecorerequirementsanddevelopaprototype.

Analysisand developmentofthemodel

Use the class diagram to describe the roles and responsibilities of objects. Use theobjectdiagram to describethedesiredbehaviorof thesystem.

Designorcreatethesystemarchitecture.

Use the class diagram to decide what classes exist and how they relate to eachother,theobjectdiagramtodecidewhatmechanismsareused,themodulediagramto map out where each class and object should be declared, and the processdiagramto determinetowhich processortoallocateaprocess.

Evolutionorimplementation-

Refinethesystemthroughmuchiteration.

Maintenance-Make localized changes to the system to add new requirementsandeliminatebugs.

MacroLevel DesignProcess MicroDevelopmentprocess

The micro process is a description of the day-to-day activities by a single orsmallgroupofsoftware developers.
Itconsistsofthefollowingsteps.

- Identifyclassesandobjects.
- Identifyclassandobjectsemantics.
- Identifyclassandobjectrelationships.
- Identifyclassandobjectinterfaceandimplementation

Themacrodevelopmentprocessconsistsofthefollowingsteps:

Conceptualization

Establishthecorerequirementsanddevelopaprototype.

Analysisanddevelopmentofthemodel

Use the class diagram to describe the roles and responsibilities of objects. Use theobjectdiagram to describethedesiredbehaviorof thesystem.

Designorcreatethesystemarchitecture.

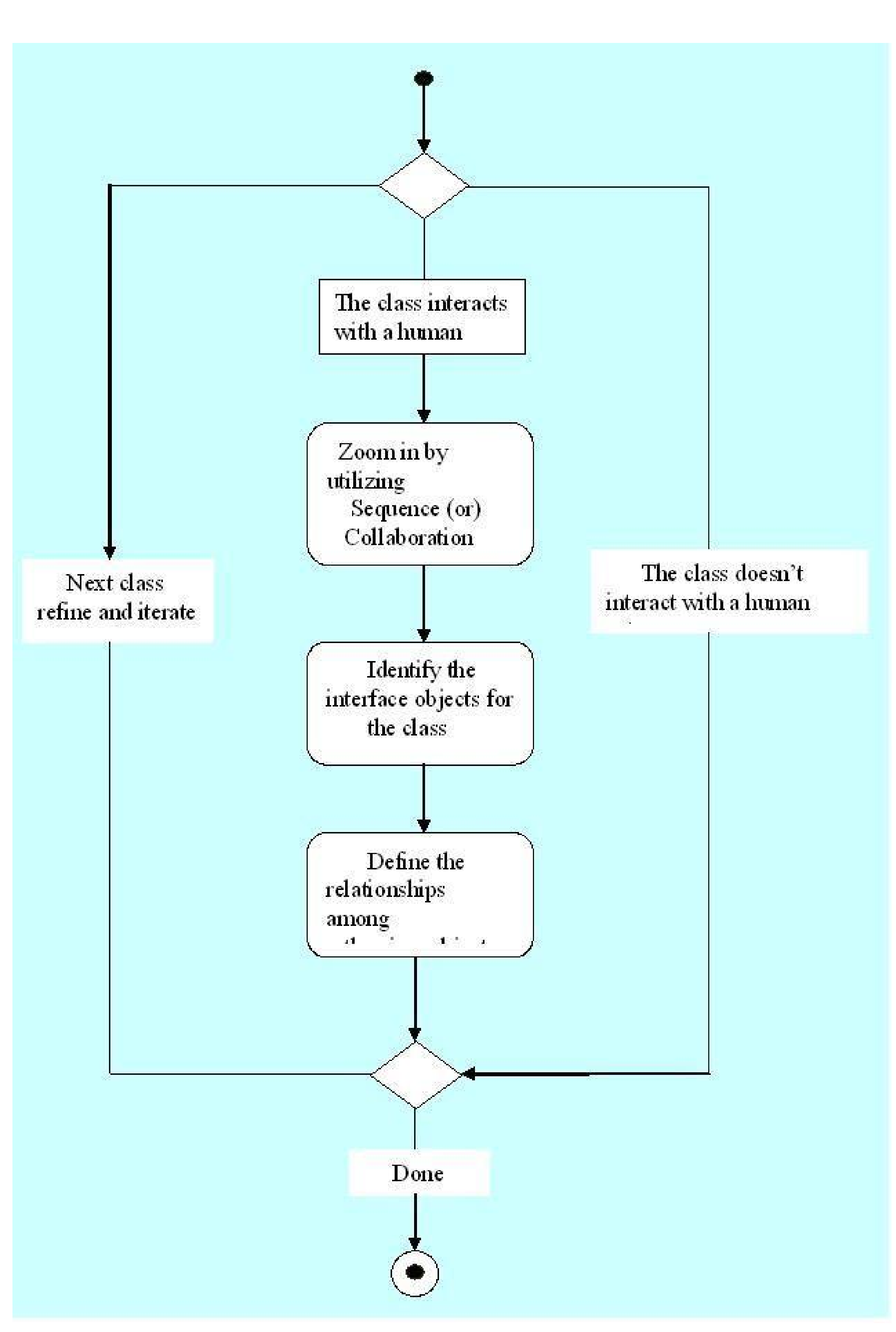
Use the class diagram to decide what classes exist and how they relate to eachother, the object diagram to decide what mechanisms are used, the module diagramtomapout where each classandobjectshouldbedeclared,andtheprocessdiagramto determinetowhich processortoallocateaprocess.

Evolutionorimplementation-

Refinethesystemthroughmuch iteration.

Maintenance-Make localized changes to the system to add new requirementsandeliminatebugs.

MacroLevel DesignProcess

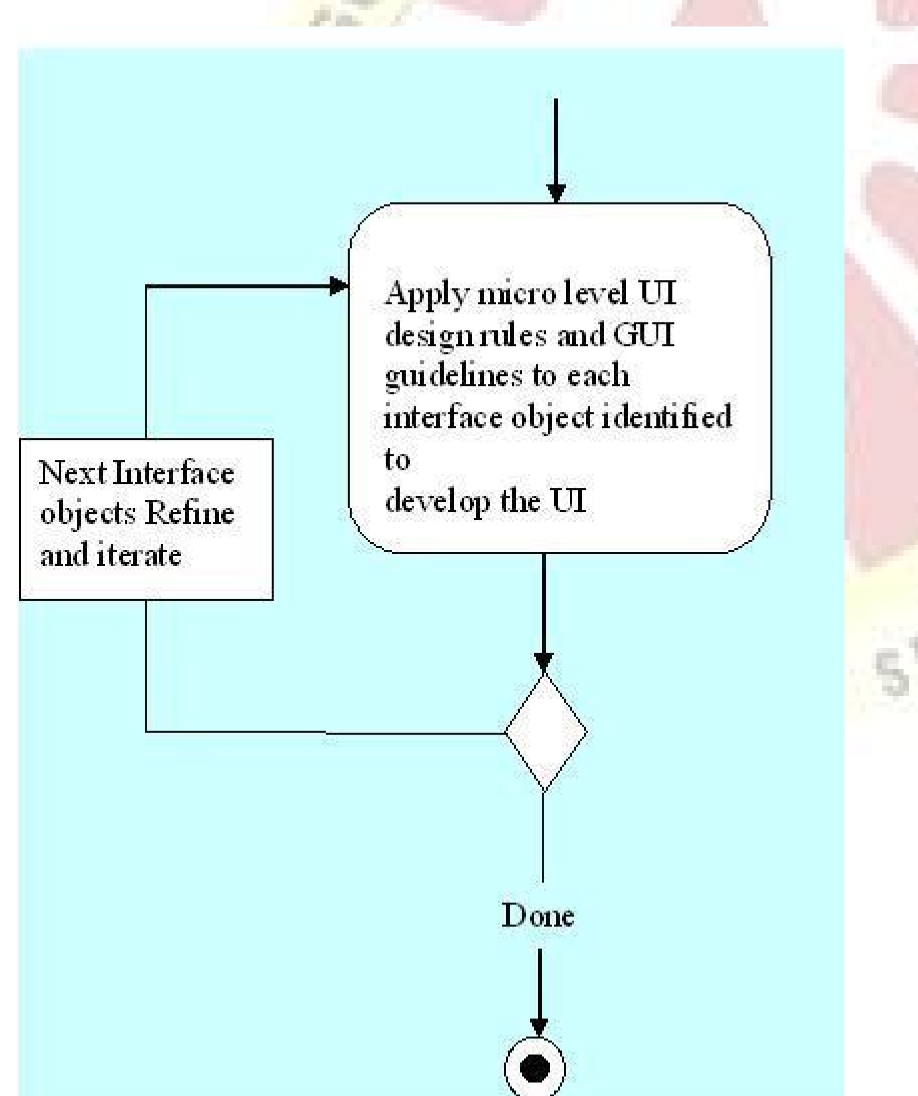

The class interacts with a human
Zoom in by utilizing Sequence (or) Collaboration
Identify the interface objects for the class
Define the relationships among
Next class refine and iterate
The class doesn't interact with a human
Done

MicroDevelopmentprocess

The micro process is a description of the day-to-day activities by a single orsmallgroupofsoftware developers.
Itconsistsofthefollowingsteps.

- Identifyclassesandobjects.
- Identifyclassandobjectsemantics.
- Identifyclassandobjectrelationships.
- Identifyclassandobjectinterfaceandimplementation.

MicroLevel DesignProcess



Designingviewlayerobjects:

o Effectiveinterfacedesignisfollowingthesetofrules.
o Italsoinvolvesearlyplanningoftheinterfaceandcontinuedworkthrough thesoftwaredevelopmentprocess.
o Theprocessofdesigningtheuserinterfaceinvolves,
▪ Classifyingthespecificneedoftheapplication.
▪ Identifytheusecasesandinterfaceobjects.
▪ Devisingthedesign thatbestmeetsuser'sneeds
▪ UIDesign Rules:

Rule1: Makingtheinterfacesimple

Rule2: Makingtheinterfacetransparentandnatural.
Rule3: Allowingusers tobeincontrol atthesoftware.

Rule1:Making theinterfacesimple
EachUIclassmusthaveasingle,clearly definedpurpose.
Factorsforaffecting Designapplication:

▪ Deadlines may require you to deliver a product to market with a minimaldesignprocess.

Comparativeevaluationsmayforceyoutoconsider additionalfeatures. **Rule 2: Making the interface transparent andnatural.**

There showed be strong mapping between the user's view ofdoingthingsandUI classes.

Example: Billing, Insurance, Inventory and Bankingapplications can represent forms that are visually equivalent tothepaperforms users are accustomedtoseeing.

UIDesign:

- UI should not make users focus on the mechanism of anapplication.
- A good user interface doesn't bother the user withmechanics.
- A goal of user interface design is to make the userinteraction with the computer as simple and natural aspossible.

Rule3:Allowinguserstobeincontrolat thesoftware.
Users always should "feel in control of the software"ratherthan"feelingcontrolledbythesoftware".

Rule:
UI object should represent, at most, one business object,perhaps that someservices

ofthatbusinessobject.

- Main idea is to avoid creating a single UI class forseveral business objects, since it makes the UI lessflexible and faces the user to perform tasks in amonolithicway.

Waysto put usersin control:

- Maketheinterfaceforgiving
- Maketheinterface visual.
- Provideimmediatefeedback
- Avoidmodeslike,
  - Modal dialog
  - Spring-loadedmodes.
  - Tool-drivenmodes.
  - Maketheinterfaceconsistent.

IDENTIFYINGVIEWCLASSESBYANALYZINGUSECASE

Use case analysis is a technique **used** to identify the requirements of asystem (normally associated with software/process design) and theinformation **used** to both define processes **used** and **classes** (which area collection of actors and processes) which will be **used** both in the **usecase** diagramandthe overall**usecase**

- Analysis is an attempt to build a model that describes theapplicationdomain--developersdothis
- Takesplace after(orduring)requirementsspecification
- The analysis model will typically consist of all three types ofmodelsdiscussed before:
  - Functionalmodel(denotedwithusecases)Analysisobject model(classandobjectidagrams)
  - Dynamicmodel
- Atthislevel,notethatwearestilllookingattheapplicationdomain.
  - Thisisnotyetsystemdesign
  - However, many things discovered in analysis couldtranslateclosely into thesystemdesign
- Goal is to completely understand the application domain (theproblemathand,anyconstraintsthatmustbeadheredto,etc.)
  - Newinsightsgainedduringanalysismightcauserequirement sto beupdated.

- Analysisactivitiesinclude:
    - Identifying objects (often from use cases as a startingpoint)
    - Identfyingassociationsbetweenobjects
    - Identifying general attributes and responsibilities ofobjects
    - Modelinginteractionsbetweenobjects
    - Modeling how individual objects change state -- helpsidentifyoperations
    - Checking the model against requirements, makingadjustments,iteratingthroughtheprocessmorethanonce
- We often think of objects in code as mapping to some object we want torepresentin therealworld.Although thisisn'talwaysthecase.
- Herearesomecategoriesofobjectstolookfor:
    - **Entity objects** -- these represent persistent information tracked by asystem.Thisis theclosestparallelto"realworld"objects.
    - **Boundary objects** -- these represent interactions between user andsystem.(For instance,abutton,aform,adisplay)
    - **Control objects** -- usually set up to manage a given usage of thesystem.Oftenrepresentthecontrolofsomeactivityperformedbya system
- UML diagrams can include a label known as a *stereotype*, above the classname in a class diagram. This would be placed inside <<>> marks, likethis:
    - <<entity>>
    - <<boundary>>

- <<control>>

- Note: Different sources and/or "experts" will give other categorizations oftypes ofobjects
- There are some different popular techniques for identifying objects. Twotraditionalandpopularonesthatwewilldiscussare:
    - naturallanguageanalysis(i.e.partsofspeech)
    - CRCcards
- It also helps to interact with domain experts -- these are people who arealready well-versedinthe realmbeingstudied.
- NotethatthegoalintheanalysisphaseisNOTtofindimplementationspecificobjects, like HashTableor Stack.
    - Thisstagestillmodelstheapplicationdomain

TheDesignSolution

User interfaces should be simple and allow the users to enter dataquickly and, where possible, without the need to go back and correcttheir mistakes. As best you can, you should stretch to predict the variouscombinations of possible entered words and numbers and target typos.In other words, you need to build some flexibility into your system, some'smart'featurethataffordsthisunpredictabilityandacceptsvariations.
You need to show the users you're listening to them while they're makingtheirwaythroughthe pages you'vecreatedfor them.

Therefore, there must be a 'Forgiving Format' in your user interfacedesignthat allows users to make mistakes, while correcting them ontheir behalf, or performing the desired function without the need forcorrective measures. You need a fluid design—happily, you *can* get theinformation you need from your users without having to resort to a rigidkey-and-keyhole approach that will tend to frustrate them when theystumble.

There are many small tasks in the majority of user experiences; if userswere forced to correct *all* of the mistakes or slip-ups they made along theway, they would soon abandon the user interface for a much lessdemoralizingexperience. Manyuserswill encounterourdesignsin rushed circumstances, or environments where they don't have the luxuryof sitting down and focusing entirely. Even when they *do* have a quietmomentinacomfortableseat—mistakescanhappenalltooeasily;users' 'job descriptions', as far as we're concerned, don't involve proofreading.In fact, they don't have any 'job description' as far as the interfaces wedesign are concerned—that would involve work on their part, remember.Any work they must do has to be entirely focused on the *other* side of thescreen with the intent they have to complete a task (e.g., paying a bill orbooking a flight with the organizations we're working for), not on thedesign itself. So, the user interface design must allow users to carry outtheir tasks in a free and easy fashion, cleaning up after them as they goalongtoensure theydonot havetomake continualbackwardsteps.

User interface is the front-end application view to which user interacts inorder to use the software. User can manipulate and control the softwareas well as hardware by means of user interface. Today, user interface isfound at almost every place where digital technology exists, right fromcomputers,mobilephones,cars,music players,airplanes,shipsetc.

User interface is part of software and is designed such a way that it isexpectedtoprovidetheuserinsightofthesoftware.UIprovidesfundamentalplatf orm forhuman-computerinteraction.

UIcanbegraphical,text-based,audio-videobased,dependinguponthe underlying hardware and software combination. UI can be hardware orsoftwareoracombinationofboth.

Thesoftwarebecomesmorepopularifitsuserinterfaceis:

- Attractive
- Simpletouse
- Responsiveinshorttime
- Cleartounderstand
- Consistent on all interfacing

screensUIisbroadly

dividedintotwocategories:

- CommandLineInterface
- GraphicalUserInterface

CommandLineInterface(CLI)

CLI has been a great tool of interaction with computers until thevideo displaymonitorscameintoexistence.CLIisfirstchoiceofmanytechnicalusersandprogrammers.CLIisminimuminterfacea software canprovideto itsusers.

CLI provides a command prompt, the place where the user types the command andfeeds to the system. The user needs to remember the syntax of command and itsuse.EarlierCLI werenotprogrammedtohandletheusererrorseffectively.

Acommandisatext-basedreferencetosetofinstructions,whichareexpectedtobeexecutedbythesystem.Therearemethodslikemacros,scriptsthatmakeiteasyfortheuser tooperate.

CLIuseslessamountofcomputerresourceascomparedtoGUI.

UserInterfaceDesign

Thevisualpartofacomputerapplicationoroperatingsystemthrough which a client interacts with a computer or software.It determines how commands are given to the computer ortheprogramandhowdataisdisplayedonthescreen.

TypesofUser Interface

There are twomaintypesofUserInterface:

- Text-BasedUserInterfaceorCommandLineInterface
- GraphicalUserInterface(GUI)

Text-Based User Interface: This method relies primarily onthekeyboard.Atypical exampleofthisisUNIX.

AVOIDMODELS

Theobject-orientedparadigmtookitsshapefromtheinitialconceptofanewprogramming approach, while the interest in design and analysis methods camemuchlater.

- Thefirstobject–orientedlanguagewasSimula(Simulationofrealsystems)thatwasdevelopedin1960byresearchersattheNorwegianComputingCenter.
- In 1970, Alan Kay and his research group at Xerox PARK created a personalcomputer named Dynabook and the first pure object-oriented programminglanguage(OOPL)-Smalltalk, forprogrammingthe Dynabook.
- In the 1980s, Grady Booch published a paper titled Object Oriented Designthat mainly presented a design for the programming language, Ada. In theensuing editions, he extended his ideas to a complete object–oriented designmethod.
  - Inthe1990s,Coadincorporatedbehavioralideastoobject-orientedmethods.

TheothersignificantinnovationswereObjectModellingTechniques(OMT)byJamesRumbaughandObject-OrientedSoftwareEngineering(OOSE)byIvarJacobson.

Object-OrientedAnalysis

Object–OrientedAnalysis(OOA)istheprocedureofidentifyingsoftwareengineeringrequirementsanddevelopingsoftwarespecificationsintermsofasoftwaresystem'sobjectmodel, which comprisesofinteractingobjects.

The main difference between object-oriented analysis and other forms of analysis isthat in object-oriented approach, requirements are organized around objects, whichintegratebothdataandfunctions.Theyaremodelledafterreal-worldobjectsthatthesysteminteractswith.Intraditionalanalysismethodologies,thetwoaspects-functionsanddata-areconsideredseparately.

Grady Booch has defined OOA as, *"Object-oriented analysis is a method of analysisthatexaminesrequirementsfromtheperspectiveoftheclassesandobjectsfoundinthevocabularyofthe problemdomain"*.

Theprimarytasksinobject-orientedanalysis(OOA)are−

- Identifyingobjects
- Organizingtheobjectsbycreatingobjectmodeldiagram
- Definingtheinternalsoftheobjects,orobjectattributes
- Definingthebehavioroftheobjects,i.e.,objectactions
- Describinghowtheobjectsinteract

ThecommonmodelsusedinOOAareusecasesandobjectmodels.

Object-OrientedDesign

Object–Oriented Design (OOD) involves implementation of the conceptual modelproduced during object-oriented analysis. In OOD, concepts in the analysis model,whicharetechnology−independent,aremappedontoimplementingclasses,constraints are

identified and interfaces are designed, resulting in a model for thesolution domain, i.e., a detailed description of how the system is to be built onconcretetechnologies.

Theimplementationdetailsgenerallyinclude−

- Restructuringtheclassdata(ifnecessary),
- Implementationofmethods,i.e.,internaldatastructuresandalgorithms,
- Implementationofcontrol,and
- Implementationofassociations.

GradyBoochhasdefinedobject-orienteddesignas *"amethodofdesignencompassingtheprocessofobject-orienteddecompositionandanotationfordepictingbothlogicalandphysicalaswellasstaticanddynamicmodelsofthesystemunderdesign"*.

Object-OrientedProgramming

Object-orientedprogramming(OOP)isaprogrammingparadigmbaseduponobjects(havingbothdataand methods)thataimstoincorporatetheadvantagesofmodularityand reusability. Objects, which are usually instances of classes, are used to interactwith one anothertodesignapplicationsand computerprograms.

Theimportantfeaturesofobject–orientedprogrammingare−

- Bottom–upapproachinprogramdesign
- Programsorganizedaroundobjects,groupedinclasses
- Focusondata withmethodstooperateuponobject'sdata
- Interactionbetweenobjectsthroughfunctions
- Reusabilityofdesignthroughcreationofnewclassesbyaddingfeaturestoexisting classes

Someexamplesofobject-orientedprogramminglanguagesareC++,Java,S

The framework just presented provides a list of generic activitiescommon to most models of the software process. However, each modeltreats the activities differently, and each model is suitable for differentprojectsandfordifferentteams.

It is important to realise that the activities outlined in the processmodelsgivenbelow shouldbe modified,based on:

- Theproblemhavingtobesolved.
- Thecharacteristicsoftheproject.
- Thenatureof thedevelopment team.
- Theorganisationalculture.

Prescriptiveandagilemodels

Prescriptive software models are those which prescribe the componentswhich make up a software model, including the activities, the inputs andoutputs of the activities, how quality assurance is performed, howchange is managed, and so on. A prescriptive model also describes howeach of these elements are related to one another (note that in thissense, “prescriptive” is not meant to indicate that these methods admitno modificationto them,as wepreviouslyused theword).

Ontheotherhand,agilesoftwaremodels haveaheavyfocus
on change in the software engineering process. Agile methods note thatnot only do the software requirements change, but so do teammembers, the technology being used, and so on. We will discuss agilemethods laterin this chapter.

MAKETHEINTERFACECONSITENT

Consistency occurs when the stimuli / usage pairs for **UI** elements arethesameasthoseforother **UI**elementsthat alreadyexist.We
achieve **consistency** when an interaction with a user **interface** (**UI**)elementmatchesusers' expectations

Userinterfaceisthefirstimpressionofasoftwaresystemfromtheuser’spointofview. Thereforeanysoftwaresystemmustsatisfytherequirementofuser.
UImainlyperformstwofunctions−

- Acceptingtheuser’sinput
- Displayingtheoutput

User interface plays a crucial role in any software system. It is possiblytheonlyvisible aspectofasoftwaresystemas−

- Userswillinitiallyseethearchitectureofsoftwaresystem’sexternaluserinterfacewithoutconsideringitsinternalarchitecture.
- A good user interface must attract the user to use the softwaresystem without mistakes. It should help the user to understand thesoftware system easily without misleading information. A bad UImaycausemarketfailureagainstthecompetitionofsoftwaresystem.

- UIhasitssyntaxandsemantics.Thesyntaxcomprisescomponenttypes such as textual, icon, button etc. and usability summarizesthe semantics of UI. The quality of UI is characterized by its lookandfeel(syntax)and itsusability(semantics).
- Therearebasicallytwomajorkindsofuserinterface−a)Textualb)Graphical.
- Software in different domains may require different style of its userinterface for e.g. calculator need only a small area for displayingnumericnumbers,butabigareaforcommands,Awebpageneedsforms,links,tabs,etc.

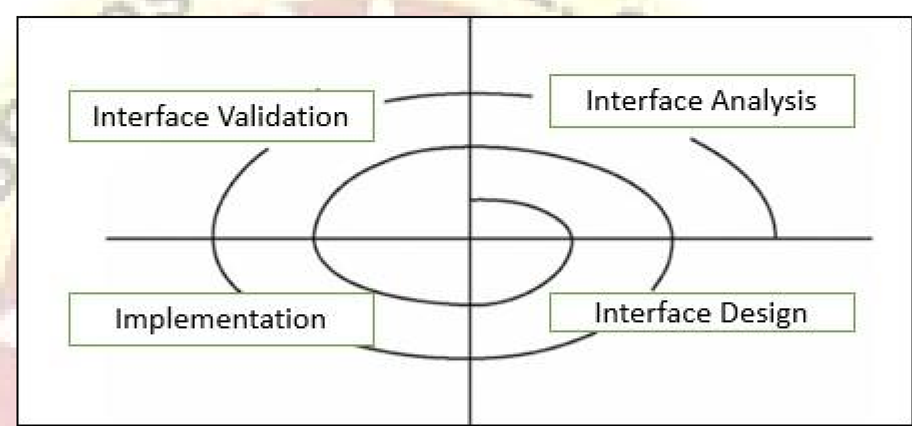
DesignofUserInterface

It starts with task analysis which understands the user's primary tasksandproblemdomain.Itshouldbedesignedintermsof User'sterminologyandout setofuser's jobratherthanprogrammer's.

- To perform user interface analysis, the practitioner needs to studyandunderstandfourelements−
  - The**users**whowillinteractwiththesystemthroughtheinterface
  - The**tasks**thatendusersmustperformtodotheirwork
  - The**content**thatispresentedaspartoftheinterface
  - The**work environment**in which these tasks will beconducted
- ProperorgoodUIdesignworksfromtheuser'scapabilitiesandlimitationsnotthe machines. While designing the UI, knowledge of the nature of the user'sworkandenvironmentis also critical.
- The task to be performed can then be divided which are assigned to the useror machine, based on knowledge of the capabilities and limitations of each.Thedesignofa userinterface isoftendividedintofourdifferent levels−
  - **The conceptual level** − It describes the basic entities considering theuser'sviewofthesystemandtheactionspossible uponthem.
  - **Thesemanticlevel** −Itdescribesthefunctionsperformedbythesystemi.e.descriptionofthefunctionalrequirementsofthesystem,butdoesnotaddress howtheuserwill invokethefunctions.
  - **Thesyntacticlevel**−Itdescribesthesequencesofinputsandoutputsrequiredtoinvoke thefunctions described.

- **Thelexicallevel**–Itdetermineshowtheinputsandoutputsareactuallyformedfromp rimitive hardware operations.

- User interface design is an iterative process, where all the iteration explainsand refines the information developed in the preceding steps. General stepsforuser interface design
  - Definesuserinterfaceobjectsandactions(operations).
  - Definesevents(useractions)thatwillcausethestateoftheuserinterfaceto change.
  - Indicateshowtheuserinterpretsthestateofthesystemfrominformationprovided throughtheinterface.
  - Describeeachinterfacestate asitwillactuallylookto theenduser.



- **Interfaceanalysis**

- Itconcentratesorfocusesonusers,tasks,content,andworkenvironmentwhowi llinteractwiththesystem.Definesthehuman -and computer-oriented tasks that are required to achieve systemfunction.
- *Interfacedesign*
- Itdefinesasetofinterfaceobjects,actions,andtheirscreenrepresentations that enable a user to perform all defined tasks in amannerthatmeetseveryusabilityobjectivedefinedforthesystem.
- *Interfaceconstruction*
- Itstartswithaprototypethatenablesusagescenariostobeevaluated and continues with development tools to complete theconstruction.
- *Interfacevalidation*
- Itfocusesontheabilityoftheinterfacetoimplementeveryusertaskcorrectly,ac commodatealltaskvariations,toachieveallgeneral

userrequirements,andthedegreetowhichtheinterfaceiseasytouseandeasytol earn.

UserInterfaceModels

Whenauserinterfaceisanalyzedanddesignedfollowingfourmodelsareused−

Userprofilemodel

- Created by a user or software engineer, which establishes theprofile of theend-usersof the systembased on age, gender,physicalabilities, education,motivation, goals,andpersonality.
- Considerssyntacticandsemanticknowledgeoftheuserandclassifiesusersasn ovices,knowledgeableintermittent,andknowledgeablefrequent users.

Designmodel

- Createdbyasoftwareengineerwhichincorporatesdata,architectural,interface, andproceduralrepresentationsofthesoftware.
- Derived from the analysis model of the requirements and controlled by theinformation in the requirements specification which helps in defining the userofthe system.

Viewlayerinterface

GUIDELINESFORDESIGININGFORMSANDDATAENT RY WINDOWS

4.1 Fields in data entry screens contain default values whenappropriate and show the structure of the data and the fieldlength.
4.2 When a task involves source documents (such as a paperform), the interface is compatible with the characteristics ofthe sourcedocument.
4.3 The site automatically enters field formatting data (e.g.currency symbols, commas for 1000s, trailing or leadingspaces).Usersdonotneedtoentercharacterslike£or%..
4.4 Fieldlabelsonformsclearlyexplainwhatentriesaredesired.

4.5 Text boxes on forms are the right length for the expectedanswer.
4.6 There is a clear distinction between "required" and "optional"fieldsonforms.
4.7 The same form is used for both logging in and registering(i.e.it'slikeAmazon).
4.8 Forms pre-warn the user if external information is needed forcompletion(e.g. apassportnumber).
4.9 Questions on forms are grouped logically, and each group hasaheading.
4.10 Fields on forms contain hints, examples or model answers todemonstrate theexpectedinput.
4.11 When field labels on forms take the form of questions, thequestionsare stated inclear,simplelanguage.
4.12 Pull-down menus, radio buttons and check boxes are used inpreference to text entry fields on forms (i.e. text entry fieldsare notoverused).
4.13 With data entry screens, the cursor is placed where the inputis needed.
4.14 Data formats are clearly indicated for input (e.g. dates) andoutput(e.g. unitsof values).
4.15 Users can complete simple tasks by entering just essentialinformation (with the system supplying the non-essentialinformationbydefault).
4.16 Forms allow users to stay with a single interaction methodfor as long as possible (i.e. users do not need to makenumerousshiftsfromkeyboardtomousetokeyboard).
4.17 Theusercanchangedefaultvaluesinformfields.
4.18 Text entry fields indicate the amount and the format of datathatneedstobeentered.
4.19 Formsarevalidatedbeforetheformissubmitted.
4.20 With data entry screens, the site carries out field-levelcheckingandform-levelcheckingattheappropriatetime.
4.21 The site makes it easy to correct errors (e.g. when a form isincomplete, positioning the cursor at the location where correctionis required).
4.22 Thereisconsistencybetweendataentryanddatadisplay.
4.23 Labels are close to the data entry fields (e.g. labels are rightjustified)

It should serve specific purpose effectively such as storing, recording,and retrieving the information. It ensures proper completion withaccuracy. It should be easy to fill and straightforward. It should focus onuser'sattention,consistency,andsimplicity.

Guidelinesfordesigningdialog boxes and errormessages

n the **message** of an **Error** alert **box**, explain what happened, the causeoftheproblem, andwhatthe usercando aboutit. Keep the **message** brief and use terms that are familiar to users. Ifappropriate, provide a Help button to open a separate onlinehelp**window**thatgivesbackgroundinformationabouttthe**error**.

Dialog boxes can be modal or modeless. A **modal dialog box**prevents users frominteracting with the application until the dialog box is dismissed. However, userscan move a modal dialog box and interact with other applications while the modaldialogbox isopen.Thisbehaviorissometimescalled"application-modal."

A **modeless dialog box**does not prevent users from interacting with theapplication they are in or with any other application. Users can go back and forthbetween amodelessdialogbox and other applicationwindows.

Use modeless dialog boxes whenever possible. The order in which usersperform tasks might vary, or users might want to check information in otherwindows before dismissing the dialog box. Users might also want to go back andforthbetweenthedialogboxand theprimary window.

Use modal dialog boxes when interaction with the application cannot proceedwhile the dialog box is displayed. For example, a progress dialog box that appearswhileyourapplication isloadingitsdatashouldbeamodaldialogbox.

A **dialog box**is a temporary, secondary window in which users perform a taskthat is supplemental to the task in the primary window. For example, a dialogboxmightenableusersto setpreferencesorchooseafilefromtheharddisk.Adialog box can contain panes and panels, text, graphics, controls (such ascheckboxes,radio buttons,orsliders),andoneormorecommand buttons.

Dialog boxes use the native window frame of the platform on which they arerunning.

Dialog BoxDesign

The following figure illustrates dialog box design guidelines for the Javalook and feel. The dialog box has a title in the window's title bar, a series ofuser interface elements, and a row of command buttons. The defaultcommand button is the OK button, indicated by its heavy border. Theunderlined letters are mnemonics, which remind users how to activatecomponents by pressing the Alt key and the appropriate character key. Thenoneditable Ruler Units combo box has initial keyboard focus, indicatingthattheuser'snextkeystrokeswilltake effectin thatcomponent.

Use the form "Application Name: Title" for the title of the dialog box (which isdisplayed inthetitlebar).

Includemnemonicsforalluserinterfaceelementsexceptthedefaultbuttonand theCancelbutton.

When opening a dialog box, provide initial keyboard focus to the componentthat you expect users to operate first. This focus is especially important for userswho must use a keyboard to navigate your application (for example, users withvisualandmobilityimpairments).

Consider the effect of internationalization on your design. Use a layoutmanager, which allows for text strings to become bigger or smaller when translatedto another language.

Formoreinformationoninternationalization,seePlanningforInternationalization and Localization. For details on keyboard support forAstrength of an iterative and incremental development process is that the resultsof a prior iteration can feed into the beginning of the next iteration (see Figure).Thus, subsequent analysis and design results are continually being refined andenhanced from prior implementation work. For example, when the code initeration N deviates from the design of iteration N (which it inevitably will), thefinal design based on the implementation can be input to the analysis and designmodels of iteration N navigating through dialog boxes, see Table 17. Forinformation on how to

capitalize text in dialog boxes, see Capitalization of Textinthe Interface.

Errormessage

Exceptionhandlinghasbeenignoredsofarinthedevelopmentofasolution. This was intentional to focus on the basic questions ofresponsibilityassignmentand objectdesign. However,inapplication development, it is wise to consider exception handlingduringdesignwork,andcertainlyduringimplementation.

Briefly, in the UML, exceptions are illustrated as asynchronousmessagesininteractiondiagram

Error handling refers to the routines in a program that respond toabnormal input or conditions. The quality of such routines is based onthe clarity of the **error** messages and the options given to users forresolvingtheproblem.

Take advantage of language specific semantics and represent whensomething exceptional has happened. **Exceptions** are thrown andcaught so the code can recover and **handle** the situation and not enteran **error** state. **Exceptions** can be thrown and caught so the applicationcanrecoverorcontinue gracefully.

Thetry**statement**allowsyouto definea blockofcodetobetested
for **errors** while it is being executed. The catch **statement** allows you todefinea blockofcodetobeexecuted,if an**error**occursinthetryblock.

Exception handling has been ignored so far in the development of asolution. This was intentional to focus on the basic questions ofresponsibility assignment and object design. However, in applicationdevelopment, it is wise to consider exception handling during designwork,andcertainlyduring implementation.

Briefly, in the UML, exceptions are illustrated as asynchronousmessagesininteractiondiagrams

Errors or mistakes in a program are often referred to as bugs. They arealmost always the fault of the programmer. The process of finding andeliminating errors is called debugging. Errors can be classified into threemajorgroups:

Syntaxerrors
Runtimeerrors
Logicalerrors

Error handling refers to the anticipation, detection, andresolution of programming, application, andcommunications**errors**. Specializedprograms, called**error** handlers,areavailableforsomeapplicationsAn example is the lack of sufficient memory to run an application oramemoryconflictwithanotherprogram.

CommandButtons:DesignGuidelines

1 Totherightofandtop-alignedwiththeother **control**.
2 Belowandleft-alignedwiththeother**control**.
3 Between **controls** that interoperate (such as Add andRemove**buttons**betweentwointeroperatinglistboxes)

clicking a command button immediately performs an action, such asopeninganothersurface.

- Use command buttons for user-initiated actions. That is,clicking a command button results in the immediateperformance of an action. Such actions might include closingor extending a surface, applying changes, opening a dialogbox,andsoon.
- When another control interoperates with a command button,such as the pairing of a text box and **Browse** button, denotethe relationship by placing the command button in one ofthreeplaces:
  - Totherightofandtop-alignedwiththeother control

- Belowandleft-alignedwiththeothercontrol
- Between controls that interoperate (suchas **Add** and **Remove** buttons between twointeroperatinglist boxes)

If multiple command buttons interoperate with thesame control, lay them out horizontally, left-alignedunderthecontrolortop-aligned toitsright. ina secondarywindow,right-alignthestandardwindowcommandbuttons(**OK**, **Cancel**, etc.) in a row along the bottom as shown. Do not use othertypes of controls besides command buttons in this row, and only usecommandbuttonsthataffectthe entirewindow.For example,
a **Help** command button should open Help about the entire window andnotaboutaspecificcontrolinthewindow.

Class Design In **Class Design** , the focus is on fleshing out the detailsof a particular **class** (for example, what operations and **classes** need tobe added to support, and how do they collaborate to support, theresponsibilities allocatedtothe**class**).

GUIDELINESFORDESIGININGAPPLICATIONWINDOWS

User interface is the front-end application view to which user interacts inorder to use the software. User can manipulate and control the softwareas well as hardware by means of user interface. Today, user interface isfound at almost every place where digital technology exists, right fromcomputers,mobilephones,cars,musicplayers, airplanes,shipsetc.

User interface is part of software and is designed such a way that it isexpectedtoprovidetheuserinsightofthesoftware.UIprovidesfundamentalplatfor m forhuman-computerinteraction.

UIcanbegraphical,text-based,audio-videobased,dependingupontheunderlying hardware and software combination. UI can be hardware orsoftwareoracombinationofboth.

Thesoftwarebecomesmorepopularifitsuserinterfaceis:

Window - An area where contents of application are displayed.Contents in a window can be displayed in the form of icons or lists, if thewindow represents file structure. It is easier for a user to navigate in thefile system in an exploring window. Windows can be minimized, resizedormaximizedtothesizeofscreen.Theycanbemovedanywhereonthe

screen. A window may contain another window of the same application,calledchildwindow.

GraphicalUserInterface

Graphical User Interface provides the user graphical means to interactwiththesystem.GUIcanbecombinationofbothhardwareandsoftware.Usin gGUI,userinterpretsthesoftware.

Typically,GUIismoreresourceconsumingthanthatofCLI.Withadvancing technology, the programmers and designers create complexGUI designsthatworkwithmoreefficiency,accuracyandspeed.

User InterfaceDesignActivities

There are a number of activities performed for designing user interface.TheprocessofGUIdesignandimplementationisalikeSDLC.Anymodelca n be used for GUI implementation among Waterfall, Iterative or SpiralModel.

A model used for GUI design and development should fulfill these GUIspecificsteps.

- **GUIRequirementGathering** -Thedesignersmayliketohavelistofallfunctional and non-functional requirements of GUI. This can betakenfrom userandtheirexistingsoftwaresolution.
- **UserAnalysis** -Thedesignerstudieswhoisgoingtousethesoftware GUI. The target audience matters as the design detailschange according to the knowledge and competency level of theuser.Ifuseristechnicalsavvy,advancedandcomplexGUIcanbe

incorporated. For a novice user, more information is included onhow-to ofsoftware.

- **Task Analysis** - Designers have to analyze what task is to be doneby the software solution. Here in GUI, it does not matter how it willbe done. Tasks can be represented in hierarchical manner takingone major task and dividing it further into smaller sub-tasks. TasksprovidegoalsforGUIpresentation.Flowofinformationamongsub-tasks determinesthe flowofGUIcontentsinthesoftware.
- **GUI Design & implementation** - Designers after having informationabout requirements, tasks and user environment, design the GUIand implements into code and embed the GUI with working ordummy software in the background. It is then self-tested by thedevelopers.
- **Testing**-GUItestingcanbedoneinvariousways.Organizationcanhave in-house inspection, direct involvement of users and releaseof beta version are few of them. Testing may include usability,compatibility, useracceptanceetc.

GUIImplementationTools

There are several tools available using which the designers can createentire GUI on a mouse click. Some tools can be embedded into thesoftwareenvironment(IDE).

GUI implementation tools provide powerful array of GUI controls. Forsoftwarecustomization,designerscanchangethecodeaccordingly.

TherearedifferentsegmentsofGUItoolsaccordingtotheirdifferentuseandplatform.

Example

MobileGUI,ComputerGUI,Touch-ScreenGUIetc.Hereisalistoffewtoolswhich

comehandyto buildGUI:

- FLUID
- AppInventor(Android)
- LucidChart
- Wavemaker
- VisualStudio

GenericGuidelines andStandards

Specific color guidelines and standards all differ with respect to the exactwordingoftheir requirements,butthegraphicproblemsthattheyaddress have a lot of overlap. Most fall into three general categories:discrimination and identification; luminance contrast; and generalproblemsofcolordesign.

These pages include our generic restatements of specific color guidelinesand standards that we have reviewed, with illustrations and links to therelatedinformationinourwebsite:GuidelinesaboutDiscriminationandIdentification of ColorsGuidelinesaboutLuminanceContrastGuidelinesaboutGeneralColorDesign

AboutColorGuidelinesandStandards

In this section we briefly discuss some of the reasons why we haveguidelines and standards. We also discuss and illustrate some problemsof guidelines and standards as a means for assurance of quality of colorusage, and suggest some research directions toward solving theproblems.

SpecificColorGuidelines andStandards

This page includes a listing of aerospace and non-aerospace guidelinesand standards that include colorusage**ABOUT COLOR GUIDELINESANDSTANDARDS**

WhydoweneedColorGuidelinesandStandards?

The consequences of poor color design are minimal in someapplications (e.g., personal webpages), but in others the consequencescan be life-threatening (e.g., flight critical displays in cockpits, air trafficcontrol displays, launch-control displays). When poor design hasimportant costs, the community needs some means of assuringsufficient usabilityandconformitytoconventions.

The only reliable way to assure quality of color design is to ensure thathumanfactorsengineeringisproperlyintegratedintothedesign,

development and testing of the product that gets deployed. This requiresthat:

- adequatespecializedcolorandhumanfactorsexpertiseisusedthroughoutt hedesignandtestingoftheapplication
- thoseexpertsaregivenadequatedesign authority,and
- resources (time and money) for design and testing are sufficient inspite of(nearly inevitable)budgetproblems andscheduledelays.

In this process color guidelines can provide the human factorsprofessionals with a checklist of the important usability concerns relatedto color. They can also be used to focus interactions with colorconsultants.

For applications that involve public safety there is sometimes anotherrole for standards. Standards can represent community agreement onuniform practices where they are needed. Universal use of green, yellowor amber, and red to label safety status, for example, allows users tounderstand their meaning in multiple environments with minimal need fortraining.

Proper integration of human factors expertise has not happened onsome projects, for a variety of cultural and practical reasons. In lieu ofthat expensive process, color guidelines and standards are sometimesexpected to provide a human factors shortcut, i.e., to allow graphicsdevelopers and design reviewers with limited knowledge of applied colorsciencetoavoidseriousmistakesincolordesign.

That's too much to expect from any document, regardless of size andquality. Color perception is complicated. Perception of even simplepatterns in the laboratory is only partly understood by color scientists.Color perception in natural work environments is far more complicatedand less understood, and information displays continue to get morecomplex. It's unreasonable to expect that any short training course orguidelines document can replace years of specialized professionaltraining and experience in applied color perception, color science andcolordesign.

The Status of Existing Color Guidelines andStandards

The degree of difficulty in getting effective colors for information displaydepends on the complexity of the graphics. Until recently most datadisplays in military and domestic command and control applicationswere very simple. When display quality, computational power, andcommunication bandwidth were expensive and scarce, the displayswere usually limited to stroke and alphanumeric graphics on uniformbackgrounds. For such simple displays it may be possible to describemost of the likely serious color design errors with a reasonably compactsetofstandards.

This situation has changed dramatically over the past several years.Inexpensive display systems are now capable of routinely computing,transmitting, and displaying graphics as complicated as dynamic maps,photos, and video. Products with advanced graphics are nowcommonplace in office and home environments. These capabilities arestarting to appear in command and control applications--in cars, planesand industrialcontrol settingswherepublicsafetyisanissue.

To safely exploit the full potential of these new capabilities we will needto adjust our approach to color standards. Conventional color standardsare limited with respect to complicated graphics. Most suffer from eithervagueness orover-restriction:

Some set desirable usability goals but are too abstract to provide usefulguidanceto designerswithlimitedappliedcolortrainingandexperience.

Forexample:

Thecontrastbetweentextanditsbackgroundshallbesufficientlyhightoensurereadabilityofthetext.
[Source:DODHCISGV2.0, 1992]FAAHFDS, p8-61.

"...In all cases the luminance contrast and/or color differences betweenall symbols, characters, lines, or all backgrounds shall be sufficient topreclude confusion or ambiguity as to information content of anydisplayedinformation."
Society of Automotive Engineers, AS8034 Minimum PerformanceStandards for Airborne Multipurpose Electronic Displays, SAE,Warrendale,Pennsylvania:1993,p.7.

These "performance-based" guidelines set excellent goals, but how aretheytobeachieved?

Other standards place constraints on specific graphic elements. Theseoften set parametric requirements that can be verified by eye orinstrument.

Forexample:

The minimum level of luminance for characters on a VDT, regardless ofwavelength,shallbe70cdperm2(20f-L).
NASA-STD-3000/T, 9.4.2.3.3.9 Visual Display Terminal DesignRequirements

Indicator Contrast - The luminance contrast within the indicator shall beat least50.0percent..
NASA-STD-3000/T,9.4.2.3.3.9DisplayContrastDesignRequirements

"An adequate contrast of at least 7:1 should be maintained betweenforeground and background colors to enhance color perception andperceivedimageresolution."
[Source:CTA,1996]FAAHumanFactorsDesignStandard,p 8-61.

These are more easily verified, but are generally too simple and rigid toallow optimal design solutions in complicated graphic contexts. Forexample, if all graphic elements are required to have high luminancecontrast the information density that can be achieved is unnecessarilylimitedbythe resultingclutter.Anotherexampleisuse ofblue:

Pure blue shall not be used on a dark background for text, thin lines, orhigh resolution information. [Source: DOE-HFAC 1, 1992] FAA HumanFactorsDesignStandard,p8-58

8.6.2.2.7Blue.Blueshould notbeusedastheforegroundcolorifresolutionoffinedetailsisrequired.

Color can set the basic mood, tone, concept, and connotation for abrand or product. Research conducted by the Institute for Color showsthat users take about 90 seconds to assess the quality of onlineproducts. From 62% to 90% of all product assessments that peoplemake are color-influenced on the subconscious level. It then follows thatchoosing the correct colors for your logo, brand, and product packagingshouldneverbe done onawhim.

Choosethe Right Color inthe ProperPattern

Different UI design colors signal various concepts to the senses. Ideally,youwanttochoosetherightoneattherighttime and inthe right pattern.They must be aimed toward the correct users, and you should choosethemtotargettheproper goals.

If your wish is to use color in UI design wisely, firstly understand colors'meaning, that they provoke the right emotions in your customers andhelptogetthedesired response.0

Let's start with emotions colors can bring and differences in itsperception:

he red refers to what we call warm colors. Those would be red, orange,and yellow. These warm colors bring about emotions having to do withwarmth and comfort. However, they might mean anger, hostility, orpassion to some individuals as well. The famous brand that uses red asmain company color to call for comfort and warm emotions is for surewell-knowCoca-Cola.

- Blue means security, trust, and safety. Numerous studies showthat blue has positive connotations for many different segments ofthe population. Blue is everywhere, including in many naturalsettings. Hundreds of prominent brands feature it, including SkypeandMicrosoftWord,andmentionedabovecompanies.
- Green brings calm feelings of renewal. These emotions fit well withsuch brand asTropicanathatusesgreen asmainlogocolor

And here, i suggest use single **color** scheme. **Color** contrast is also apractical method of **color** in **UI design**. Contrast can evoke variousemotionalresponsefromusers. **Colors**on oppositesidesof the **color** wheel can generate the strongest contrast, like black andwhite.

UI guidelines are common **design** concepts that are used to buildengaginganduniqueuserexperiences.Following these**guidelines** helps you to enhance usability and beauty of yourproducts. For more detailed rules of **UX guidelines**, refer to **UserInterfaceDesignGuidelines**

- Brightness
- Create dark color variants through increasing saturation and turningdown the brightness. If your product would be better served with brightercolorvariations,youmustdotheopposite.

  - Contrast

Contrast is another form of UI design that is considered to be practicalmethodology. This is what you would employ if you were trying to deviseasimpleinterface.It’saneasywaytoevokeemotionaluserresponses

GUIDELINESFORUSINGFONT

WhatFontShouldIUse?5
Principles for Choosing and UsingTypefaces

Formanybeginners,thetaskofpickingfonts isamystifying

process. There seem to be endless choices — from normal,conventional-lookingfontstonoveltycandycanefontsandbunnyfonts — with no way of understanding the options, only never-endinglistsofcategoriesandrecommendations.

Selecting the right typeface is a mixture of firm rules and looseintuition, and takes years of experience to develop a feeling for.Here are **five guidelines for picking and using fonts** that I’vedevelopedinthecourseof usingandteachingtypography

For better or for worse, picking a typeface is more like getting dressed inthe morning. Just as with clothing, there's a distinction between typefacesthat are expressive and stylish versus those that are useful and appropriateto many situations, and our job is to try to find the right balance for theoccasion. **While appropriateness isn't a sexy concept, it's the acid testthat shouldguideourchoiceof font.**

Most people only experience a font in its final context, whether they'rereading a billboard or typing a document. In this way, we experience fontsmuch like we experience a car—most of us are concerned mainly with howitdrivesandhowitlooks,butthere'sa lotgoing onunder thehood.

A font is software that is made up of both data and code, both of whichinteract with each other as well as the applications and operating system inwhich the font is installed. The data includes items like the contours ofglyphs, metrics such as advance widths or vertical spacing, or text stringssuchasfontfamilyandstylenames orcopyrightandtrademark strings.

The code can include aspects of the font like TrueType instructions orhinting, and the OpenType line layout code which helps 'shape' text toproduce the correct order and combination of glyphs from a set of inputstrings.

1 Typeface vs. Font

"Typeface"and"**Font**"meandifferentthings,

- **typeface**isthedesign(e.g.shape)ofacollectionofletters, numbers and symbols (also called glyphs),whereas
- **font**isaspecificsize,weight,orstyle(e.g.regular,bold,itali c)ofatypeface.

- 2.CommonFontCategories

- Typefaces come in all different shapes, and there is nosingleclassificationsystem.Belowareafewofthemostcomm

onlyseencategories.

- **2.1Serif**

- A “serif” is a small stroke attached to the ends of letters,givingthematraditionalfeel.The“serif”categoryincludesa few sub-categories such as Old Style, Classical, Neo-Classical,Transitional,Clarendon,etc.Thesetypefacesaremostlyusedinbooksandnewspapers.

```
body{
  font-family:'custom-font',fallback1,fallback2;
}
```

- Whereis thecustom-fontcoming from?Well,itcouldbe fromeitheryourOSifyouhaveitinstalled(e.g.SegoeUIonWindows,orRobotoonAndroid),or athirdpartysuchas Google Fonts, Adobe Fontsetc. In the later case, youmost likely need to tell the browser to download it byincluding the<link>tagintheheadof yourpage,likebelow

Mosttypefacescanbeclassifiedintooneof four basic groups: those with serifs,those without serifs, scripts anddecorativestyles.

U1designrule3

Rule 3: Allowing the users to be in Control of the software. (Application ofcorollary1)Someofthewaysto putusers incontrol are:Maketheinterfaceforgiving (errors should not cause serious effect) Make the interface visual(provide users with list of items) Provide immediate feedback (feedbackshouldbegivenforallchoices) AvoidmodesMaketheinterfaceconsistent

- *commandline(cli)*
- **graphicaluserinterface(GUI)**
- menudriven(**mdi**)

continuetoqualityassurance testing

A numberofschemesare usedfortestingpurposes Another important aspect is the fitness of purpose of a program that ascertainswhether the program serves the purpose which it aims for. The fitnessdefinesthesoftware**quality**.

Software QualityAssurance

SoftwareQuality

Schulmeyer and McManus have defined software quality as “the fitnessfor use of the total software product”. A good quality software doesexactlywhatitissupposedtodoandisinterpretedintermsofsatisfactionofthereq uirementspecificationlaiddownbythe user.

QualityAssurance

Software quality assurance is a methodology that determines the extentto which a software product is fit for use. The activities that are includedfordeterminingsoftwarequalityare−

- Auditing
- Developmentofstandardsandguidelines
- Productionofreports
- Reviewofqualitysystem

QualityFactors

- **Correctness**− Correctness determines whether the softwarerequirementsareappropriatelymet.
- **Usability**−Usabilitydetermineswhetherthesoftwarecanbeusedbydifferentc ategoriesofusers(beginners,non-technical,andexperts).
- **Portability** −Portabilitydetermineswhetherthesoftwarecanoperateindifferentplatform swithdifferent hardwaredevices.
- **Maintainability** −Maintainabilitydeterminestheeaseatwhicherrorscan becorrectedandmodulescan beupdated.

- **Reusability** − Reusability determines whether the modules andclassescanbereusedfordevelopingothersoftwareproducts.

- **SoftwareQualityAssurancePlan**
- Abbreviated as SQAP, the software quality assurance plancomprises of the procedures, techniques, and tools that areemployed to make sure that a product or service aligns with therequirements defined in the SRS(software requirementspecification).

Products must be tested in different ways, with different users anddifferent scenarios to make sure that the software that end-users receiveis aconsistent,high-qualityexperienceinarangeofsituations.

While testing and quality are inextricably linked, it’s important tounderstand that quality assurance testing and software testing aren’t onein the same. Part of quality assurance is finding a solution to thechallengeandimplementingit.

QA’sprocesstendstolooksomethinglikethis:

1. Generatingrequirements
2. Makingestimations
3. Developingaplan
4. Documentation
5. Day-to-daysprintexecution
6. Definingwhatneedstohappenbeforeaproductisconsidered “finished.”
7. Testing

Where the process was once defined by contracts, checklists, andcontrol,today’sQAteamisembeddedalongsidedevelopers.
Agile QA testing is less about performing the tests, and instead brings adeep understanding of the consumer into the fold, functioning as anadvocateformeeting expectations.

Conclusion

Onceaprogramcodeiswritten,itmustbetestedtodetectandsubsequently handle all errors in it. A number of schemes are used fortestingpurposes.

Another important aspect is the fitness of purpose of a program thatascertainswhethertheprogramservesthepurposewhichitaimsfor.Thefitness definesthesoftwarequality.

TestingObject-OrientedSystems

Testing is a continuous activity during software development. In object-orientedsystems,testingencompassesthreelevels,namely,unittesting,subsystemtesting,andsystemtesting.

UnitTesting

In unit testing, the individual classes are tested. It is seen whether theclassattributesareimplementedasperdesignandwhetherthemethodsand the interfaces are error-free. Unit testing is the responsibility of theapplicationengineerwhoimplementsthe structure.

SubsystemTesting

This involves testing a particular module or a subsystem and is theresponsibility of the subsystem lead. It involves testing the associationswithinthesubsystemaswellastheinteractionofthesubsystemwiththeoutside.Subsystemtestscanbeusedasregressiontestsforeachnewlyreleasedversionofthesubsystem

softwareQuality

SchulmeyerandMcManushavedefinedsoftware qualityas"thefitness for use of the total software product". A good qualitysoftwaredoesexactlywhatitissupposedtodoandisinterpretedintermsofsatisfactionoftherequirementspecificationlaiddownbytheuser.

QualityAssurance

Software quality assurance is a methodology that determines the extentto which a software product is fit for use. The activities that are

includedfordeterminingsoftwarequalityare−

UNIT V

SOFTWARE QUALITY

INTRODUCTION:

Schulmeyer and McManus have defined software quality as “the fitness for use of the total software product”. A good quality software does exactly what it is supposed to do and is interpreted in terms of satisfaction of the requirement specification laid down by the user

QUALITY ASSURANCE TESTS:

Software quality assurance is a methodology that determines the extent towhich a software product is fit for use. The activities that are included fordetermining software quality are −

- Auditing
- Development of standards and guidelines
- Production of reports
- Review of quality system

Quality Factors:

- Correctness − Correctness determines whether the software requirements are appropriately met.
- Usability − Usability determines whether the software can be used bydifferent categories of users (beginners, non-technical, and experts).
- Portability − Portability determines whether the software can operate indifferent platforms with different hardware devices.
- Maintainability − Maintainability determines the ease at which errors can becorrected and modules can be updated.
- Reusability − Reusability determines whether the modules and classes can bereused for developing other software products.

TESTING STRATEGIES:

BLACK BOX TESTING:

Black box testing involves testing a system with no prior knowledge of its

internal workings. A tester provides an input, and observes the output generated by the system under test. This makes it possible to identify how the system responds to expected and unexpected user actions, its response time, usability issues and reliability issues.

Black box testing is a powerful testing technique because it exercises a system end-to-end. Just like end-users "don't care" how a system is coded or architected, and expect to receive an appropriate response to their requests, a tester can simulate user activity and see if the system delivers on its promises. Along the way, a black box test evaluates all relevant subsystems, including UI/UX, web server or application server, database, dependencies, and integrated systems.

An example of a security technology that performs black box testing is Dynamic Application Security Testing (DAST), which tests products in staging or production and provides feedback on compliance and security issues.

Black box testing can be done in following ways:

1. Syntax Driven Testing – This type of testing is applied to systems that can be syntactically represented by some language. For example- compilers,language that can be represented by context free grammar. In this, the test cases are generated so that each grammar rule is used at least once.

2. Equivalence partitioning– It is often seen that many type of inputs work similarly so instead of giving all of them separately we can group them together and test only one input of each group. The idea is to partition the input domain of the system into a number of equivalence classes such that each member of class works in a similar way, i.e., if a test case in one class results in some error, other members of class would also result into same error.

The technique involves two steps:

1. **Identification of equivalence class –** Partition any input domain into minimum two sets: **valid values** and **invalid values**. For example, if the valid range is 0 to 100 then select one valid input like 49 and one invalid like 104.
2. Generating test cases –
(i) To each valid and invalid class of input assign unique identification number.
(ii) Write test case covering all valid and invalid test case considering that no two invalid inputs mask each other.

Black Box **Testing** is a software **testing** method in which the internal structure/ design/ implementation of the item being **tested** is not known to the **tester**.

White Box Testing is a software **testing** method in which the internal structure/ design/ implementation of the item

White box testing:

White box testing is a **testing** technique, that examines the program structure and derives **test** data from the program logic/code. The other names of glass **box testing** are clear **box testing**, open **box testing**, logic driven **testing** or path driven **testing** or structural **testing**.

g **tested** is known to the **tester**

White Box Testing Techniques:

- **Statement Coverage -** This technique is aimed at exercising all programming statements with minimal tests.
- **Branch Coverage -** This technique is running a series of tests to ensure that all branches are tested at least once
- **Path Coverage -** This technique corresponds to testing all possible paths

Advantages of White Box Testing:

• Forces test developer to reason carefully about implementation.

• Reveals errors in "hidden" code.

• Spots the Dead Code or other issues with respect to best programming practices.

Disadvantages of White Box Testing:

• Expensive as one has to spend both time and money to perform white box testing.

• Every possibility that few lines of code are missed accidentally.

• In-depth knowledge about the programming language is necessary to perform white box testing.

• which means that each statement and branch is covered.

• **White Box Testing** is software testing technique in which internal structure, design and coding of software are tested to verify flow of input-output and to improve design, usability and security. In white box testing, code is visible to testers so it is also called Clear box testing, Open box testing, Transparent box testing, Code-based testing and Glass box testing.

• It is one of two parts of the Box Testing approach to software testing. Its counterpart, Blackbox testing, involves testing from an external or end-user type perspective. On the other hand, White box testing in software engineering is based on the inner workings of an application and revolves around internal testing.

• The term "WhiteBox" was used because of the see-through box concept. The clear box or WhiteBox name symbolizes the ability to see through the software's outer shell (or "box") into its inner workings. Likewise, the "black box" in "Black Box Testing" symbolizes not being able to see the inner workings of the software so that only the end-user experience can be tested.

White box testing techniques analyze the internal structures the used data structures, internal design, code structure and the working of the software rather than just the functionality as in black box testing. It is also called glass box testing or clear box testing or structural testing.

Working process of white box testing:

• **Input**: Requirements, Functional specifications, design documents, source code.
• **Processing:** Performing risk analysis for guiding through the entire process.
• **Proper test planning:** Designing test cases so as to cover entire code. Execute rinse-repeat until error-free software is reached. Also, the results are communicated.
• **Output**: Preparing final report of the entire testing process.

Testing techniques:

Statement coverage: In this technique, the aim is to traverse all statement at least once. Hence, each line of code is tested. In case of a flowchart, every node must be traversed at least once. Since all lines of code are covered, helps in pointing out faulty code.
White box testing refers to a scenario where (as opposed to black box testing), the tester deeply understands the inner workings of the system or system component being tested.

Gaining a deep understanding of the system or component is possible when the tester understands these at program- or code-level. So almost all the time, the tester needs to either understand or have access to the source code that makes up the system – usually in the form of specification documents.
Armed with the level of technical detail that is normally visible only to a developer, a Tester will then be able to design and execute test cases that cover all possible scenarios and conditions that the system component is designed to handle.
We'll see how this is done in our example later.
By performing testing at the most granular level of the system, you are able to build a robust system that works exactly as expected, and ensure it will not throw up any surprises whatsever.
The key principles that assist you in executing white box tests successfully are:
Statement Coverage – ensure every single line of code is tested. White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) is a method of software testing that tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing). In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the expected outputs. This is analogous to testing nodes in a circuit, e.g. in-circuit testing (ICT). White-box testing can be applied at the unit, integration and system levels of the software testing process. Although traditional testers tended to think of white-box testing as being done at the unit level, it is used for integration and system testing more frequently today. It can test paths within a unit, paths between units during integration, and between subsystems during a system–level test. Though this method of test design can uncover many errors or problems, it has the potential to miss unimplemented parts of the specification or missing requirements. Where white-box testing is design-driven,[1] that is, driven exclusively by agreed specifications of howeach component of software is required to behave (as in DO-178C and ISO 26262 processes) then white-box test techniques can accomplish assessment for unimplemented or missing requirements.
White-box test design techniques include the following code coverage criteria:

- Control flow testing
- Data flow testing
- Branch testing
- Statement coverage
- Decision coverage
- Modified condition/decision coverage
- Prime path testing

• Path testing

If we go by the definition, "White box testing" (also known as clear, glass box or structural testing) is a testing technique which evaluates the code and the internal structure of a program.
White box testing involves looking at the structure of the code. When you know the internal structure of a product, tests can be conducted to ensure that the internal operations performed according to the specification. And all internal components have been adequately exercised.

TOP DOWN TESTING:

Top-down **integration testing** is an **integration testing** technique used in order to simulate the behaviour of the lower-level modules that are not yet integrated. Stubs are the modules that act as temporary replacement for a called module and give the same output as that of the actual product.

Top-down testing is a type of incremental integration testing approach in which testing is done by integrating or joining two or more modules by moving down from top to bottom through control flow of architecture structure. In these, high-level modules are tested first, and then low-level modules are tested. Then, finally, integration is done to ensure that system is working properly. Stubs and drivers are used to carry out this project. This technique is used to increase or stimulate behavior of Modules that are not integrated into a lower level.

Processing :

Following are the steps that are needed to be followed during processing 1Upper models or high levels modules should be tested properly to maintain quality and for processing lower-level modules or stubs of top-down testing.

1. As we know that in this type of pf testing, stubs temporarily replace a lower-level module, but data do not move upwards with this replacement. Due to this, testing cannot be done on time which results in delays in testing.
2. Due to replacement, stubs might become more and more complex after each replacement.
3. Losing control over correspondence between specific tests and specific modules is main problem that might arises during process.

4. Sometimes, modules at lower levels are tested inadequately (unsatisfactory that lacks quality)

Advantages :

• There is no need to write drivers.
• Interface errors are identified at an early stage and fault localization is also easier.
• Low-level utilities that are not important are not tested well and high-level testers are tested well in an appropriate manner.
• Representation of test cases is easier and simple once Input-Output functions are added

Disadvantages :

• It requires a lot of stubs and mock objects.
• Representation of test cases in stubs can be not easy and might be difficult before Input-Output functions are added.
• Low-level utilities that are important are also not tested well.
Top Down --> BIG SYSTEM to smaller components
Top-down approach is simple and not data intensive

Bottom up testing:

A type of integration **testing**, **bottom-up** approach is a **testing** strategy in which the modules at the lower level are **tested** with higher modules until all the modules and aspects of the software are **tested** properly. This approach is also known as inductive reasoning, and in many cases is used as a synonym of synthesis.

Bottom-Up approach is an immensely beneficial approach, used more often than its counterpart and the well-known testing technique, **top down approach**. This approach of integration testing is utilized when off-the-shelf or existing components are selected and integrated into the product.

Advantages:

It is appropriate for applications where bottom-up design methodology is used.

- Test conditions can be created easily.

- If the low level modules and their combined functions are often invoked by other modules, then it is more useful to test them first so that meaningful effective integration of other modules can be done.
- Always starting at the bottom of the hierarchy again means that the critical modules are generally built and tested first and therefore any errors or mistakes in these forms of modules are identified early in the process.
- Advantageous if major flaws occur towards the bottom of the program.

Disadvantage:

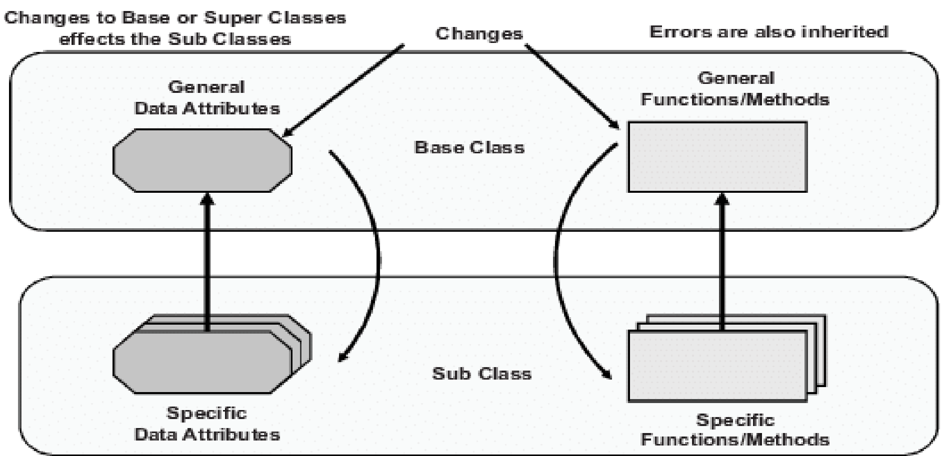
- Test engineers cannot observe system level functions from a partly integrated system.
- They cannot observe the system level functions until the top level test driver is in place.
- The program as an entity does not exist until the last module is added.
- One big disadvantage of bottom up strategy is that, in this sort of testing no working model can be represented as far as several modules have been built.
- This approach is driven by the existing infrastructure instead of the business processes.

Impact of object orientation on testing:

In the **object-oriented** model, interaction errors can be uncovered by scenario-based **testing**. This form of **Object oriented-testing** can only **test** against the client's specifications, so interface errors are still missed. Class **Testing** Based on Method **Testing**: This approach is the simplest approach to **test** classes.

Impact of inheritance in testing:

If all the methods of a class are assumed to be equally complex, then a class with more methods is more complex and thus more susceptible to errors. **Inheritance** Structure − Systems with several small **inheritance** lattices are more well–structured than systems with a single large **inheritance** lattice.



Derived Classes are dependent on the Base Classes

Object-oriented Software Systems present a particular challenge to the software testing community. This review of the problem points out the particular aspects of object-oriented systems which makes it costly to test them. The flexibility and reusability of such systems is described from the negative side which implies that there are many ways to u...

Software typically undergoes many levels of testing, from unit testing to system or acceptance testing. Typically, in-unit testing, small “units”, or

modules of the software, are tested separately with focus on testing the code of that module. In higher, order testing (e.g, acceptance testing), the entire system (or a subsystem) is tested with the focus on testing the functionality or external behavior of the system.

As information systems are becoming more complex, the object-oriented paradigm is gaining popularity because of its benefits in analysis, design, and coding. Conventional testing methods cannot be applied for testing classes because of problems involved in testing classes, abstract classes, inheritance, dynamic binding, message, passing, polymorphism, concurrency, etc.

Testing classes is a fundamentally different problem than testing functions. A function (or a procedure) has a clearly defined input-output behavior, while a class does not have an input-output behavior specification. We can test a method of a class using approaches for testing functions, but we cannot test the class using these.

REUSABILITY OF TESTS:

Re-usability in software testing can save you the time and effort from repetitive tasks of test creation and its execution. You can easily achieve test and application stability by leveraging the work you have already done in the past, with a little smartness upfront.

In today's fast-paced world where there is a constant need to build software effectively, all disciplines are constantly on the look-out to maximize their productivity and minimize re-work. One of the buzz words amongst all of them is "Re-usability".

In computer science and software engineering, **reusability** is the use of existing assets in some form within the software product development process; these assets are products and by-products of the software development life cycle and include code, software components, test suites, designs and documentation.

Reusability in OOP achieves through the features of C++ where it possible to extend or reuse the properties of parent class or super class or base class in a subclass and in addition to that, adding extra more features or data members in the subclass or child class or derived class.

Introduction. **Software reuse** is the process of implementing or updating **software** systems using existing **software** components. A good **software reuse** process facilitates the increase of productivity, quality, and reliability, and the decrease of costs and implementation time.

Reusability is a major contributor to the development goal of achieving high speed, low cost and quality. Usually, it is better and faster to employ proven off-the-shelf designs rather than specially-crafted designs that might have problems.

The concept of **reusability** is widely used in order to reduce cost, effort, and time of **software development**. **Reusability** also increases the productivity, maintainability, portability, and reliability of the **software** products. That is the **reusable software** components are evaluated several times in other systems before.

GUIDELINES FOR DEVELOPING QUALITY ASSURANCE TEST CASES:

quality assurance of methods and processes is a prerequisite for delivering quality statistics. The scope of the guidelines under development is the quality assurance of new or redesigned methods and processes.

State the test cases for developing quality assurance

.1 First analyze, what are the features and services our test attempts to cover.
.2 Test object's interactions and the messages sent among them.
.3 Boundary conditions need to be test.
.4 Specify the methods which are included to testing.
.5 Test the normal use of the object's methods.
.6 Test the abnormal use of the object's methods, but the abnormal to be reasonable.
.7 Test the abnormal use of the object's methods, but the abnormal to be unreasonable.
.8 Document the cases, when the revisions have been made.
.9 Our test case is based on use case then we can refer it with use-case name.
.10 Assess the internal quality, reusability and extendability of software.

Once a program code is composed, it must be tried to test hence handle all blunders in it. Various plans are utilized for testing purposes.Another vital viewpoint is the wellness of motivation behind a program that finds out whether the program fills the need which it goes for. The wellness characterizes the product quality.

Software quality assurance (SQA) is a process which assures that all software engineering processes, methods, activities and work items are

monitored and comply against the defined standards. These defined standards could be one or a combination of any like ISO 9000, CMMI model, ISO15504, etc. SQA incorporates all software development processes starting from defining requirements to coding until release. Its prime goal is to ensure quality.

GUIDELINES FOR DEVELOPING TEST PLANS:

- Analyze the Product. The first step towards creating a **test plan** is to analyze the product, its features and functionalities to gain a deeper understanding. ...
- **Develop Test** Strategy…..
- Define Scope. ...
- **Develop** a **Schedule**. ...
- Define Roles and Responsibilities. ...
- Anticipate Risks

The important points that should be under consideration during test includes Scope of the testing, **money**, timeline, Risk **analysis** etc. A Good Test plan ensures less hurdles during execution phase and helps it in making smoother.

Test plan can be defined as a document for a software project which defines the approach, scope, and intensity on the effort of software **testing**. The **test strategy** is a set of instructions or protocols which explain the **test** design and determine how the **test** should be performed.

Reliability requirements for test are derived several sources, typically described in Supplementary Specifications, User-Interface Guidelines, Design Guidelines, and Programming Guidelines.

Review these artifacts and pay especial attention to statements that include the following:

- statements reliability or resistance to failure, run-time errors (such as memory leaks)
- statements indicating code integrity and structure (compliance to language and syntax)
- statements regarding resource usage

At least one requirement for test should be derived from each statement in the artifacts that reflects information listed above.

Successful testing requires that the test effort successfully balance factors such as resource constraints and risks. To accomplish this, the test effort should be prioritized so that the most important, significant, or riskiest use cases or components are tested first. To prioritize the test effort, a risk assessment and operational profile are performed and used as the basis for establishing the test priority

The following sections describe how to determine test priority.

Continuous testing:

Continuous Testing is the process of executing automated **tests** as part of the software delivery pipeline in order to obtain feedback on the business risks associated with a software release candidate as rapidly as possible.

Continuous Testing in DevOps is a software testing type that involves testing the software at every stage of the software development life cycle. The goal of Continuous testing is evaluating the quality of software at every step of the Continuous Delivery Process by testing early and testing often.

The Continuous Testing process in DevOps involves stakeholders like Developer, DevOps, QA and Operational system.

Continuous testing is the process of checking if the software meets the business needs and that there is no risk involved in the release of the software. Continuous testing provides a safety net to the entire software development and deployment by ensuring that there are no untoward incidents.

Unlike the old times, today continuous testing is embedded in the development process of the software development lifecycle. It is an integrated software delivery pipeline which more and more enterprises today are implementing in order to get the best out of their software. The only requirement of the continuous testing is that the entire environment has to be stable and there should be valid test data for every test iteration.

It is about facilitating the pipeline so that the right set of tests is executed at the right time in the delivery pipeline so that there are no roadblocks. The entire framework of CT is based on getting the actionable feedback for the various

stages. The entire layer of the modern software architecture is being evaluated with the Continuous Testing at the right stages of the delivery pipeline.

DEBUGGING PRINCIPLES:

The conceptual framework of object–oriented systems is based upon the object model. There are two categories of elements in an object-oriented system –

Major Elements − By major, it is meant that if a model does not have any one of these elements, it ceases to be object oriented. The four major elements are –

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

Minor Elements − By minor, it is meant that these elements are useful, but not indispensable part of the object model. The three minor elements are –

- Typing
- Concurrency
- Persistence
- Structure. A good **argument** must meet the fundamental structural requirements of a well-formed **argument**. ...
- Relevance. ...
- Acceptability. ...
- Sufficiency. ...
- Rebuttal.

Object-Oriented Principles. **Encapsulation**, **inheritance**, and **polymorphism** are usually given as the three fundamental principles of object-oriented languages (OOLs) and object-oriented methodology. These principles depend somewhat on the type of the language.

- Single Responsibility principle.
- Open/Closed principle.
- Liskov Substitution principle.

• Interface Segregation principle.
• Dependency Inversion principle.

SYSTEM USABILITY AND MEASURING USER SATISFACTION

User satisfaction test• Process of quantifying usability test with measurable attributes of the test such as functionality, cost or ease of use – Ex: 90 5 of the people should know to withdraw money from ATM without training or error.

Usability should be a subset of s/w quality characteristics. This means usability must be place at same level as other characteristics such as reliability, correctness and maintainability o Usability testing deal with how well the interface of s/w fits the use cases, which are reflections of users' needs and expectations. To ensure user satisfaction, we must measure it throughout system development with user satisfaction tests. It forms comm., vehicle between designers and end users Usability Testing: - o ISO defines usability as effectiveness, efficiency and satisfaction with which a specified set of users can achieve a specified set of tasks in particular environments. It requires i. Defining tasks. What are the tasks? ii. Defining users. Who are the users? iii. A means for measuring effectiveness, efficiency and satisfaction. How do we measure usability? o Usability testing measures the ease of use as well as degree of comfort and satisfaction users have with the s/w. Usability test cases begin with identification of use cases that can specify the target audience, tasks and test goals. When designing test, focus on use cases or tasks The main advantage is that all design traces directly back to user requirements.Use cases and usage scenarios can become test scenarios; and therefore, the use case will drive usability, user satisfaction & quality assurance test cases p Guidelines for developing usability testing Usability testing should include all of a s/w's components Usability testing need not be very expensive or elaborate All tests need not involve many subjects. Typically, quick, iterative tests with small, welltargeted sample of 6 – 10 participants can identify 80 – 90 percent of most design problems User's experience also as part of s/w usability. 80 – 90 percent of most design problems can be studied with target few users of single skill level of users, such as novices or intermediate level Apply usability testing early and often

Recording the Usability Test: A quiet location, free from distractions environment is best for conducting test and intervention yields better results. It is done with test data along with guides or hints around a problem. Always

records techniques & search patterns users employ when attempting to work though a difficulty & number and type of hints provided to them

USABILTY TESTING

Usability testing, a non-functional **testing** technique that is a measure of how easily the system can be used by end users. It is difficult to evaluate and measure but can be evaluated based on the below parameters: Level of Skill required to learn/use the software. ... Time required to get used to in using the software.

Usability Testing is a type of testing, that is done from an end user's perspective to determine if the system is easily usable. Usability testing is generally the practice of testing how to easy design is to use on a group of representative users. A very common mistake in usability testing is conducting a study too late in the design process and If you wait until right before your product is released, you won't have the time or money to fix any issues – and you'll have wasted a lot of effort developing your product the wrong way.

Needs of Usability Testing:

Usability testing provides some benefits and the main benefits and purpose of usability testing are to identify usability problems with a design as early as possible, so they can be fixed before the design is implemented or mass produced and then such, usability testing is often conducted on prototypes rather than finished products, with different levels of fidelity depending on the development phase.

Phases of usability testing:

There are five phases in usability testing which are followed by the system when usability testing is performed.These are given below:

1 **Prepare your product or design to test:**

In the first phase of usability testing is choosing a product and then prepare the product for usability testing. For usability testing more function and operation are required then this phase provided that type of requirement. Hence a this is one of most important phase in usability testing.

2 **Write a test plan:**

This is the third phases of usability testing. The plan is one of the first steps in each round of usability testing is to develop a plan for the test. The main purpose of the plan is to document what you are going to do, how you are going to conduct the test, what metrics you are going to find, the

number of participants you are going to test, and what scenarios you will use.

GIDELINES FOR DEVELOPING USABILITY TESTING

Usabilityis the glue that sticks your user to your web and mobile designs. Well executed user interface (UI) design does not mean much if your users do not know how to engage with it. It needs to be usable, useful and credible as well as desirable. If the UX-factor is not enough of an incentive, think of it this way: optimising your *usability* pays off. A usable web or mobile application can boost conversion rates, lower support costs and reduce design and development rework.

Luckily, it is possible to optimise your *usability*. By performing a *usability test*, you can assess how easily your users can connect with your UI. These studies help you adjust your design strategy to benefit the user and encourage them to return to your website and convert.

A *usability test* is how UX researchers evaluate how easy or difficult a task is to complete. In *web design*, *usability* research involves evaluating the way a user interacts with the UI, by observing and listening to users complete typical tasks such as completing a purchase or subscribing to a newsletter.

Studies are performed early on during the design process so that errors can be corrected as soon as possible and do not affect the fabric of the final product. In the worstcase scenario, you will have plenty of time to start over and improve the overall user experience.

Usability tests can be split into two categories:

• **Qualitative tests**, or 'qual' research: These involve direct observation and assessment of how *test* participants engage with specific UI elements to determine which components are problematic.

• **Quantitative tests**, or 'quant' research: These consist of an indirect assessment of the *UI design*, either based on participants' performance of tasks or their perception of *usability* (e.g. a survey or poll).

For a *usability test*, it is recommended that both quantitative and qualitative data is gathered.

Once the *usability test* is complete, the UX team will go back to the drawing board, or indeed wireframe or prototype, and correct the *usability* errors based on the participants' behaviour and interactions.

RECORDING THE USEABILITY TEST

One of the most important parts of conducting usability tests is communicating the results. This means you should share the most valuable findings in ways that people can easily understand. One way to do this is to share recordings of the actual tests. There aren't many requirements for recording other than having access to a device that can record the study and asking your participants if you can record them.

- Recruiting unsuitable participants. ...
- Not **testing** early and often **during** the project lifecycle. ...
- Following too rigid a **test** plan. ...
- Not rehearsing your setup. ...
- Using a **one**-way mirror. ...
- Not meeting participants in reception. ...
- Asking leading questions. ...
- Interrupting the participant.

SATISFACTION TEST

User **satisfaction testing**:

It is the process of quantifying the usability **test** with some measurable attributes of the **test**, such as functionality, cost, or ease of use. User **satisfaction** cycle: Create a user **satisfaction test for** your own project. Conduct **test** regularly and frequently.

User satisfaction testing: It is the process of quantifying the usability test with some measurable attributes of the test, such as functionality, cost, or ease of use. User satisfaction cycle:

- Create a user satisfaction test for your own project
- Conduct test regularly and frequently
- Read the comments very carefully, especially if they express a strong feeling.
- Use the information from user satisfaction test, usability test, reactions to prototypes, interviews recorded, and other comments to improve the product.

Important benefit of user satisfaction testing is you can continue using it even after the product is delivered.

•Format of every user satisfaction test is basically the same, but its context's different for each project.
• Use cases provide excellent source of information throughout this process.
• Work with users (or) clients to find out what attributes should be included in the test.
• Ask the users to select limited numbers (5 to 10) of attributes by which the final product can be evaluated.

Principal objectives of the user satisfaction:

1. As a communication vehicle between designers as well as between users and designers.
2. To detect and evaluate changes during theDEBUGGING design process.
3. To provide a periodic indication of divergence of opinion about the current design.
4. To provide a periodic indication of divergence of opinion about the current design.
5. To enable pinpointing specific areas of dissatisfaction for remedy.
6. To provide a clear understanding of just how the completed design is to be evaluated.

EXAMPLE
Steps:
- Develop test objectives.
- Develop test cases.
- Analyze the tests.

GUIDELINES FOR DEVELOPING USER SATISFACTION TESTING

User satisfaction testing:

Conduct **test** regularly and frequently. Read the comments very carefully, especially if they express a strong feeling. Use the information from **user satisfaction test**, usability **test**, reactions to prototypes, interviews recorded, and other comments to improve the product

User satisfaction test Process of quantifying usability **test** with measurable attributes of the **test** such as functionality, cost or ease of use – Ex: 90 5 of the people should know to withdraw money from ATM without training or error

7. A TOOL FOR ANALYZING USER SATISFACTION

Usability testing, a non-functional testing technique that is a measure of how easily the system can be used by end users. It is difficult to evaluate and measure but can be evaluated based on the below parameters: Level of Skill required to learn/use the software. ... Time required to get used to in using the software.

- Guerilla**testing**. ...
- Lab **usability testing**. ...
- Unmoderated remote **usability testing**. ...
- Contextual inquiry. ...
- Phone interview. ...
- Card sorting. ...
- Session recording.
- Usability testing permeates product development. ...
- Usability testing involves studying real users as they use the product. ...
- Usability testing involves setting measurable goals anddetermining whether the product meets them.

******