



DEPARTMENT OF CSE
CS8592 - OBJECT ORIENTED ANALYSIS AND DESIGN

UNIT I - UNIFIED PROCESS AND USE CASE DIAGRAMS

Introduction to OOAD with OO Basics – Unified Process – UML diagrams – Use Case – Case study – the Next Gen POS system, Inception - Use case Modelling – Relating Use cases – include, extend and generalization – When to use Use-cases

INTRODUCTION TO OOAD WITH OO BASICS

Object Oriented analysis and design skills are essential for the creation of well-designed, robust, and maintainable software using OO technologies and languages such as Java or C#.

Requirements analysis and OOA/D needs to be presented and practiced in the context of some development process. In this case, an **agile** (light, flexible) approach to the well-known **Unified Process** (UP) is used as the sample **iterative development process** are taken. It includes

- Apply principles and patterns to create better object designs.
- Iteratively follow a set of common activities in analysis and design, based on an agile approach to the UP as an example.
- Create frequently used diagrams in the UML notation.

What is Analysis and Design?

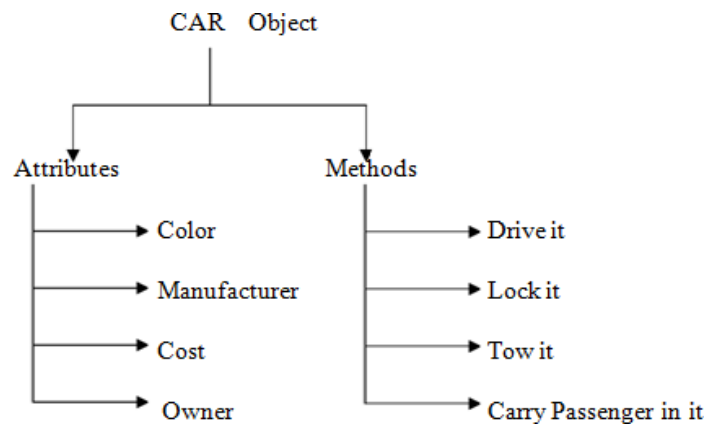
Analysis (do the right thing) emphasizes an investigation of the problem and requirements, rather than a solution. For example, if a new online trading system is desired, how will it be used? What are its functions?

Design (do the thing right) emphasizes a conceptual solution (in software and hardware) that fulfills the requirements, rather than its implementation. For example, a description of a database schema and software objects. Design ideas often exclude low-level or "obvious" detail.

OO Basics

Object: A car is an object a real-world entity, identifiably separate from its surroundings. A

car has a well-defined set of attributes in relation to other object.



Attributes: Data of an object. ,Properties of an object.

Methods: Procedures of an object. or Behavior of an object.

The term object was for formal utilized in the similar language. The term object means a combination or data and logic that represent some real-world entity.

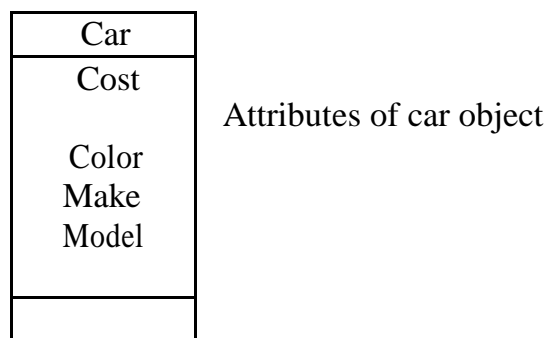
When developing an object oriented applications, two basic questions arise,

1. What objects does the application need?
2. What functionality should those objects have?

Programming in an object-oriented system consists of adding new kind of objects to the system and defining how they behave. The new object classes can be built from the objects supplied by the object-oriented system.

Object state and properties (Attributes)

Properties represent the state of an object. In an object oriented methods we want to refer to the description of these properties rather than how they are represented in a particular programming language.



We could represent each property in several ways in a programming languages.

For example: Color

1. Can be declared as character to store sequence or character [ex: red, blue, ..]
2. Can declared as number to store the stock number of paint [ex: red paint, blue paint, ..]

3. Can be declared as image (or) video file to refer a full color video image.

The importance of this distinction is that an object abstract state can be independent of its physical representation.

Object Behavior and Methods:

We can describe the set of things that an object can do on its own (or) we can do with it. For example: Consider an object car, We can drive the car. We can stop the car.

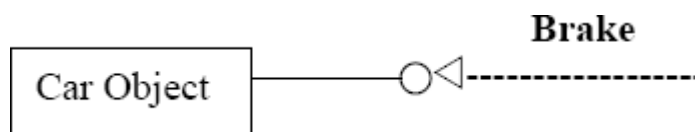
Each of the above statements is a description of the objects behavior. The objects behavior is described in methods or procedures. A method is a function or procedures that is defined in a class and typically can access to perform some operation. Behavior denotes the collection of methods that abstractly describes what an object is capable of doing. The object which operates on the method is called receiver. Methods encapsulate the behavior or the object, provide interface to the object and hide any of the internal structures and states maintained by the object. The procedures provide us the means to communicate with an object and access it properties.

For example: An employee object knows how to compute salary. To compute an employee salary, all that is required is to send the compute payroll message to the employee object.

Objects Respond to Messages: The capability of an object's is determined by the methods defined for it. To do an operation, a message is sent to an object. Objects represented to messages according to the methods defined in its class.

For example:

When we press on the brake pedal of a car, we send a stop message to the car object. The car object knows how to respond to the stop message since brake have been designed with specialized parts such as brake pads and drums precisely respond to that message.

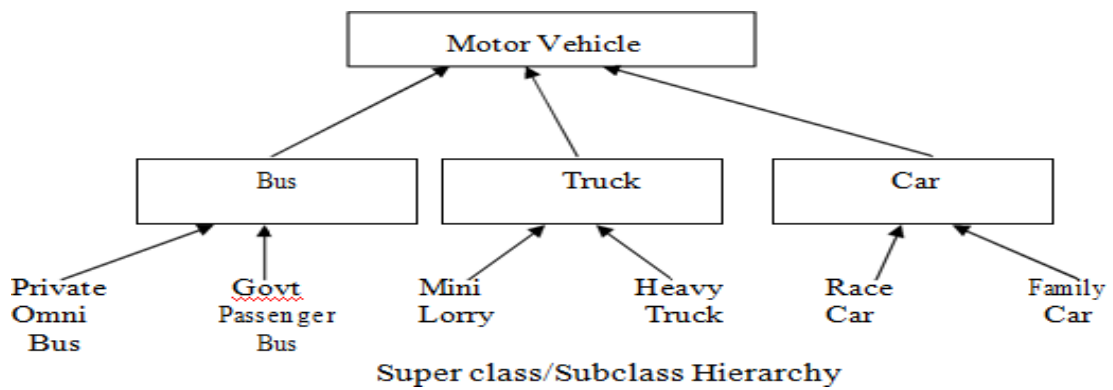


Different object can respond to the same message in different ways. The car, motorcycle and bicycle will all respond to a stop message, but the actual operations performed are object specific.

It is the receiver's responsibility to respond to a message in an appropriate manner. This gives the great deal of flexibility, since different object can respond to the same message in different ways. This is known as polymorphism.

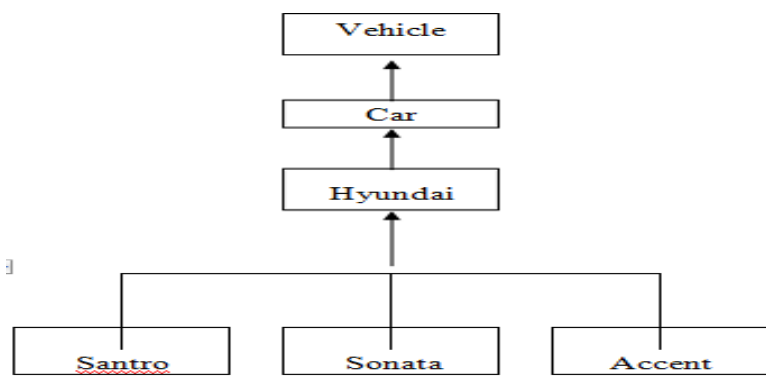
Class Hierarchy

An object-oriented system organizes classes into a subclass super class hierarchy.



The properties and behaviors are used as the basis for making distinctions between classes are at the top and more specific are at the bottom of the class hierarchy. The family car is the subclass of car. A subclass inherits all the properties and methods defined in its super class.

Inheritance: It is the property of object-oriented systems that allow objects to be built from other objects. Inheritance allows explicitly taking advantage of the commonality of objects when constructing new classes. Inheritance is a relationship between classes where one class is the parent class of another (derived) class. The derived class holds the properties and behavior of base class in addition to the properties and behavior of derived class.



Dynamic Inheritance

Dynamic inheritance allows objects to change and evolve over time. Since base classes provide properties and attributes for objects, hanging base classes changes the properties and attributes of a class.

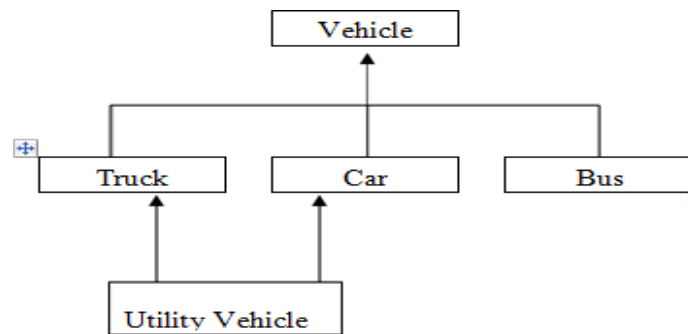
Example:

A window objects change to icon and back again. When we double click the folder the contents will be displayed in a window and when close it, changes back to icon. It involves changing a base class between a windows class and icon class.

Multiple Inheritances

Some object-oriented systems permit a class to inherit its state (attributes) and behavior from more than one super class. This kind of inheritance is referred to as multiple inheritances.

For example: Utility vehicle inherits the attributes from the Car and Truck classes.



Encapsulation and Information Hiding

Information hiding is the principle of concealing the internal data and procedures of an object. In C++ , encapsulation protection mechanism with private, public and protected members.

A car engine is an example of encapsulation. Although engines may differ in implementation, the interface between the driver and car is through a common protocol.

Polymorphism

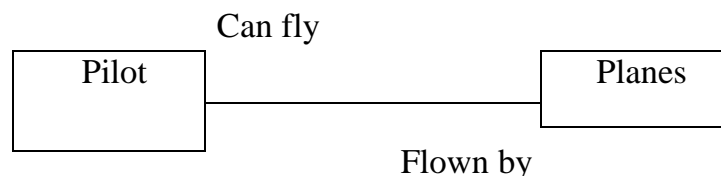
Poly → "many" Morph → "form"

It means objects that can take on or assume many different forms. Polymorphism means that the same operations may behave differently on different classes. Booch defines polymorphism as the relationship of objects many different classes by some common super class. Polymorphism allows us to write generic, reusable code more easily, because we can specify general instructions and delegate the implementation detail to the objects involved.

Example: In a pay roll system, manager, office worker and production worker objects all will respond to the compute payroll message, but the actual operations performed are object specific.

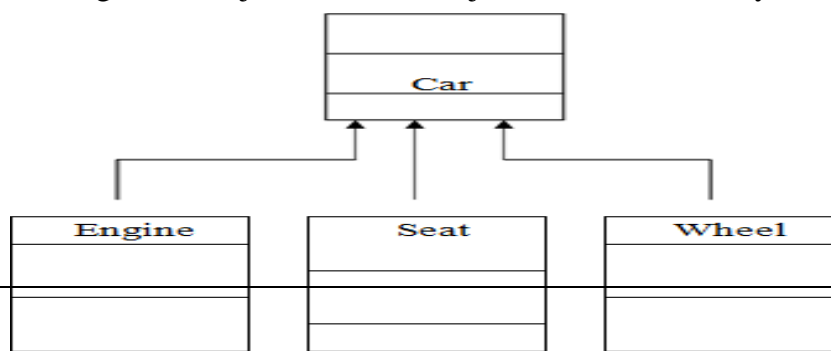
Object Relationship and Associations

Association represents the relationships between objects and classes. Associations are bi-directional. The directions implied by the name are the forward direction and the opposite is the inverse direction.



A pilot "can fly" planes. The inverse of can fly is "is flown by ". Plane "is flown by" pilot

Aggregations: All objects, except the most basic ones, are composed of and may contain other objects. Breaking down objects in to the objects from which they are composed is de



composition. This is possible because an object attributes need not be simple data fields, attributes can reference other objects. Since each object has an identity, one object can refer to other objects. This is known as aggregation. The car object is an aggregation of other objects such as engine, seat and wheel objects.

Static and Dynamic Binding:

Determining which function has to be involved at compile time is called static binding. Static binding optimized the calls. (Ex) function call.

The process of determining at run time which functions to involve is termed dynamic binding. Dynamic binding occurs when polymorphic call is issued. It allows some method invocation decision to be deferred until the information is known.

Object Persistence:

Objects have a lifetime. They are explicitly created and can exist for a period of time that has been the duration of the process in which they were created. A file or database can provide support for objects having a longer lifeline, longer than the duration of the process for which they are created. This characteristic is called object persistence.

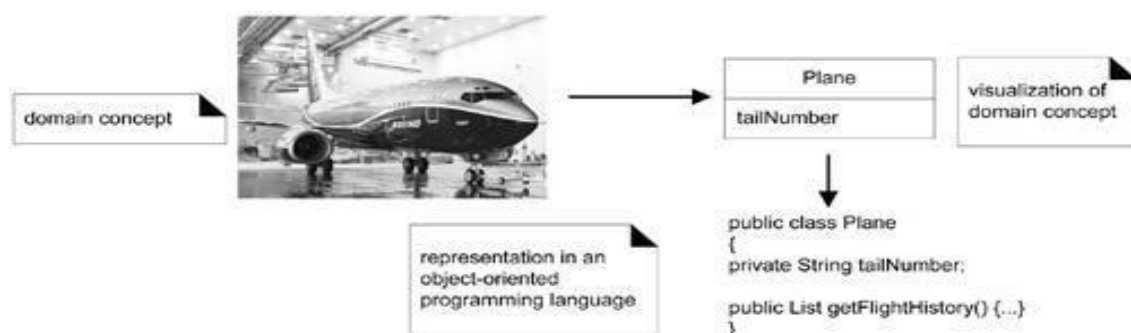
What is Object-Oriented Analysis and Design?

Object-Oriented Analysis : It emphasis on finding and describing the objects or concepts in the problem domain. For example, in the case of the flight information system, some of the concepts include Plane, Flight, and Pilot.

Object-Oriented Design : It emphasis on defining software objects and how they collaborate to fulfill the requirements. For example, a Plane software object may have a tail Number attribute and a get FlightHistory method.

During implementation or object-oriented programming, design objects are implemented, such as a Plane class in Java.

Object-orientation emphasizes representation of objects.



Example: a simple example a "dice game" in which software simulates a player rolling two dice. If the total is seven, they win; otherwise, they lose.



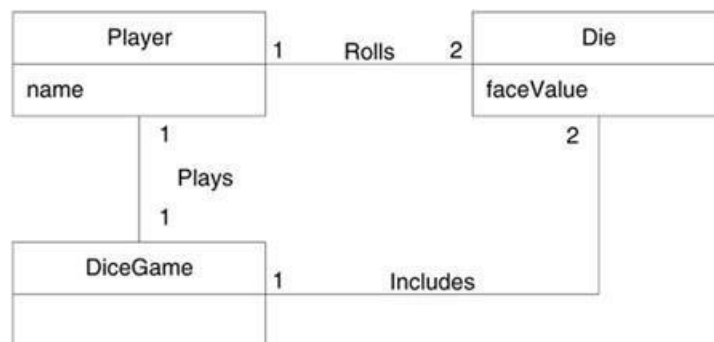
i) **Define Use Cases :** Requirements analysis may include stories or scenarios of how people use the application; these can be written as use cases. They are a popular tool in requirements analysis. For example, the Play a Dice Game use case:

Play a Dice Game: Player requests to roll the dice. System presents results: If the dice face value totals seven, player wins; otherwise, player loses.

ii) Define a Domain Model

There is an identification of the concepts, attributes, and associations that are considered important. The result can be expressed in a domain model that shows the important domain concepts or objects.

Partial domain model of the dice game.



This model illustrates the important concepts Player, Die, and Dice Game, with their associations and attributes. It is a visualization of the concepts or mental models of a real-world domain. Thus, it has also been called a **conceptual object model**.

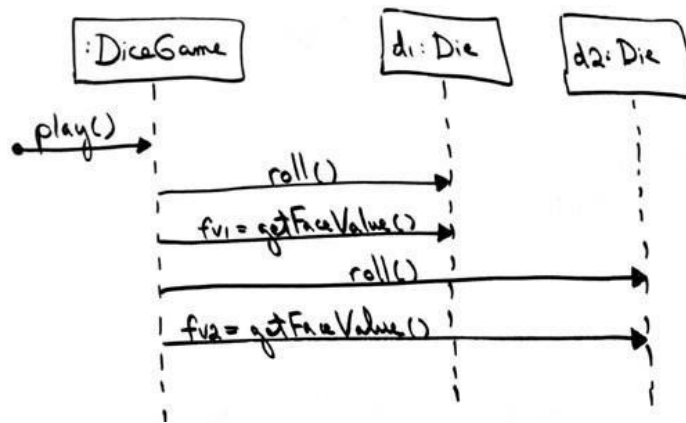
iii) Assign Object Responsibilities and Draw Interaction Diagrams

Object-oriented design is concerned with defining software objects their responsibilities and collaborations. It shows the flow of messages between software objects, and thus the invocation of methods.

For example, the sequence diagram in Figure 1.4 illustrates an OO software design, by sending messages to instances of the DiceGame and Die classes.

Notice that although in the real world a player rolls the dice, in the software design the DiceGame object "rolls" the dice (that is, sends messages to Die objects). Software object designs and programs do take some inspiration from real-world domains, but they are not direct models or simulations of the real world.

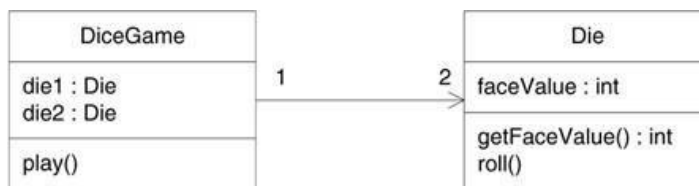
Sequence diagram illustrating messages between software objects.



iv) **Define Design Class Diagrams:** a static view of the class definitions is usefully shown with a design class diagram. This illustrates the attributes and methods of the classes.

For example, in the dice game, an inspection of the sequence diagram leads to the partial design class diagram shown in Figure 1.5. Since a play message is sent to a DiceGame object, the DiceGame class requires a play method, while class Die requires a roll and get FaceValue method.

Partial design class diagram.



UML DIAGRAMS

What is the UML?

The Unified Modeling Language is a visual language for specifying, constructing and documenting the artifacts of systems. The word visual in the definition is a key point -the UML is the de facto standard diagramming notation for drawing or presenting pictures. The standard is managed, and was created, by the Object Management Group. It was first added to the list of OMG adopted technologies in 1997.

UML is composed of 9 graphical diagrams:

- 1) **Class Diagram** - describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.
- 2) **Use – Case Diagram** - describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases.
- 3) **Behavior Diagram**

a. Interaction Diagram

- i. **Sequence Diagram** - shows how objects communicate with each other in terms of a sequence of messages. Also indicates the lifespan of objects relative to those messages.
- ii. **Communication diagram:-** how the interactions between objects or parts in terms of sequenced messages.

b. State Chart Diagram - describes the states and state transitions of the system.

c. Activity Diagram - describes the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.

4) Implementation Diagram

a. Component Diagram - describes how a software system is split up into components and shows the dependencies among these components.

b. Deployment Diagram - describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.

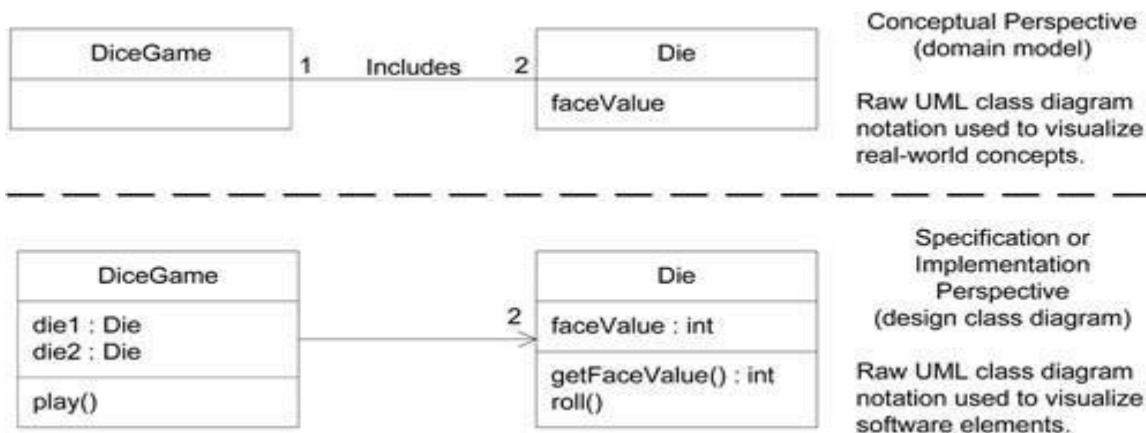
Three Ways to Apply UML

- **UML as sketch :** Informal and incomplete diagrams created to explore difficult parts of the problem or solution space, exploiting the power of visual languages.
- **UML as blueprint :** Relatively detailed design diagrams used either for
 - 1) **Reverse Engineering :** UML tool reads the source or binaries and generates UML package, class, and sequence diagrams to visualize and better understanding of existing code in UML diagrams .
 - 2) **Forward Engineering :** code generation . Before programming, some detailed diagrams can provide guidance for code generation (e.g., in Java), either manually or automatically with a tool. It's common that the diagrams are used for some code, and other code is filled in by a developer while coding
- **UML as programming language :** Complete executable specification of a software system in UML. Executable code will be automatically generated

Three Perspectives to Apply UML

The same UML class diagram notation can be used to draw pictures of concepts in the real world or software classes in Java.

1. **Conceptual perspective** - the diagrams are interpreted as describing things in a situation of the real world or domain of interest.
2. **Specification (software) perspective** - the diagrams (using the same notation as in the conceptual perspective) describe software abstractions or components with specifications and interfaces, but no commitment to a particular implementation (for example, not specifically a class in C# or Java).
3. **Implementation (software) perspective** - the diagrams describe software implementations in a particular technology (such as Java).



The Meaning of "Class" in Different Perspectives

- **Conceptual class** real-world concept or thing. A conceptual or essential perspective. The UP Domain Model contains conceptual classes.
- **Software class** a class representing a specification or implementation perspective of a software component, regardless of the process or method.
- **Implementation class** a class implemented in a specific OO language such as Java.

UNIFIED PROCESS (UP)

What is the UP?

A software development process describes an approach to building, deploying, and possibly maintaining software. The Unified Process has emerged as a popular iterative software development process for building object-oriented systems. In particular, the Rational Unified Process or RUP a detailed refinement of the Unified Process, has been widely adopted.

The UP combines commonly accepted best practices, such as an **iterative lifecycle and risk-driven development**, into a **cohesive and well-documented process description**.

UP for three reasons

1. The UP is an iterative process.
2. UP practices provide an example structure for how to do and thus how to explain OOA/D.
3. The UP is flexible, and can be applied in a lightweight and agile approach that includes practices from other agile methods

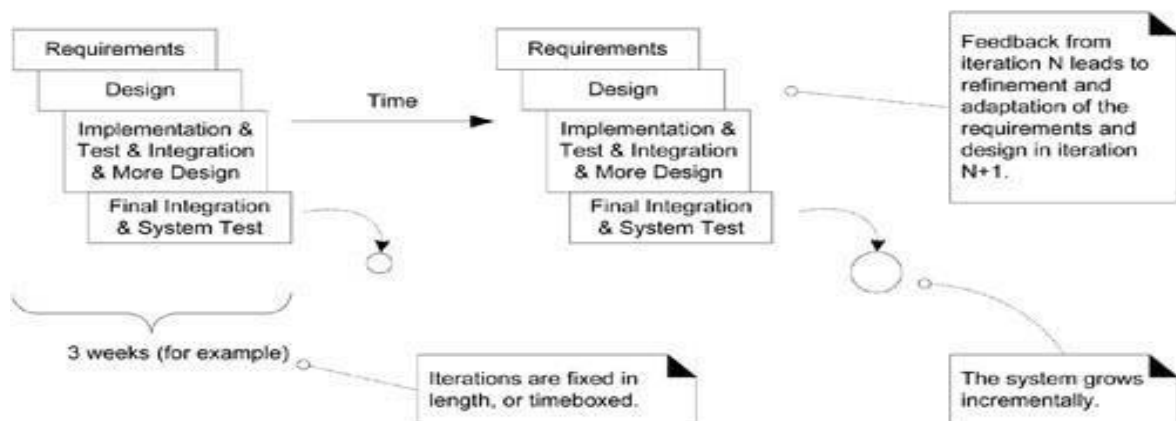
Iterative and Evolutionary Development

A key practice in both the UP and most other modern methods is iterative development.

- In this lifecycle approach, development is organized into a series of short, fixed-length (for example, three-week) mini-projects called iterations; the outcome of each is a tested, integrated, and executable partial system. Each iteration includes its own requirements analysis, design, implementation, and testing activities.
- The system grows incrementally over time, iteration by iteration, and thus this approach is also known as iterative and incremental development. Because feedback and adaptation evolve the specifications and design, it is also known as iterative and evolutionary development.

Iterative and evolutionary development.

2 – ITERATIVE, EVOLUTIONARY, AND AGILE

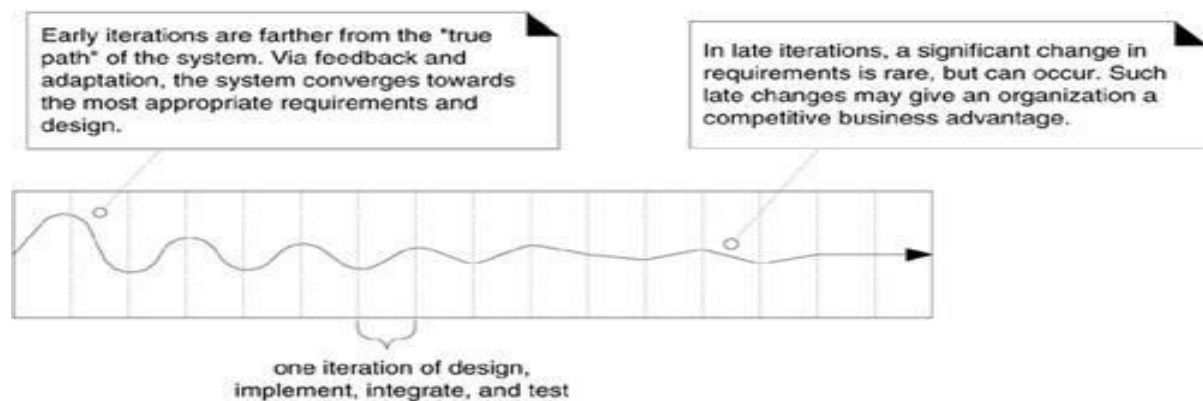


Notice in this example that there is neither a rush to code, nor a long drawn-out design step that attempts to perfect all details of the design before programming. The system may not be eligible for production deployment until after many iterations; for example, 10 or 15 iterations.

To Handle Change on an Iterative Project

- Each iteration involves choosing a small subset of the requirements, and quickly designing, implementing, and testing
- In addition to requirements clarification, activities such as load testing will prove if the partial design and implementation are on the right path, or if in the next iteration, a change in the core architecture is required.

Iterative feedback and evolution leads towards the desired system. The requirements and design instability lowers over time.



- Work proceeds through a series of structured build-feedback-adapt cycles. In early iterations the deviation from the "true path" of the system (in terms of its final requirements and design) will be larger than in later iterations. Over time, the system converges towards this path, as illustrated in Figure

Benefits of Iterative Development

- less project failure, better productivity, and lower defect rates; shown by research into iterative and evolutionary methods
- early rather than late mitigation of high risks (technical, requirements, objectives, usability, and so forth)
- early visible progress
- early feedback, user engagement, and adaptation, leading to a refined system that more closely meets the real needs of the stakeholders
- managed complexity; the team is not overwhelmed by "analysis paralysis" or very long and complex steps
- the learning within an iteration can be methodically used to improve the development process itself, iteration by iteration

What is Iteration Time boxing?

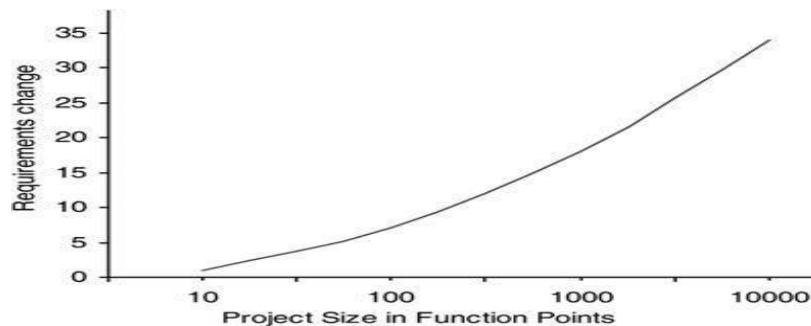
- Most iterative methods recommend an iteration length between two and six weeks.
- Small steps, rapid feedback, and adaptation are central ideas in iterative development; long iterations subvert the core motivation for iterative development and increase project risk.

- A very long time-boxed iteration misses the point of iterative development. Short is good.

Iterations are time-boxed, or fixed in length. For example, if the next iteration is chosen to be three weeks long, then the partial system must be integrated, tested, and stabilized by the scheduled date-date slippage is illegal. If it seems that it will be difficult to meet the deadline, the recommended response is to de-scope-remove tasks or requirements from the iteration, and include them in a future iteration, rather than slip the completion date.

In a waterfall lifecycle process there is an attempt to define all or most of the requirements before programming. It is strongly associated with

- high rates of failure
- lower productivity
- higher defect rates



Percentage of change on software projects of varying sizes.

The Need for Feedback and Adaptation

In complex, changing systems feedback and adaptation are key ingredients for success.

- Feedback from early development, programmers trying to read specifications, and client demos to refine the requirements.
- Feedback from tests and developers to refine the design or models.
- Feedback from the progress of the team tackling early features to refine the schedule and estimates.
- Feedback from the client and marketplace to re-prioritize the features to tackle in the next iteration.

What is Risk-Driven and Client-Driven Iterative Planning?

The UP encourages a combination of risk-driven and client-driven iterative planning. This means that the goals of the early iterations are chosen to 1) identify and drive down the highest risks, and 2) build visible features that the client cares most about.

Risk-driven iterative development includes more specifically the practice of architecture-centric iterative development, i.e early iterations focus on building, testing, and stabilizing the core architecture

What are Agile Methods and Attitudes?

Agile development methods usually apply time boxed iterative and evolutionary development, employ adaptive planning, promote incremental delivery, and include other values and practices that encourage agility rapid and flexible response to change.

The Agile Manifesto and Principles

The Agile Manifesto

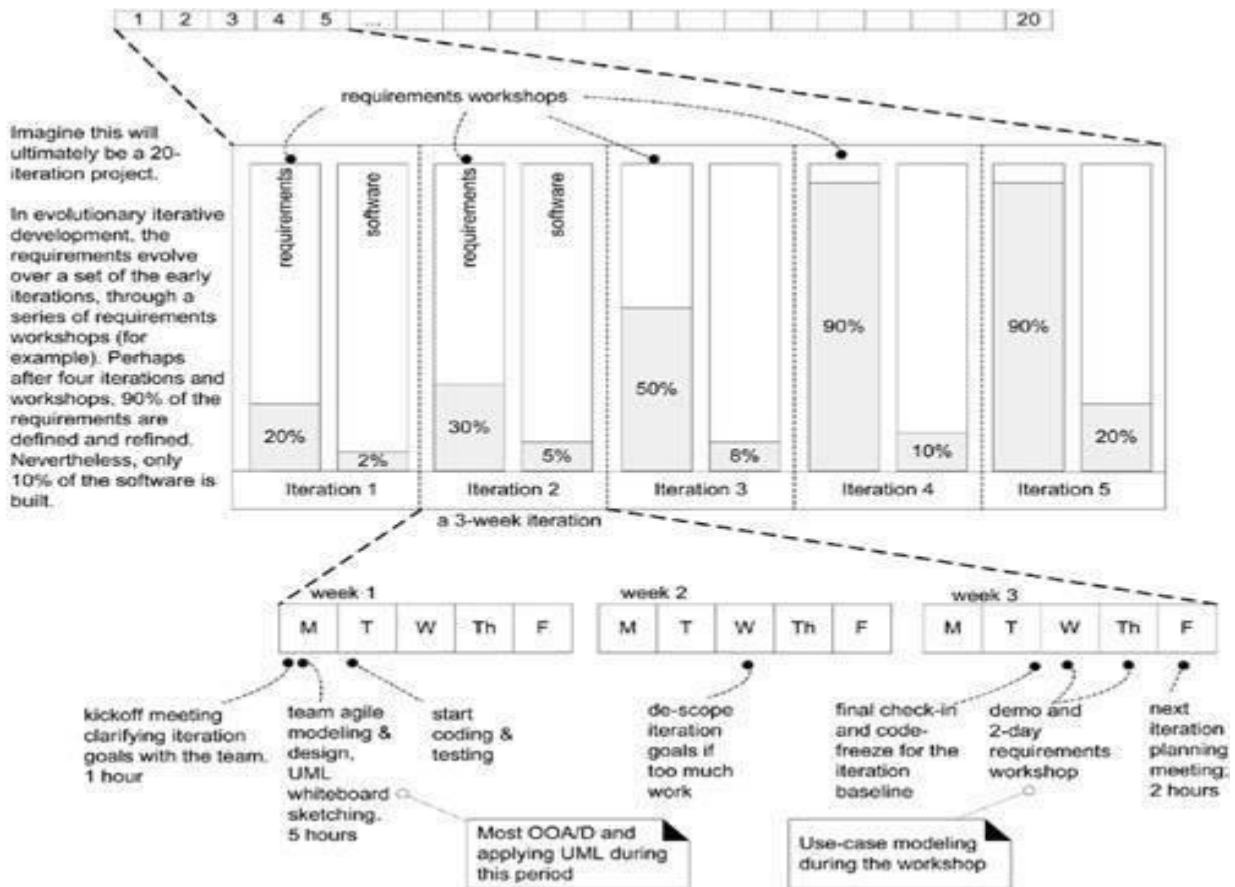
Individuals and interactions	over processes and tools
Working software	over comprehensive documentation
Customer collaboration	over contract negotiation
Responding to change	over following a plan

The Agile Principles

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.
4. Business people and developers must work together daily throughout the project
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development.
9. The sponsors, developers, and users should be able to maintain a constant pace indefinitely
10. Continuous attention to technical excellence and good design enhances agility
11. Simplicity the art of maximizing the amount of work not done is essential
12. The best architectures, requirements, and designs emerge from self-organizing teams.

13. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

This example assumes there will ultimately be 20 iterations on the project before delivery:



Evolutionary analysis and design the majority in early iterations.

UP Phases

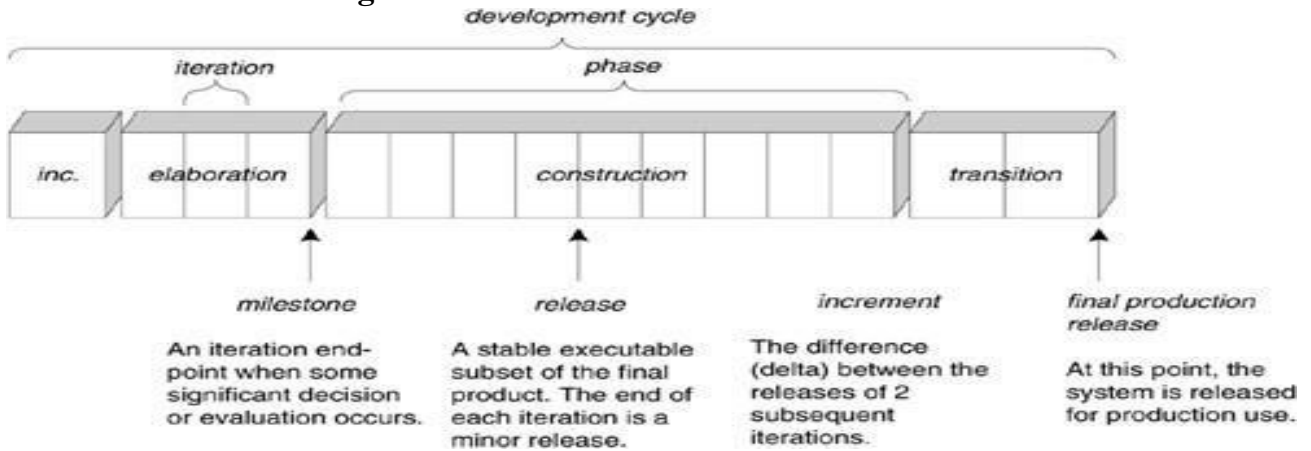
A UP project organizes the work and iterations across four major phases:

1. **Inception** - approximate vision, business case, scope, vague estimates.
2. **Elaboration** - refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates.
3. **Construction** - iterative implementation of the remaining lower risk and easier elements, and preparation for deployment.
4. **Transition** - beta tests, deployment.

This is not the old "waterfall" or sequential lifecycle of first defining all the requirements, and then doing all or most of the design. Inception is not a requirements phase; rather, it is a feasibility phase, where investigation is done to support a decision to continue or stop.

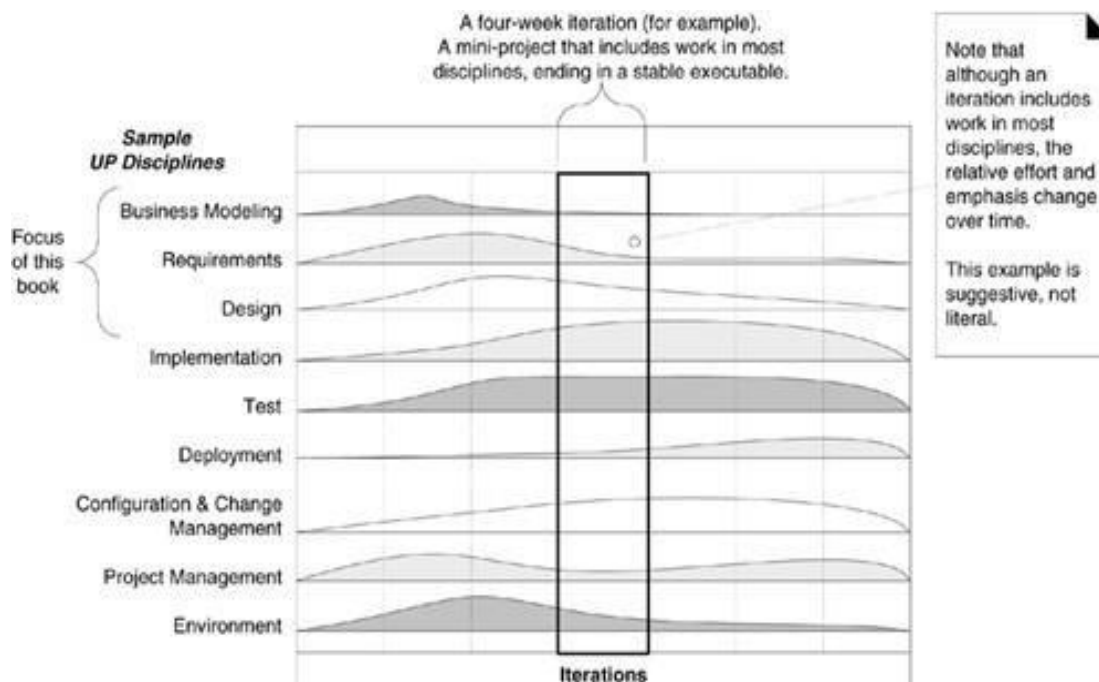
Similarly, elaboration is a phase where the core architecture is iteratively implemented, and high-risk issues are mitigated.

Figure - Schedule-oriented terms in the UP.



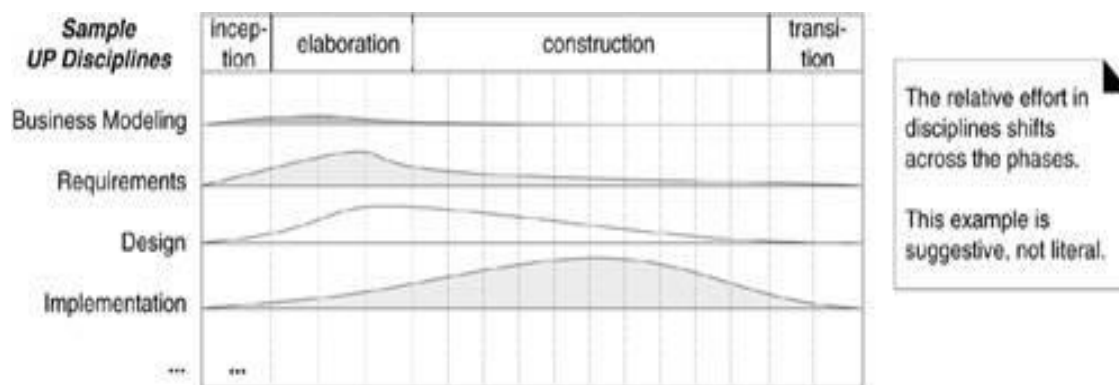
The UP Disciplines

Disciplines a set of activities (and related artifacts) in one subject area, such as the activities within requirements analysis. In the UP, an artifact is the general term for any work product: code, Web graphics, database schema, text documents, diagrams, models, and so on. There are several disciplines in the UP:



- **Business Modeling** - The Domain Model artifact, to visualize noteworthy concepts in the application domain.
- **Requirements** - The Use-Case Model and Supplementary Specification artifacts to capture functional and non-functional requirements.
- **Design** - The Design Model artifact, to design the software objects

What is the Relationship Between the Disciplines and Phases?



Definition: the Development Case: The choice of practices and UP artifacts for a project may be written up in a short document called the Development Case (an artifact in the Environment discipline).

Discipline	Practice	Artifact	Incep.	Elab.	Const.	Trans.
		Iteration	I1	E1..En	C1..Cn	T1..T2
Business Modeling	agile modeling req. workshop	Domain Model		s		
Requirements	req. workshop vision box exercise dot voting	Use-Case Model	s	r		
		Vision	s	r		
		Supplementary Specification	s	r		
		Glossary	s	r		
Design	agile modeling test-driven dev.	Design Model		s	r	
		SW Architecture Document		s		
		Data Model		s	r	
Implementation	test-driven dev. pair programming continuous integration coding standards	...				
Project Management	agile PM daily Scrum meeting	...				
...						

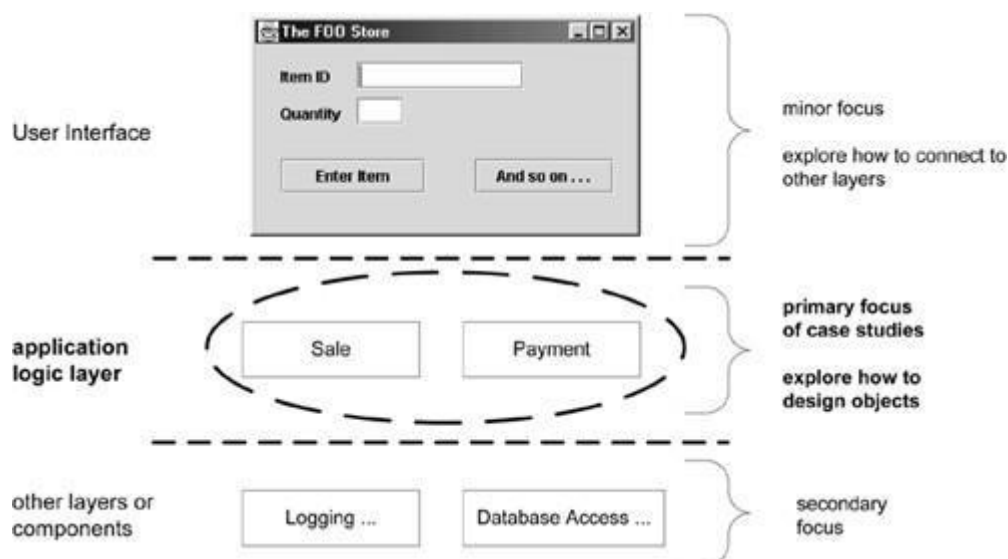
Table- Sample Development Case. s - start; r - refine

CASE STUDY

Applications include UI elements, core application logic, database access, and collaboration with external software or hardware components. This introduction to OOA/D focuses on the core application logic layer, with some secondary discussion of the other layers.

Why focus on OOA/D in the core application logic layer?

- Other layers are usually technology/platform dependent. For example, to explore the OO design of a Web UI or rich client UI layer in Java, we would need to learn in detail about a framework such as Struts or Swing.
- In contrast, the OO design of the core logic layer is similar across technologies.
- The essential OO design skills learned in the context of the application logic layer are applicable to all other layers or components.
- The design approach/patterns for the other layers tends to change quickly as new frameworks or technologies emerge.



Sample layers and objects in an object-oriented system, and the case study focus.

NEXTGEN POS SYSTEM

The NextGen POS(Point of Sale) System

- A POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store.
- It includes hardware components such as a computer and bar code scanner, and software to run the system.
- It interfaces to various service applications, such as a third-party tax calculator and inventory control. These systems must be relatively fault-tolerant; that is, even if

remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).



- A POS system increasingly must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input, wireless PDAs, and so forth.
- we are creating a commercial POS system that we will sell to different clients with disparate needs in terms of business rule processing. Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added. Therefore, we will need a mechanism to provide this flexibility and customization.

Using an iterative development strategy, we are going to proceed through requirements, object-oriented analysis, design, and implementation.

INCEPTION

Inception: The purpose of the inception phase is not to define all the requirements, or generate a believable estimate or project plan. Most requirements analysis occurs during the elaboration phase, in parallel with early production-quality programming and testing.

- Inception (in one sentence) - Envision the product scope, vision, and business case.
- It may include the first requirements workshop, planning for the first iteration, and then quickly moving forward to elaboration. Common inception artifacts and indicates the issues they address.
- For example, the Use-Case Model may list the names of most of the expected use cases and actors, but perhaps only describe 10% of the use cases in detail done in the service of developing a rough high-level vision of the system scope, purpose, and risks.
- Note that some programming work may occur in inception in order to create "proof of concept" prototypes, to clarify a few requirements via UI-oriented prototypes, and to do programming experiments for key "show stopper" technical questions.

Table -Sample inception artifacts.

Artifact	Comment
Vision and Business Case	Describes the high-level goals and constraints, the business case, and provides an executive summary.
Use-Case Model	Describes the functional requirements. During inception, the names of most use cases will be identified, and perhaps 10% of the use cases will be analyzed in detail.
Supplementary Specification	Describes other requirements, mostly non-functional. During inception, it is useful to have some idea of the key non-functional requirements that have will have a major impact on the architecture.
Glossary	Key domain terminology, and data dictionary.
Risk List & Risk Management Plan	Describes the risks (business, technical, resource, schedule) and ideas for their mitigation or response.
Prototypes and proof-of-concepts	To clarify the vision, and validate technical ideas.
Iteration Plan	Describes what to do in the first elaboration iteration.
Phase Plan & Software Development Plan	Low-precision guess for elaboration phase duration and effort. Tools, people, education, and other resources.
Development Case	A description of the customized UP steps and artifacts for this project. In the UP, one always customizes it for the project.

Activities in Inception:

Inception is a short step to elaboration. It determines basic feasibility, risk, and scope, to decide if the project is worth more serious investigation.

Activities and artifacts in inception include:

- a short requirements workshop
- most actors, goals, and use cases named
- most use cases written in brief format; 10-20% of the use cases are written in fully dressed detail to improve understanding of the scope and complexity
- most influential and risky quality requirements identified
- version one of the Vision and Supplementary Specification written
- risk list
- Technical proof-of-concept prototypes and other investigations to explore the technical feasibility of special requirements
- user interface-oriented prototypes to clarify the vision of functional requirements
- recommendations on what components to buy/build/reuse, to be refined in elaboration
 - For example, a recommendation to buy a tax calculation package.

- high-level candidate architecture and components proposed
 - This is not a detailed architectural description, and it is not meant to be final or correct. Rather, it is brief speculation to use as a starting point of investigation in elaboration. For example, "A Java client-side application, no application server, Oracle for the database," In elaboration, it may be proven worthy, or discovered to be a poor idea and rejected.
- plan for the first iteration

USE CASES & USECASE MODELLING

Use cases are text stories, widely used to discover and record requirements. use cases are text stories of some actor using a system to meet goals. There are 3 formats to represent the use case

- I) Brief Format
- II) Casual Format
- III) Fully Dressed Format

Definition: What are Actors, Scenarios, and Use Cases

An **actor** is something with behavior, such as a person (identified by role), computer system, or organization; for example, a cashier.

A **scenario** is a specific sequence of actions and interactions between actors and the system; it is also called a use case instance. It is one particular story of using a system, or one path through the use case; for example, the scenario of successfully purchasing items with cash, or the scenario of failing to purchase items because of a credit payment denial.

A **use case** is a collection of related success and failure scenarios that describe an actor using a system to support a goal.

Use Cases and the Use-Case Model

- Use cases are text documents, not diagrams, and use-case modeling is primarily an act of writing text, not drawing diagrams.
- There are also the Supplementary Specification, Glossary, Vision, and Business Rules. These are all useful for requirements analysis.
- The Use-Case Model may optionally include a UML use case diagram to show the names of use cases and actors, and their relationships. This gives a nice context diagram of a system and its environment. It also provides a quick way to list the use cases by name.

Motivation: Why Use Cases?

- Lack of user involvement in software projects is near the top of the list of reasons for project failure. Use cases are a good way to help keep it simple, and make it possible for domain experts or requirement donors to themselves write use cases.
- Another value of use cases is that they emphasize the user goals and perspective; we ask the question "Who is using the system, what are their typical scenarios of use, and what are their goals?" This is a more user-centric emphasis compared to simply asking for a list of system features.

Definition: Are Use Cases Functional Requirements

Use cases are requirements, primarily functional or behavioral requirements that indicate what the system will do. A related viewpoint is that a use case defines a contract of how a system will behave

What are Three Kinds of Actors?

An actor is anything with behavior, including the system under discussion (SuD) itself when it calls upon the services of other systems. **Primary and supporting** actors will appear in the action steps of the use case text. **Actors are roles played not only by people, but by organizations, software, and machines.** There are three kinds of external actors in relation to the SuD:

1. **Primary actor** has user goals fulfilled through using services of the SuD. For example, the cashier.
 - Why identify? To find user goals, which drive the use cases.
2. **Supporting actor** provides a service (for example, information) to the SuD. The automated payment authorization service is an example. Often a computer system, but could be an organization or person.
 - Why identify? To clarify external interfaces and protocols.
3. **Offstage actor** has an interest in the behavior of the use case, but is not primary or supporting; for example, a government tax agency.
 - Why identify? To ensure that all necessary interests are identified and satisfied. Offstage actor interests are sometimes subtle or easy to miss unless these actors are explicitly named.

Three Common Use Case Formats

- **Brief** - Terse one-paragraph summary, usually of the main success scenario. The prior Process Sale example was brief. It was created during early requirements analysis, to get a quick sense of subject and scope. May take only a few minutes to create.

Example : Process Sale: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items. Notice that use cases are not diagrams, they are text.

- **Casual** - Informal paragraph format. Multiple paragraphs that cover various scenarios. It was created during early requirements analysis, to get a quick sense of subject and scope. May take only a few minutes to create.

Example : Handle Returns Usecase

Main Success Scenario: A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item ...

Alternate Scenarios: If the customer paid by credit, and the reimbursement transaction to their credit account is rejected, inform the customer and pay them with cash.

If the item identifier is not found in the system, notify the Cashier and suggest manual entry of the identifier code . If the system detects failure to communicate with the external accounting system,

- **Fully Dressed** - All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees. It was created , after many use cases have been identified and written in a brief format, then during the first requirements workshop a few (such as 10%) of the architecturally significant and high-value use cases are written in detail.

Use Case Section	Comment
Use Case Name	Start with a verb.
Scope	The system under design.
Level	"user-goal" or "subfunction"
Primary Actor	Calls on the system to deliver its services.
Stakeholders and Interests	Who cares about this use case, and what do they want?
Preconditions	What must be true on start, and worth telling the reader?
Success Guarantee	What must be true on successful completion, and worth telling the reader.
Main Success Scenario	A typical, unconditional happy path scenario of success.
Extensions	Alternate scenarios of success or failure.
Special Requirements	Related non-functional requirements.

Use Case Section	Comment
Technology and Data Variations List	Varying I/O methods and data formats.
Frequency of Occurrence	Influences investigation, testing, and timing of implementation.
Miscellaneous	Such as open issues.

Use Case UC1: Process Sale : Fully Dressed Format Example

Scope: NextGen POS application

Level: user goal

Primary Actor: Cashier

Stakeholders and Interests:- Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer shortages are deducted from his/her salary.

- **Salesperson:** Wants sales commissions updated.

- **Customer:** Wants purchase and fast service with minimal effort. Wants easily visible display of entered items and prices. Wants proof of purchase to support returns.

- **Company:** Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory.

- **Manager:** Wants to be able to quickly perform override operations, and easily debug Cashier problems.

- **Government Tax Agencies:** Want to collect tax from every sale. May be multiple agencies, such as national, state, and county.

- **Payment Authorization Service:** Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.

Preconditions: Cashier is identified and authenticated.

Success Guarantee (or Postconditions): Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.

Special Requirements:

- Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
- Credit authorization response within 30 seconds 90% of the time.
- Somehow, we want robust recovery when access to remote services such the inventory system is failing.
- Language internationalization on the text displayed.
- Pluggable business rules to be insertable at steps 3 and 7.
- ...

Technology and Data Variations List:

*a. Manager override entered by swiping an override card through a card reader, or entering an authorization code via the keyboard.

3a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard.

3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.

...

Frequency of Occurrence: Could be nearly continuous.

Open Issues:

- What are the tax law variations?
- Explore the remote service recovery issue.
- What customization is needed for different businesses?
- Must a cashier take their cash drawer when they log out?
- Can the customer directly use the card reader, or does the cashier have to do it?

Format Description:

Scope : It can be either system use case or Business Usecase .

- system use case : A use case describes use of one software system
- business use case: At a broader scope, use cases can also describe how a business is used by its customers and partners. Such an enterprise-level process description is called a business use case.

Level : Use cases are classified as at the user-goal level or the sub function level, among others.

A user-goal level use case is the common kind that describe the scenarios to fulfill the goals of a primary actor to get work done.

A subfunction-level use case describes substeps required to support a user goal, and is usually created to factor out duplicate substeps shared by several regular use cases an example is the subfunction use case Pay by Credit, which could be shared by many regular use cases.

Primary Actor : The principal actor that calls upon system services to fulfill a goal.

Stakeholders and Interests List It satisfies all the stakeholders' interests. by starting with the stakeholders and their interests before writing the remainder of the use case, we have a method to remind us what the more detailed responsibilities of the system should be.

Stakeholders and Interests:

Stakeholders and Interests:

- Cashier: Wants accurate, fast entry and no payment errors, as cash drawer shortages are deducted from his/her salary.
 - Salesperson: Wants sales commissions updated.
 - ...
-

Preconditions and Success Guarantees (Postconditions)

Preconditions state what must always be true before a scenario is begun in the use case. Preconditions are not tested within the use case; rather, they are conditions that are assumed to be true. Typically, a precondition implies a scenario of another use case, such as logging in, that has successfully completed.

Main Success Scenario and Steps (or Basic Flow) This has also been called the "happy path" scenario, or the more prosaic "Basic Flow" or "Typical Flow." It describes a typical success path that satisfies the interests of the stakeholders.

The scenario records the steps, of which there are three kinds:

1. An interaction between actors.
2. A validation (usually by the system).
3. A state change by the system (for example, recording or modifying something).

Main Success Scenario:

1. Customer arrives at a POS checkout with items to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. ...

Cashier repeats steps 3-4 until indicates done.

5. ...

Extensions (or Alternate Flows)

Extension scenarios are branches (both success and failure) from the main success scenario, and so can be notated with respect to its steps 1...N. For example, at Step 3 of the main success scenario there may be an invalid item identifier, either because it was incorrectly entered or unknown to the system. An extension is labeled "3a"; it first identifies the condition and then the response. Alternate extensions at Step 3 are labeled "3b" and so forth.

Extensions:

3a. Invalid identifier:

1. System signals error and rejects entry.

3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):

1. Cashier can enter item category identifier and the quantity.

An extension has two parts: the condition and the handling.

Guideline: When possible, write the condition as something that can be detected by the system or an actor.

This extension example also demonstrates the notation to express failures within extensions.

7b. Paying by credit:

1. Customer enters their credit account information.
2. System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.

2a. System detects failure to collaborate with external system:

1. System signals error to Cashier.
2. Cashier asks Customer for alternate payment.

Special Requirements

If a non-functional requirement, quality attribute, or constraint relates specifically to a use case, record it with the use case. These include qualities such as performance, reliability, and usability, and design constraints (often in I/O devices) that have been mandated or considered likely.

Special Requirements:

- Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
- Credit authorization response within 30 seconds 90% of the time.
- Language internationalization on the text displayed.
- Pluggable business rules to be insertable at steps 2 and 6.

Technology and Data Variations List

A common example is a technical constraint imposed by a stakeholder regarding input or output technologies. For example, a stakeholder might say, "The POS system must

support credit account input using a card reader and the keyboard." It is also necessary to understand variations in data schemes, such as using UPCs or EANs for item identifiers, encoded in bar code symbology.

Technology and Data Variations List:

- 3a. Item identifier entered by laser scanner or keyboard.
- 3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.
- 7a. Credit account information entered by card reader or keyboard.
- 7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.

Guidelines For Use Case Modeling:

Guideline 1. Write in an Essential UI-Free Style

Guideline : Write use cases in an essential style; keep the user interface out and focus on actor intent.

Essential Style

Assume that the Manage Users use case requires identification and authentication:

- 1. Administrator identifies self.
- 2. System authenticates identity.
- 3. ...

The design solution to these intentions and responsibilities is wide open: biometric readers, graphical user interfaces (GUIs), and so forth.

Concrete Style Avoid During Early Requirements Work

In contrast, there is a concrete use case style. In this style, user interface decisions are embedded in the use case text. The text may even show window screen shots, discuss window navigation, GUI widget manipulation and so forth. For example:

- 1. Administrator enters ID and password in dialog box
- 2. System authenticates Administrator.
- 3. System displays the "edit users" window
- 4. ...

Guideline 2. Write Terse Use Cases : Delete "noise" words. Even small changes add up, such as "System authenticates..." rather than "The System authenticates..."

Guideline 3 : Write Black-Box Use Cases : Black-box use cases are the most common and recommended kind; they do not describe the internal workings of the system, its components, or design. Rather, the system is described as having responsibilities, which is a common unifying metaphorical theme in object-oriented thinking software elements have responsibilities and collaborate with other elements that have responsibilities.

Black-box style	Not Recommended
The system records the sale.	The system writes the sale to a database. ...or (even worse): The system generates a SQL INSERT statement for the sale...

Guideline 4 : Take an Actor and Actor-Goal Perspective : Here's the RUP use case definition, from the use case founder Ivar Jacobson:

A set of use-case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor. It stresses two attitudes during requirements analysis:

- Write requirements focusing on the users or actors of a system, asking about their goals and typical situations.
- Focus on understanding what the actor considers a valuable result.

Guideline 5: To Find Use Cases

Use cases are defined to satisfy the goals of the primary actors. Hence, the basic procedure is:

1.	Choose the system boundary. Is it just a software application, the hardware and application as a unit, that plus a person using it, or an entire organization?
2.	Identify the primary actors those that have goals fulfilled through using services of the system.
3.	Identify the goals for each primary actor.
4.	Define use cases that satisfy user goals; name them according to their goal. Usually, user-goal level use cases will be one-to-one with user goals, but there is at least one exception, as will be examined.

Step 1: Choose the System Boundary

The POS system itself is the system under design; everything outside of it is outside the system boundary, including the cashier, payment authorization service, and so on. For example, is the complete responsibility for payment authorization within the system boundary? No, there is an external payment authorization service actor.

Steps 2 and 3: Find Primary Actors and Goals

Guideline: Identify the primary actors first, as this sets up the framework for further investigation. The following questions help to identify other actors.

Who starts and stops the system?	Who does system administration?
Who does user and security management?	Is "time" an actor because the system does something in response to a time event?
Is there a monitoring process that restarts the system if it fails?	Who evaluates system activity or performance?
How are software updates handled? Push or pull update?	Who evaluates logs? Are they remotely retrieved?
In addition to human primary actors, are there any external software or robotic systems that call upon services of the system?	Who gets notified when there are errors or failures?

Representing Goals of an Actor :

There are at least two approaches:

1. Draw them in a use case diagram, naming the goals as use cases.
2. Write an actor-goal list first, review and refine it, and then draw the use case diagram.

For example:

Actor	Goal	Actor	Goal
Cashier	process sales process rentals handle returns cash in cash out...	System Administrator	add users modify users delete users manage security manage system tables...
Manager	start up shut down...	Sales Activity System	analyze sales and performance data
...

Is the Cashier or Customer the Primary Actor?

The answer depends on the system boundary of the system under design, and who we are primarily designing the system for the viewpoint of the POS system the system services the goal of a trained cashier (and the store) to process the customer's sale.

Primary actors and goals at different system boundaries.



The customer is an actor, but in the context of the NextGen POS, not a primary actor; rather, the cashier is the primary actor because the system is being designed to primarily serve the trained cashier's "power user" goals. The system does not have a UI and functionality that could equally be used by the customer or cashier. Rather, it is optimized to meet the needs and training of a cashier.

Second Method to Find Actors and Goals - Event Analysis

Another approach to aid in finding actors, goals, and use cases is to identify external events. What are they, where from, and why? Often, a group of events belong to the same use case. For example:

External Event	From Actor	Goal/Use Case
enter sale line item	Cashier	process a sale
enter payment	Cashier or Customer	process a sale
...		

Step 4: Define Use Cases

In general, define one use case for each user goal. Name the use case similar to the user goal for example,

Goal: process a sale; Use Case: Process Sale.

Start the name of use cases with a verb. A common exception to one use case per goal is to collapse CRUD (create, retrieve, update, delete) separate goals into one CRUD use case, idiomatically called Manage <X>. For example, the goals "edit user," "delete user," and so forth are all satisfied by the Manage Users use case.

Guideline 6: Tests To Find Useful Use Cases

There are several rules of thumb, including:

- The Boss Test

- The EBP Test
- The Size Test

The Boss Test : To check for achieving results of measurable value

Your boss asks, "What have you been doing all day?" You reply: "Logging in!" Is your boss happy?. If not, the use case fails the Boss Test, which implies it is not strongly related to achieving results of measurable value. It may be a use case at some low goal level, but not the desirable level of focus for requirements analysis.

The EBP Test

- An Elementary Business Process (EBP) is a term from the business process engineering field, defined as:

EBP is similar to the term user task in usability engineering, although the meaning is less strict in that domain. Focus on use cases that reflect EBPs.

- A task performed by one person in one place at one time, in response to a business event, which adds measurable business value and leaves the data in a consistent state, e.g., Approve Credit or Price Order
- The EBP Test is similar to the Boss Test, especially in terms of the measurable business value qualification.

The Size Test

A use case typically contains many steps, and in the fully dressed format will often require 3- 10 pages of text. A common mistake in use case modeling is to define just a single step within a series of related steps as a use case by itself, such as defining a use case called Enter an Item ID. You can see a hint of the error by its small size the use case name will wrongly suggest just one step within a larger series of steps, and if you imagine the length of its fully dressed text, it would be extremely large.

Example: Applying the Tests

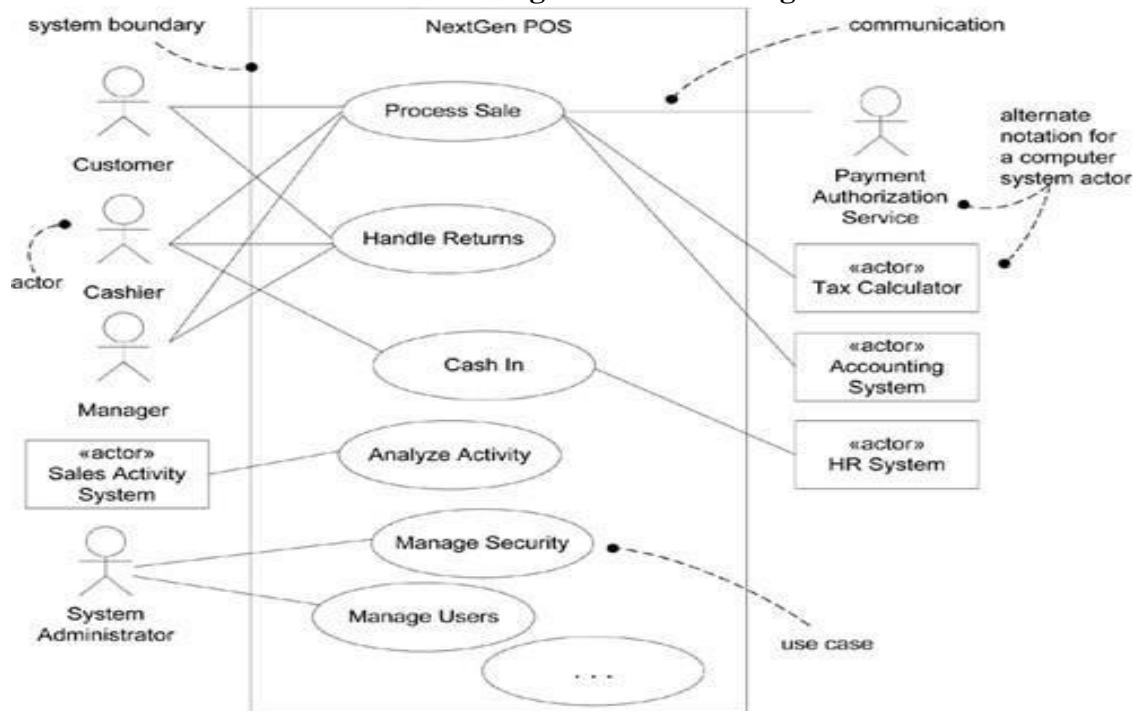
- Negotiate a Supplier Contract
 - Much broader and longer than an EBP. Could be modeled as a business use case, rather than a system use case.
- Handle Returns
 - OK with the boss. Seems like an EBP. Size is good.
- Log In
 - Boss not happy if this is all you do all day!
- Move Piece on Game Board
 - Single step fails the size test.

Applying UML: Use Case Diagrams

- The UML provides use case diagram notation to illustrate the names of use cases and actors, and the relationships between them

- Use case diagrams and use case relationships are secondary in use case work. Use cases are text documents. Doing use case work means to write text.

Partial use case Diagram - context diagram.



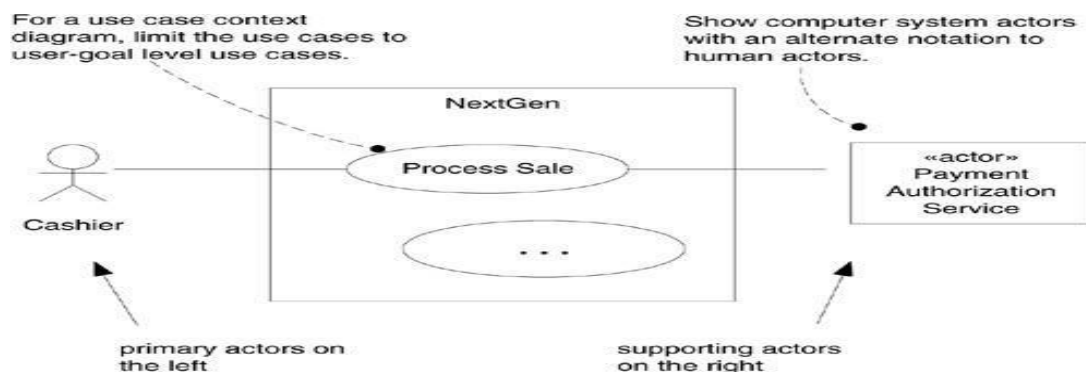
Guideline

- Use case diagram is an excellent picture of the system context
- It makes a good context diagram that is, showing the boundary of a system, what lies outside of it, and how it gets used.
- It serves as a communication tool that summarizes the behavior of a system and its actors.

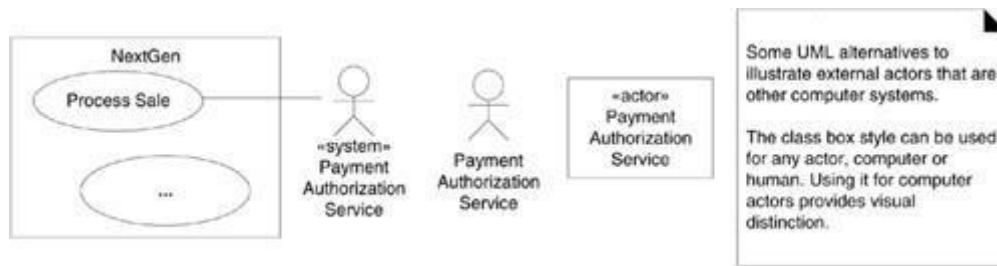
Guideline: Diagramming

Notice the actor box with the symbol «actor». This style is used for UML keywords and stereotypes, and includes guillemet symbols special single-character brackets («actor», not <<actor>>)

Notation suggestions.



Alternate actor notation.



RELATING USE CASES

Use cases can be related to each other. For example, a sub function use case such as Handle Credit Payment may be part of several regular use cases, such as Process Sale and Process Rental. It is simply an organization mechanism to (ideally) improve communication and comprehension of the use cases, reduce duplication of text, and improve management of the use case documents.

Kinds of relationships

1. The include Relationship
2. The extend Relationship
3. The generalize Relationship

1. The include Relationship

It is common to have some partial behavior that is common across several use cases. For example, the description of paying by credit occurs in several use cases, including Process Sale, Process Rental, Contribute to Lay-away Plan.. Rather than duplicate this text, it is desirable to separate it into its own subfunction use case, and indicate its inclusion.

UC1: Process Sale

...Main Success Scenario:

1.Customer arrives at a POS checkout with goods and/or services to purchase.

...

7.Customer pays and System handles payment.

...

Extensions:

7b. Paying by credit: Include Handle Credit Payment.

7c. Paying by check: Include Handle Check Payment.

...

UC7: Process Rental

...

Extensions:

6b. Paying by credit: Include Handle Credit Payment.

...

This is the include relationship. A slightly shorter (and thus perhaps preferred) notation to indicate an included use case is simply to underline it or highlight it in some fashion. For example:

UC1: Process Sale

...

Extensions:

7b. Paying by credit: Handle Credit Payment.

7c. Paying by check: Handle Check Payment.

...

Notice that the Handle Credit Payment subfunction use case was originally in the Extensions section of the Process Sale use case, but was factored out to avoid duplication.

Another use of the include relationship is to describe the handling of an asynchronous event, such as when a user is able to, at any time, select or branch to a particular window, function, or Web page, or within a range of steps.

The basic notation is to use the a*, b*, ... style labels in the Extensions section

UC1: Process FooBars

...

Main Success Scenario:

1. ...

Extensions:

a*. At any time, Customer selects to edit personal information: Edit Personal Information.

b*. At any time, Customer selects printing help: Present Printing Help.

2-11. Customer cancels: Cancel Transaction Confirmation.

Use cases and use the include relationship when:

- They are duplicated in other use cases.

- A use case is very complex and long, and separating it into subunits aids comprehension.

Concrete, Abstract, Base, and Addition Use Cases

A **concrete use case** is initiated by an actor and performs the entire behavior desired by the actor. These are the elementary business process use cases. For example, Process Sale is a concrete use case.

An **abstract use case** is never instantiated by itself; it is a subfunction use case that is part of another use case. Handle Credit Payment is abstract; it doesn't stand on its own, but is always part of another story, such as Process Sale.

A use case that includes another use case, or that is extended or specialized by another use case is called **a base use case**. Process Sale is a base use case with respect to the included Handle Credit Payment subfunction use case. On the other hand, the use case that is an inclusion, extension, or specialization is called an addition use case. Handle Credit Payment is the addition use case in the include relationship to Process Sale. Addition use cases are usually abstract. Base use cases are usually concrete.

2. The extend Relationship

The idea is to create an extending or addition use case, and within it, describe where and under what condition it extends the behavior of some base use case. For example:

UC1: Process Sale (the base use case)

...

Extension Points: VIP Customer, step 1. Payment, step 7.

Main Success Scenario:

1. Customer arrives at a POS checkout with goods and/or services to purchase.

...

7. Customer pays and System handles payment.

...

UC15: Handle Gift Certificate Payment (the extending use case)

...Trigger: Customer wants to pay with gift certificate.

Extension Points: Payment in Process Sale.

Level: Subfunction

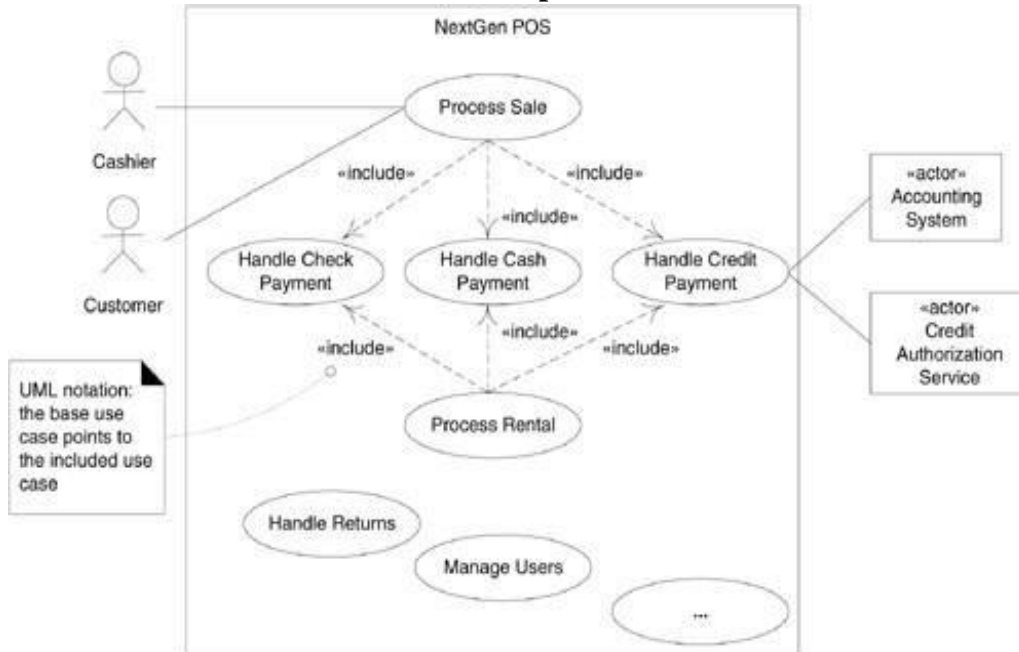
Main Success Scenario:

1. Customer gives gift certificate to Cashier.
2. Cashier enters gift certificate ID.

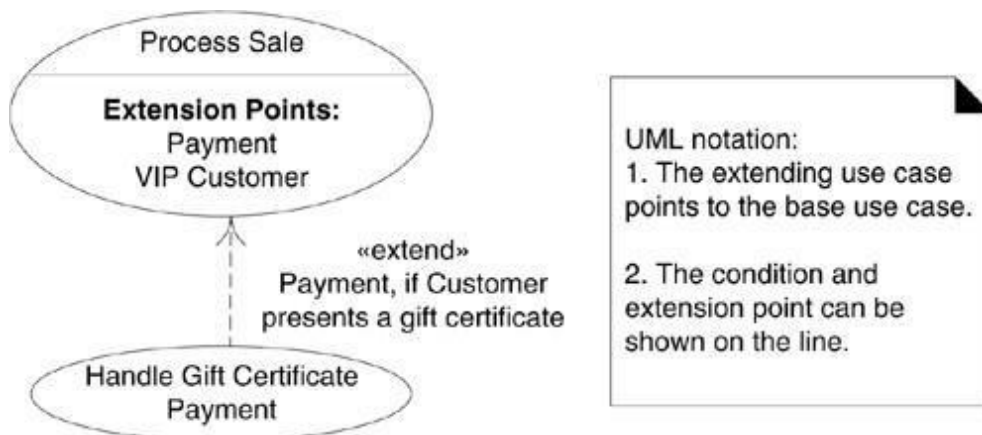
...

This is an example of an extend relationship. The use of an extension point, and that the extending use case is triggered by some condition. Extension points are labels in the base use case which the extending use case references as the point of extension, so that the step numbering of the base use case can change without affecting the extending use case.

Use case include relationship in the Use-Case Model.



The extend relationship.



3. The Generalization Relationship

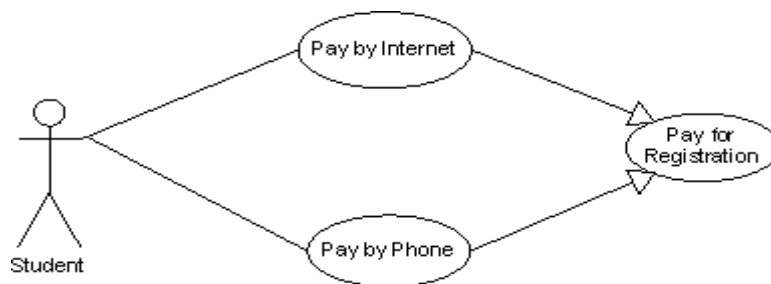
In the context of use case modeling the use case generalization refers to the relationship which can exist between two use cases and which shows that one use case (child) inherits the structure, behavior, and relationships of another actor (parent). The child use case is also referred to the more specialized use case while the parent is also referred to as the more abstract use case of the relationship.

For those of you familiar with object oriented concepts: use cases in UML are classes and the generalization is simply the inheritance relationship between two use cases by which one use case inherits all the properties and relationships of another use case.

You can use the generalization relationship when you find two or more use cases which have common behavior/logic. In this instance, you can describe the common parts in a separate use case (the parent) which then is specialized into two or more specialized child use cases.

Example:

If you are creating a payment system which allows students of a training provider to pay for courses both on-line and by phone, there will many things in common between the two scenarios: specifying personal info, specifying payment info, etc. However, there would also be differences between the two. So, the best way to accomplish this is to create one use case (the parent) which contains the common behavior and then create two specialized child use cases which inherit from the parent and which contain the differences specific to registering on-line vs. by phone.



WHEN TO USE USECASE DIAGRAM

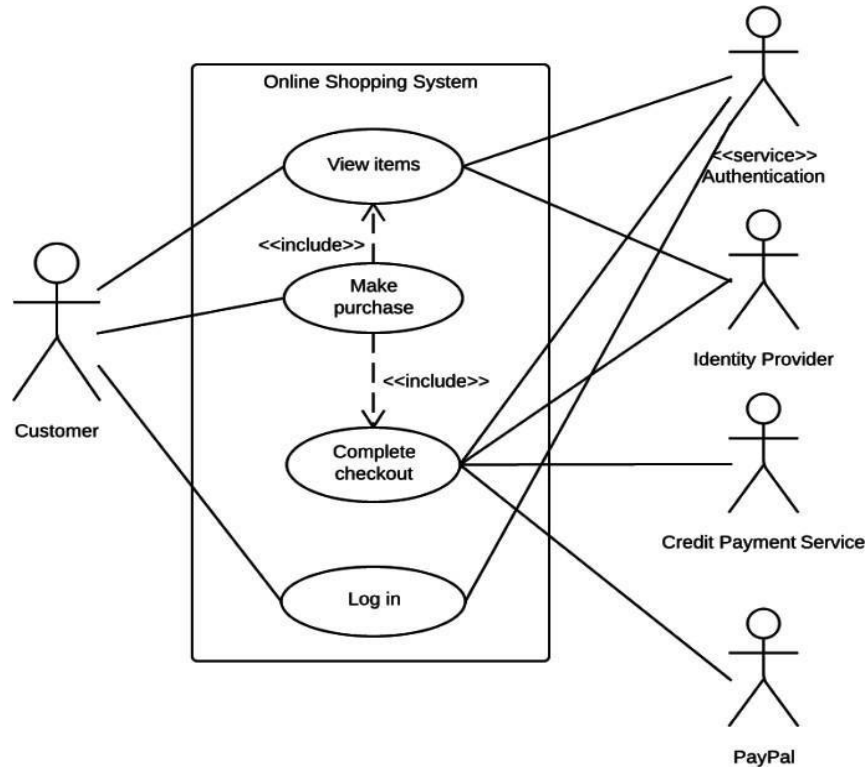
When to Use USECASE DIAGRAM

UML use case diagrams are ideal for:

- Representing the goals of system-user interactions
- Defining and organizing functional requirements in a system
- Specifying the context and requirements of a system
- Modeling the basic flow of events in a use case

- Reverse engineering.
- Forward engineering.

Ex: Online shopping System



Use case diagrams are valuable for visualizing the functional requirements of a system that will translate into design choices and development priorities. They also help identify any internal or external factors that may influence the system and should be taken into consideration.

Use case diagrams specify the events of a system and their flows. But use case diagram never describes how they are implemented. Use case diagram can be imagined as a black box where only the input, output, and the function of the black box is known. Although use case is not a good candidate for forward and reverse engineering, still they are used in a slightly different way to make forward and reverse engineering. The same is true for reverse engineering. Use case diagram is used differently to make it suitable for reverse engineering. In forward engineering, use case diagrams are used to make test cases and in reverse engineering use cases are used to prepare the requirement details from the existing application.

UNIT II - STATIC UML DIAGRAMS

Class Diagram— Elaboration – Domain Model – Finding conceptual classes and description classes – Associations – Attributes – Domain model refinement – Finding conceptual class Hierarchies – Aggregation and Composition – Relationship between sequence diagrams and use cases – When to use Class Diagrams

CLASS DIAGRAM

Introduction

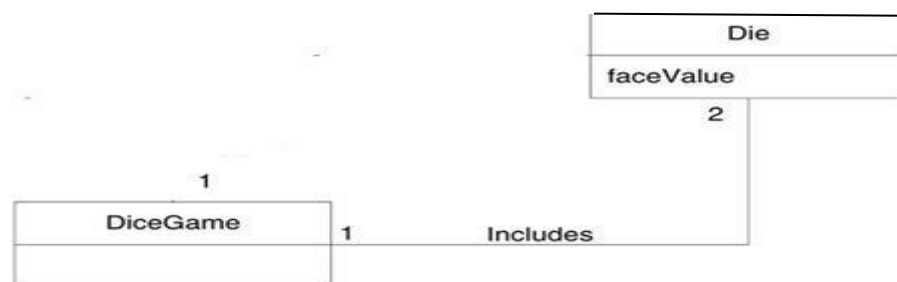
- The UML includes class diagrams to illustrate classes, interfaces, and their associations. They are used for static object modeling.
- Used for static object modeling . It is used to depict the classes within a model.
- It describes responsibilities of the system , it is used in forward and reverse engineering
- Keywords used along with class name are { abstract , interface, actor }

Definition: Design Class Diagram

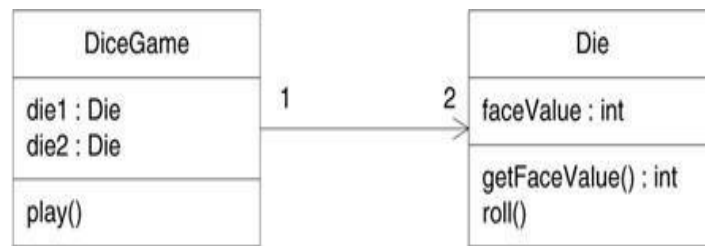
The class diagram can be used to visualize a domain model. we also need a unique term to clarify when the class diagram is used in a software or design perspective. A common modeling term for this purpose is design class diagram (DCD).

UML class diagrams in two perspectives

Domain Model

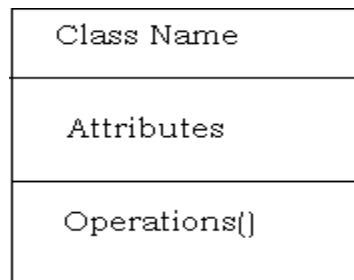


Design Model



Class Diagram Representation

Class is represented as rectangular box showing classname, attributes , operations.



The main elements of class are

- 1 *Attributes*
- 2 *Operations & Methods*
- 3 *Relationship between classes*

1.Attributes (refer pg no 26)

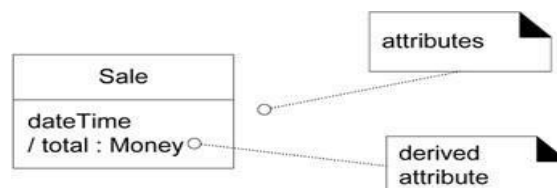
An **attribute** is a logical data value of an object. Attributes of a classifier also called structural properties in the UML. The full format of the attribute text notation is:

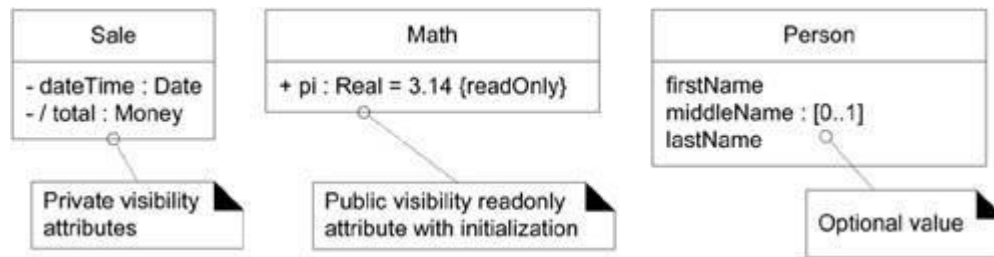
Syntax:

visibility name : type multiplicity = default {property-string}

visibility marks include + (public), - (private)

Examples:



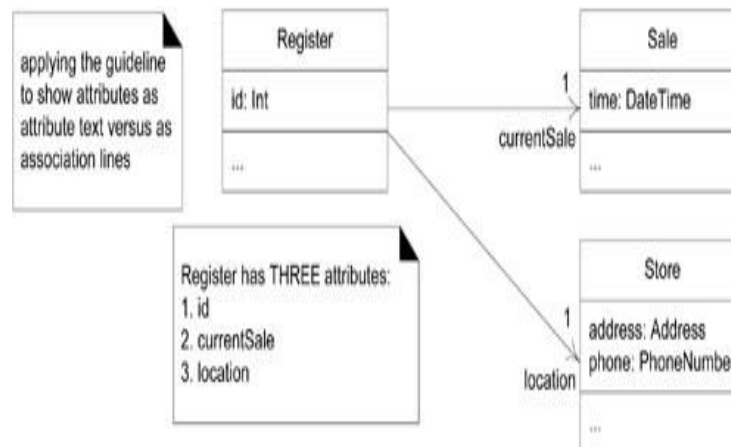


different formats

```

- classOrStaticAttribute : int
+ publicAttribute : string
- privateAttribute
  assumedPrivateAttribute
isInitializedAttribute : Bool = true
aCollection : VeggieBurger [*]
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : int = 5 { readOnly }
/derivedAttribute
  
```

Guideline: Use the attribute text notation for data type objects and the association line notation for others.



2. Operations and Methods

Operations : One of the compartments of the UML class box shows the signatures of operations. Assume the version that includes a return type. Operations are usually assumed public if no visibility is shown. both expressions are possible

An operation is not a method. A UML **operation** is a declaration, with a name, parameters, return type, exceptions list, and possibly a set of constraints of pre- and post-conditions. methods are implementations.

Syntax :

```
visibility name (parameter-list) : return-type {property-string}
```

Example:

UML REPRESENTATION	+ getPlayer(name : String) : Player {exception IOException}
JAVA CODING	public Player getPlayer(String name) throws IOException

```

+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
- privateMethod()
# protectedMethod()
~ packageVisibleMethod()
«constructor» SuperclassFoo( Long )
methodWithParms(parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() {exception IOException}
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }

```

To Show Methods in Class Diagrams:

A UML **method** is the implementation of an operation. A method may be illustrated several ways, including:

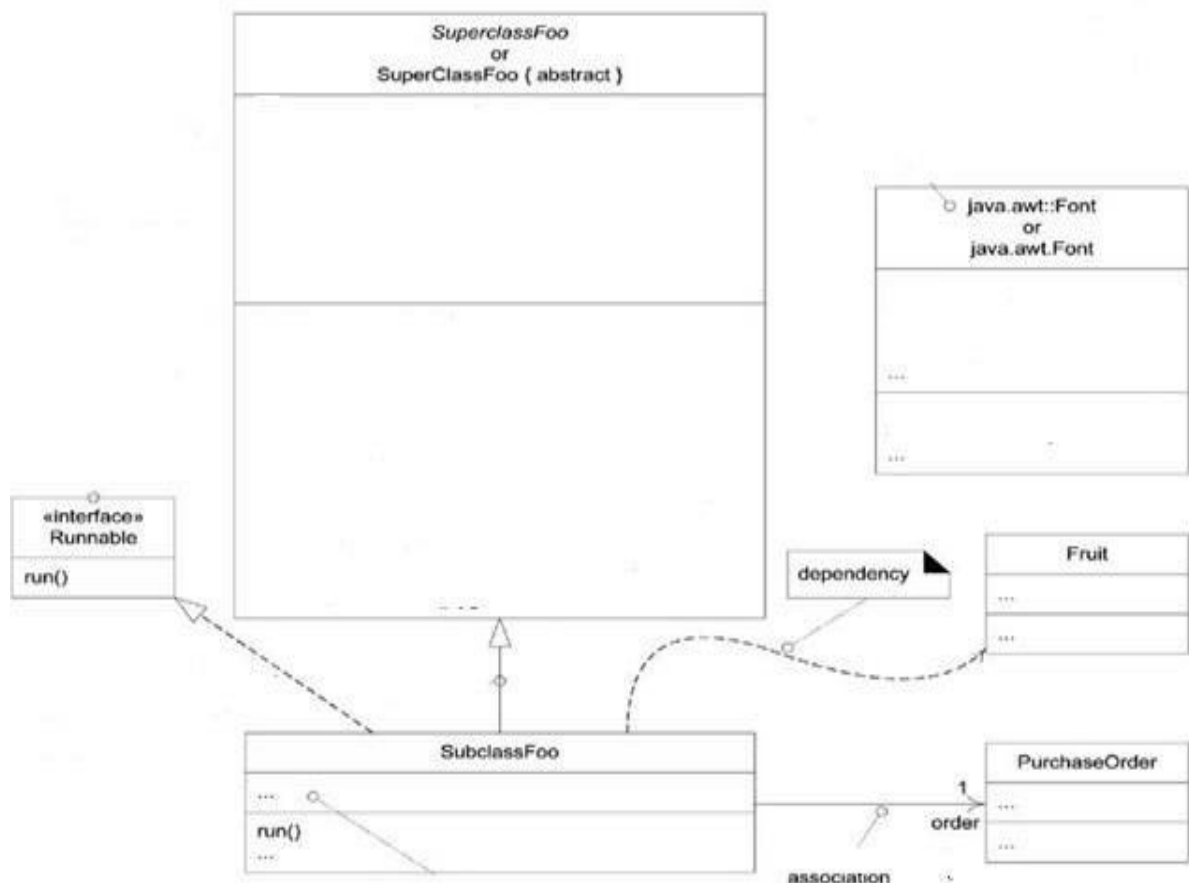
- in interaction diagrams, by the details and sequence of messages
- in class diagrams, with a UML note symbol stereotyped with «method»



3. Relationship between classes

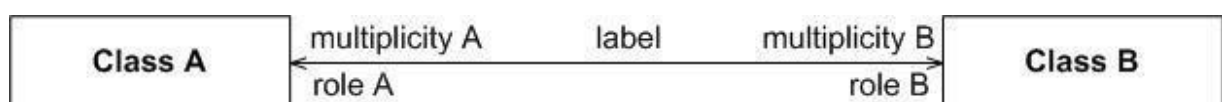
There are different relationship exists between classes. They are

- Association
- Generalization & specialization
- Composition and aggregation
- Dependency
- Interface realization

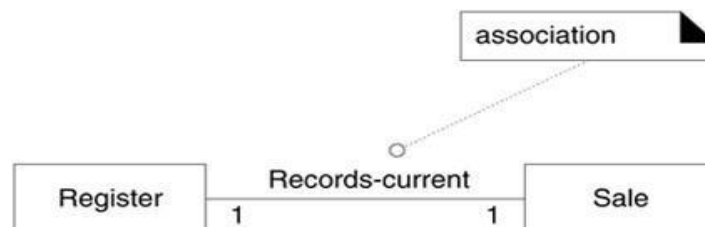


A) Association (refer page no 21)

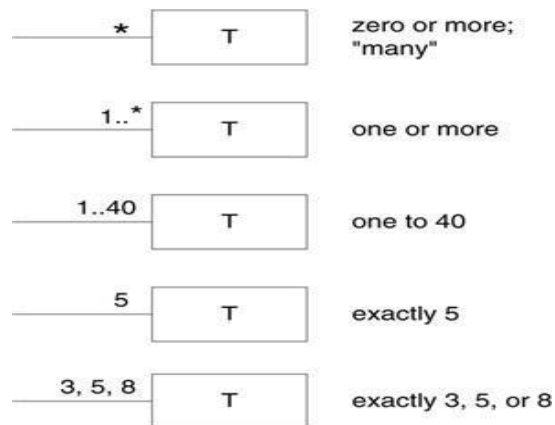
An **association** is a relationship between classes. The semantic relationship between two or more classifiers that involve connections among their instances.



Example



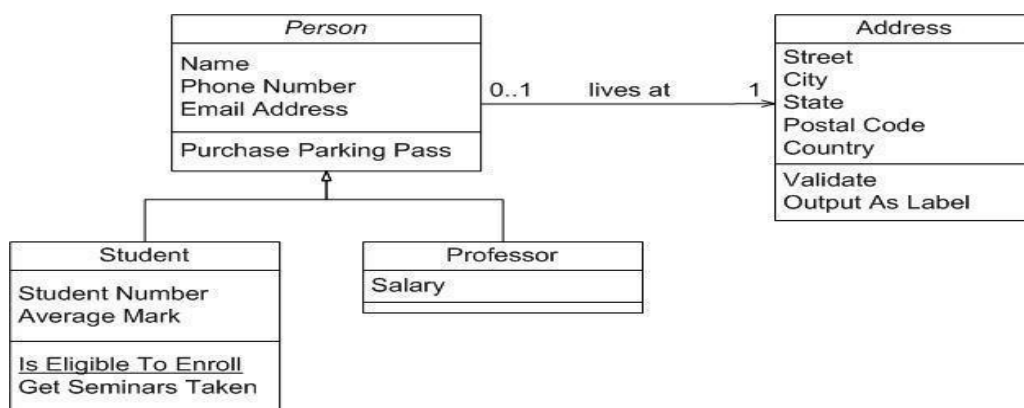
For example, a single instance of a class can be associated with "many" (zero or more, indicated by the *) Item instances.



B) Generalization & Specialization

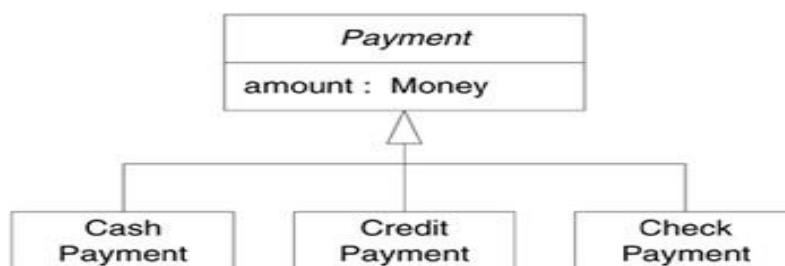
Generalization is the activity of identifying commonality among concepts and defining superclass (general concept) and subclass (specialized concept) relationships.

Ex1:



In the above example person is the generalized class and specialized classes are student and professor

Ex2:

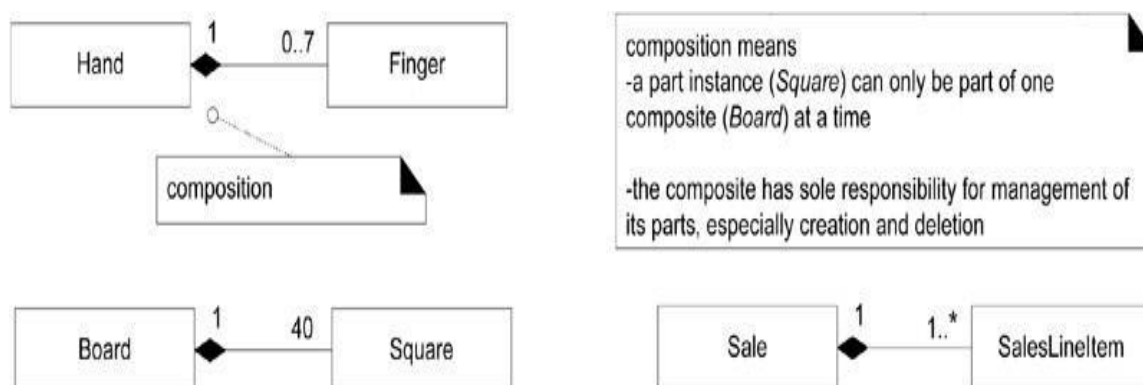


In the above example payment is the generalized class and specialized classes are cash payment , credit payment and check payment .

C) Composition and Aggregation

Composition, also known as composite aggregation, is a strong kind of whole-part aggregation and is useful to show in some models. A composition relationship implies that

- 1) an instance of the part (such as a Square) belongs to only one composite instance (such as one Board) at a time,
- 2) the part must always belong to a composite (no free-floating Fingers)
- 3) the composite is responsible for the creation and deletion of its parts either by itself creating/deleting the parts, or by collaborating with other objects.



Aggregation is a vague kind of association in the UML that loosely suggests whole-part relationships. Aggregation implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.



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.

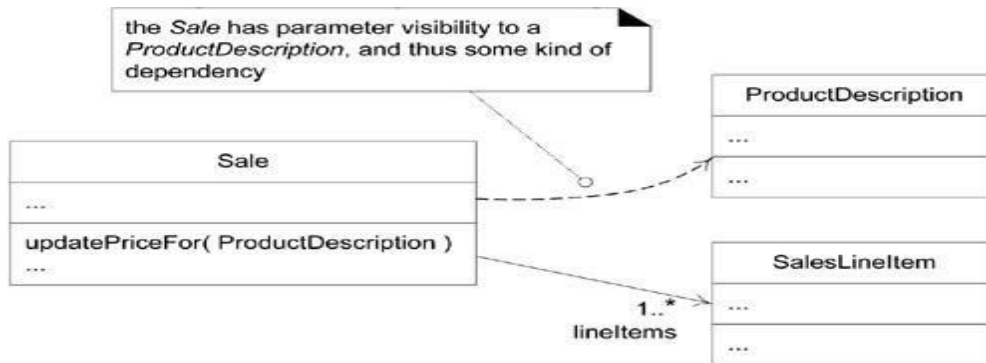
D) Dependency

A general dependency relationship indicates that a client element (of any kind, including classes, packages, use cases, and so on) has knowledge of another supplier element and that a change in the supplier could affect the client.

Dependency can be viewed as another version of **coupling**, a traditional term in software development when an element is coupled to or depends on another.

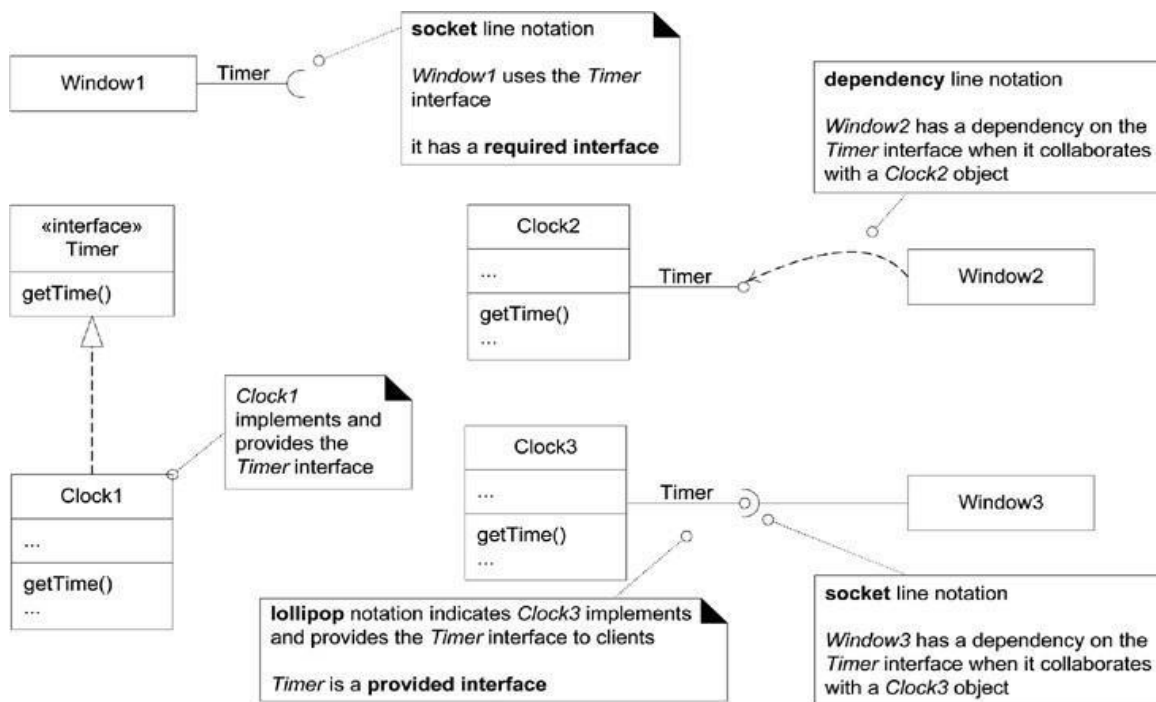
There are many kinds of dependency

- having an attribute of the supplier type
- sending a message to a supplier; the visibility to the supplier could be:
 - an attribute, a parameter variable, a local variable, a global variable, or class visibility (invoking static or class methods)
- receiving a parameter of the supplier type
- the supplier is a superclass or interface



E) Interface realization

The UML provides several ways to show **interface** implementation, providing an interface to clients, and interface dependency (a required interface). In the UML, interface implementation is formally called interface realization

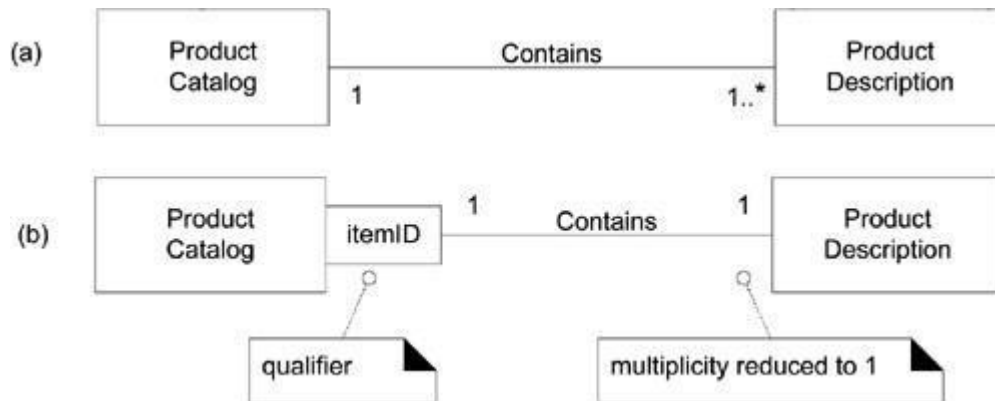


In the above example , Clock is the server program implementing Timer interface giving Timer as the provided interface, window is the client program with Timer

as required interface. The Timer interface contains the services provided by the server object.

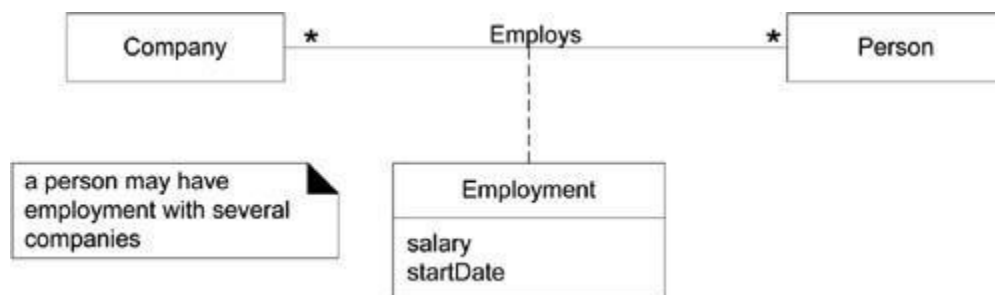
Qualified Association

A **qualified association** has a qualifier that is used to select an object (or objects) from a larger set of related objects, based upon the qualifier key



Association Class

An association class allows you treat an association itself as a class, and model it with attributes, operations, and other features. For example, if a Company employs many Persons, modeled with an Employs association, you can model the association itself as the Employment class, with attributes such as startDate.



ELABORATION

Elaboration is the initial series of iterations during which, on a normal project:

- the core, risky software architecture is programmed and tested
- the majority of requirements are discovered and stabilized
- the major risks are mitigated or retired
- Build the core architecture, resolve the high-risk elements, define most requirements, and estimate the overall schedule and resources.
- Elaboration is the initial series of iterations during which the team does serious investigation, implements (programs and tests) the core architecture, clarifies most requirements, and tackles the high-risk issues.

- Elaboration often consists of two or more iterations; Each iteration is recommended to be between two and six weeks; prefer the shorter versions unless the team size is massive. Each iteration is time boxed, i.e its end date is fixed.
- Elaboration is not a design phase or a phase when the models are fully developed in preparation for implementation in the construction step that would be an example of superimposing waterfall ideas on iterative development and the UP.
- During this phase, no prototypes are created ; rather, the code and design are production-quality portions of the final system.
- Architectural prototype means a production subset of the final system. More commonly it is called the executable architecture or architectural baseline.

Key Ideas and Best Practices will manifest in elaboration:

- do short time boxed risk-driven iterations
- start programming early
- adaptively design, implement, and test the core and risky parts of the architecture
- test early, often, realistically
- adapt based on feedback from tests, users, developers
- write most of the use cases and other requirements in detail, through a series of workshops, once per elaboration iteration

Table -Sample elaboration artifacts, excluding those started in inception.

Artifact	Comment
Domain Model	This is a visualization of the domain concepts; it is similar to a static information model of the domain entities.
Design Model	This is the set of diagrams that describes the logical design. This includes software class diagrams, object interaction diagrams, package diagrams, and so forth.
Software Architecture Document	A learning aid that summarizes the key architectural issues and their resolution in the design. It is a summary of the outstanding design ideas and their motivation in the system.
Data Model	This includes the database schemas, and the mapping strategies between object and non-object representations.
Use-Case Storyboards, UI Prototypes	A description of the user interface, paths of navigation, usability models, and so forth.

Process: Planning the Next Iteration

Organize requirements and iterations by risk, coverage, and criticality.

- Risk includes both technical complexity and other factors, such as uncertainty of effort or usability.
- Coverage implies that all major parts of the system are at least touched on in early iterations perhaps a "wide and shallow" implementation across many components.
- Criticality refers to functions the client considers of high business value.

These criteria are used to rank work across iterations. Use cases or use case scenarios are ranked for implementation early iterations implement high ranking scenarios. The ranking is done before iteration-1, but then again before iteration-2, and so forth, as new requirements and new insights influence the order.

For example:

Rank	Requirement (Use Case or Feature)	Comment
High	Process Sale Logging ...	Scores high on all rankings. Pervasive. Hard to add late. ...
Medium	Maintain Users ...	Affects security sub domain. ...
Low

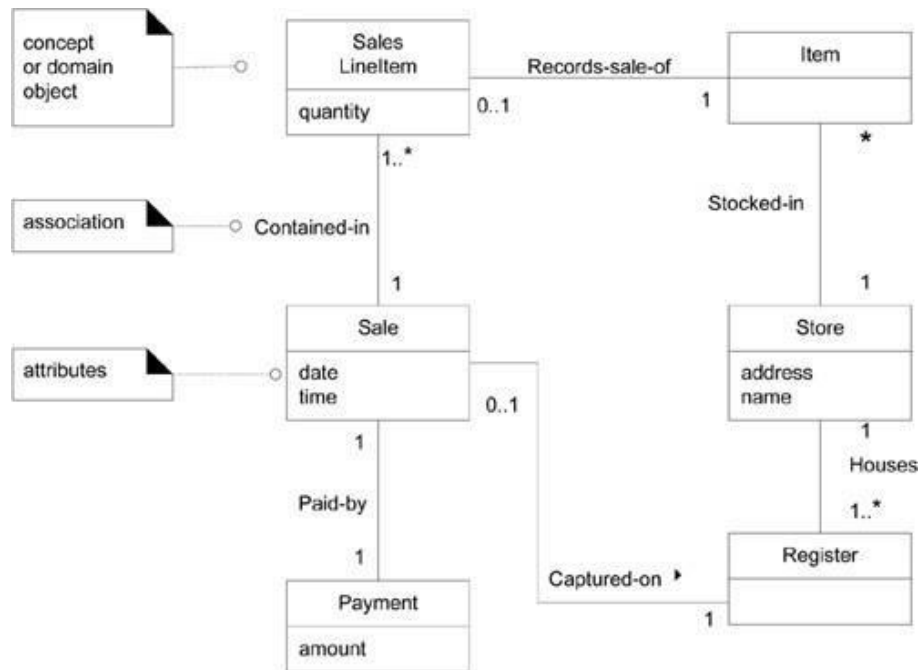
Based on this ranking, we see that some key architecturally significant scenarios of the Process Sale use case should be tackled in early iterations.

DOMAIN MODEL

Domain Models

The figure shows a partial domain model drawn with UML class diagram notation. It illustrates that the conceptual classes of Payment and Sale are significant in this domain, that a Payment is related to a Sale in a way that is meaningful to note, and that a Sale has a date and time, information attributes we care about.

Applying the UML class diagram notation for a domain model yields a conceptual perspective model. Identifying a rich set of conceptual classes is at the heart of OO analysis.



What is a Domain Model?

A domain model is a visual representation of conceptual classes or real-situation objects in a domain . Domain models have also been called conceptual models domain object models, and analysis object models.

Definition

In the UP, the term "Domain Model" means a representation of real-situation conceptual classes, not of software objects. The term does not mean a set of diagrams describing software classes, the domain layer of a software architecture, or software objects with responsibilities.

A domain model is illustrated with a set of class diagrams in which no operations (method signatures) are defined. It provides a conceptual perspective. It may show:

- domain objects or conceptual classes
- associations between conceptual classes
- attributes of conceptual classes

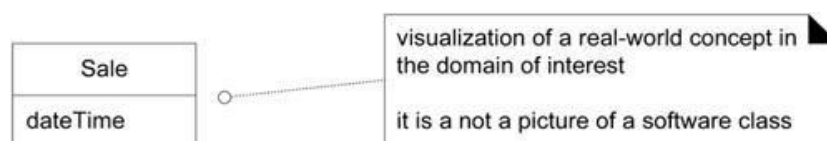
Why Call a Domain Model a "Visual Dictionary"?

Domain Model visualizes and relates words or concepts in the domain. It also shows an abstraction of the conceptual classes, because there are many other things one could communicate about registers, sales, and so forth.

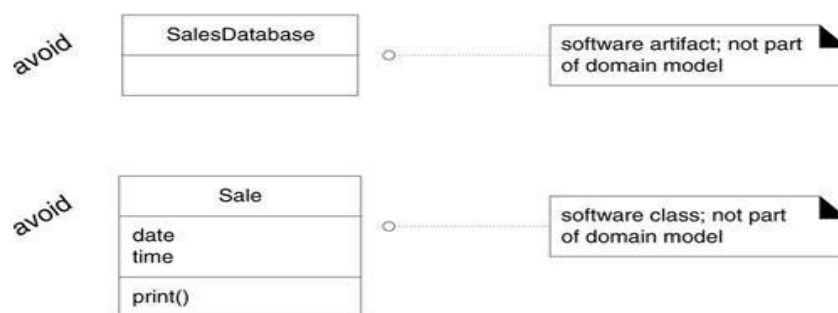
The domain model is a visual dictionary of the noteworthy abstractions, domain vocabulary, and information content of the domain.

A UP Domain Model is a visualization of things in a real-situation domain of interest, not of software objects such as Java or C# classes, or software objects with responsibilities. Therefore, the following elements are not suitable in a domain model:

- Software artifacts, such as a window or a database, unless the domain being modeled are of software concepts, such as a model of graphical user interfaces.
- Responsibilities or methods.



A domain model shows real –situation conceptual classes, not software classes



A domain model does not show software artifacts or classes

Two Traditional Meaning of Domain Model

Meaning 1 : "Domain Model" is a conceptual perspective of objects in a real situation of the world, not a software perspective.

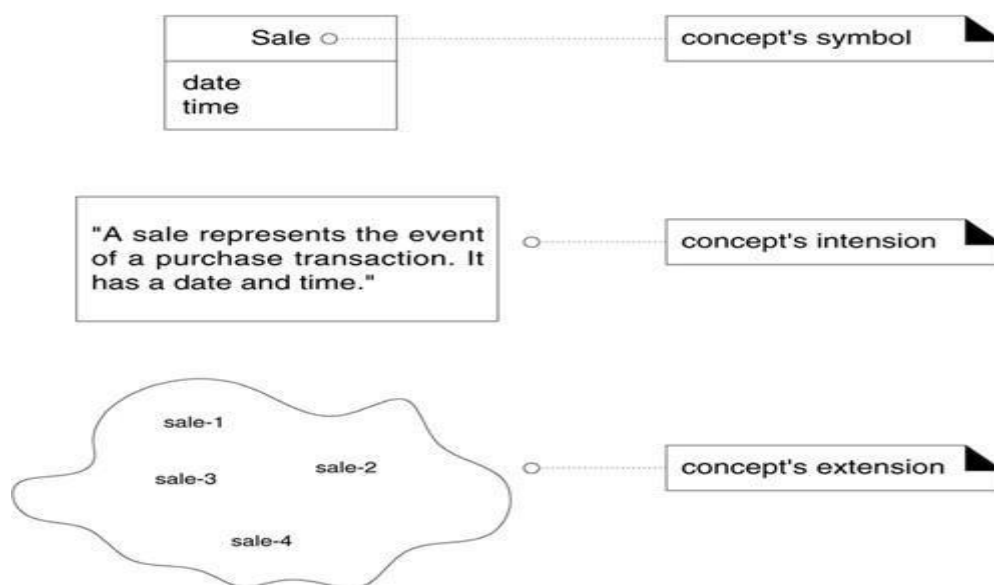
Meaning 2 : "the domain layer of software objects." That is, the layer of software objects below the presentation or UI layer that is composed of domain objects software objects that represent things in the problem domain space with related "business logic" or "domain logic" methods.

CONCEPTUAL CLASSES

A conceptual class is an idea, thing, or object. It may be considered in terms of its symbol, intension, and extension (see Figure).

- Symbol words or images representing a conceptual class.
- Intension the definition of a conceptual class.
- Extension the set of examples to which the conceptual class applies.

For example, consider the conceptual class for the event of a purchase transaction. I may choose to name it by the (English) symbol Sale. The intension of a Sale may state that it "represents the event of a purchase transaction, and has a date and time." The extension of Sale is all the examples of sales; in other words, the set of all sale instances in the universe.



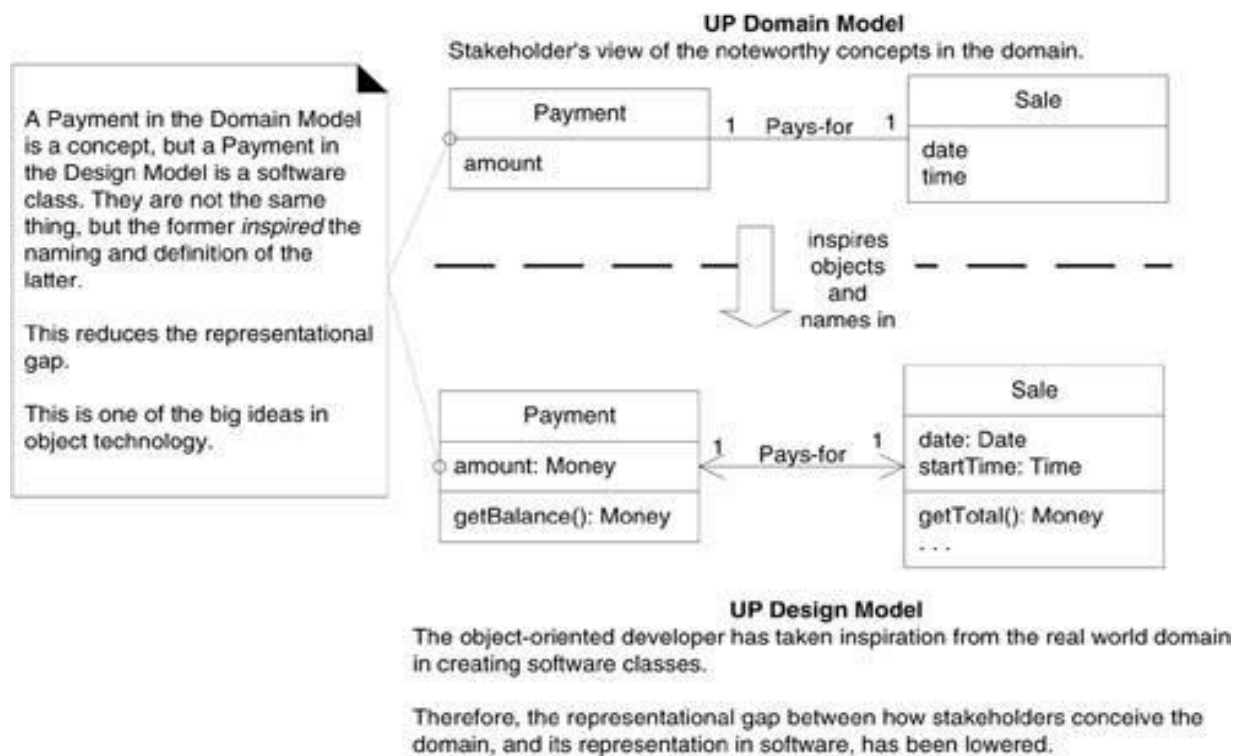
A conceptual class has a symbol, intension and extension

Domain and Data Models the Same Thing?

A domain model is not a data model (which by definition shows persistent data to be stored somewhere), so do not exclude a class simply because the requirements don't indicate any obvious need to remember information about it or because the conceptual class has no attributes. For example, it's valid to have attribute less conceptual classes, or conceptual classes that have a purely behavioral role in the domain instead of an information role.

Motivation: Why Create a Domain Model ?

Lower Representational Gap with OO Modeling : This is a key idea in OO: Use software class names in the domain layer inspired from names in the domain model, with objects having domain-familiar information and responsibilities. This supports a low representational gap between our mental and software models.



Lower Representational Gap with OO Modeling

Guideline: How to Create a Domain Model?

Bounded by the current iteration requirements under design:

1. Find the conceptual classes (see a following guideline).
2. Draw them as classes in a UML class diagram.
3. Add associations and attributes.

Guideline: To Find Conceptual Classes

Three Strategies to Find Conceptual Classes :

1. Reuse or modify existing models. This is the first, best, and usually easiest approach. There are published, well-crafted domain models and data models for many common domains, such as inventory, finance, health, and so forth.
2. Use a category list. (Method 2)
3. Identify noun phrases. (Method 3)

Method 2: Use a Category List

We can create a domain model by making a list of candidate conceptual classes. The guidelines also suggest some priorities in the analysis. Examples are drawn from the 1) POS, 2) Monopoly game 3) airline reservation domains.

Table - Conceptual Class Category List.

Conceptual Class Category	Examples
business transactions Guideline: These are critical (they involve money), so start with transactions.	Sale, Payment Reservation
transaction line items Guideline: Transactions often come with related line items, so consider these next.	SalesLineItem
product or service related to a transaction or transaction line item Guideline: Transactions are for something (a product or service). Consider these next.	Item Flight, Seat, Meal
where is the transaction recorded? Guideline: Important.	Register, Ledger FlightManifest
roles of people or organizations related to the transaction; actors in the use case Guideline: We usually need to know about the parties involved in a transaction.	Cashier, Customer, Store MonopolyPlayer Passenger, Airline
place of transaction; place of service	Store Airport, Plane, Seat
noteworthy events, often with a time or place we need to remember	Sale, Payment MonopolyGame Flight
physical objects Guideline: This is especially relevant when creating device-control software, or simulations.	Item, Register Board, Piece, Die Airplane
descriptions of things	ProductDescription FlightDescription
Guideline: Descriptions are often in a catalog.	ProductCatalog FlightCatalog
containers of things (physical or information)	Store, Bin Board Airplane

Table - Conceptual Class Category List.	
Conceptual Class Category	Examples
things in a container	Item Square (in a Board) Passenger
other collaborating systems	CreditAuthorizationSystem AirTrafficControl
records of finance, work, contracts, legal matters	Receipt, Ledger MaintenanceLog
financial instruments	Cash, Check, LineOfCredit TicketCredit
schedules, manuals, documents that are regularly referred to in order to perform work	DailyPriceChangeList RepairSchedule

Method 3: Finding Conceptual Classes with Noun Phrase Identification

Another useful technique (because of its simplicity) suggested is linguistic analysis: Identify the nouns and noun phrases in textual descriptions of a domain, and consider them as candidate conceptual classes or attributes.

Guideline

Linguistic analysis has become more sophisticated; it also goes by the name natural language modeling. for example, The current scenario of the Process Sale use case can be used.

Main Success Scenario (or Basic Flow):

1. **Customer** arrives at a **POS checkout** with goods and/or **services** to purchase.
2. **Cashier** starts a new **sale**.
3. **Cashier** enters item **identifier**.
4. System records **sale line item** and presents **item description** , **price**, and running **total**. Price calculated from a set of price rules.

Underlined words are nouns. The next level of scrutiny derives class names.

Example: Find and Draw Conceptual Classes

Case Study: POS Domain

From the category list and noun phrase analysis, a list is generated of candidate conceptual classes for the domain. There is no such thing as a "correct" list. It is a somewhat arbitrary collection of abstractions and domain vocabulary

Initial POS domain model.



Guidelines

1. Agile Modeling Sketching a Class Diagram : The sketching style in the UML class diagram is to keep the bottom and right sides of the class boxes open. This makes it easier to grow the classes as we discover new elements.

2. Agile Modeling Maintain the Model in a Tool? The purpose of creating a domain model is to quickly understand and communicate a rough approximation of the key concepts.

3. Report Objects - Include 'Receipt' in the Model? Receipt is a term in the POS domain. But it's only a report of a sale and payment, and thus duplicate information.

4. Use Domain Terms :

Make a domain model in the spirit of how a cartographer or mapmaker works:

- Use the existing names in the territory. For example, if developing a model for a library, name the customer a "Borrower" or "Patron" the terms used by the library staff.
- Exclude irrelevant or out-of-scope features. For example, in the Monopoly domain model for iteration-1
- Do not add things that are not there.

5. How to Model the Unreal World? Some software systems are for domains that find very little analogy in natural or business domains; software for

telecommunications is an example. For example, here are candidate conceptual classes related to the domain of a telecommunication switch: Message, Connection, Port, Dialog, Route, and Protocol.

6. A Common Mistake with Attributes vs. Classes If we do not think of some conceptual class X as a number or text in the real world, X is probably a conceptual class, not an attribute. As an example, should store be an attribute of Sale, or a separate conceptual class Store?



In the real world, a store is not considered a number or text the term suggests a legal entity, an organization, and something that occupies space. Therefore, Store should be a conceptual class.

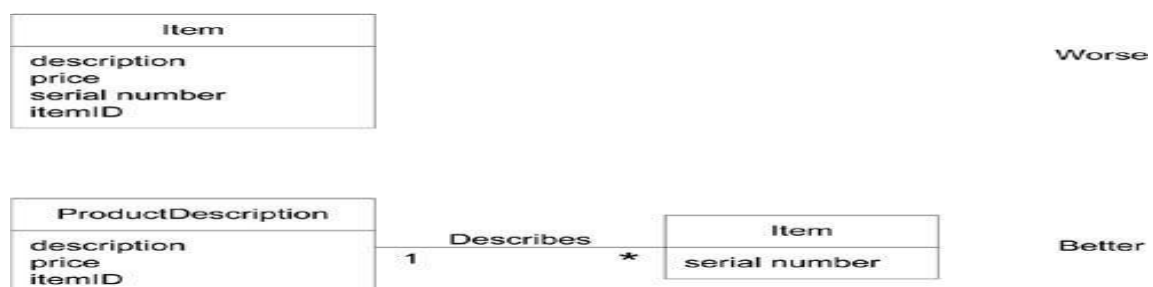
As another example, consider the domain of airline reservations. Should destination be an attribute of Flight, or a separate conceptual class Airport?



In the real world, a destination airport is not considered a number or text-it is a massive thing that occupies space. Therefore, Airport should be a concept.

7 When to Model with 'Description' Classes? A description class contains information that describes something else. For example, a Product Description that records the price, picture, and text description of an Item.

Motivation: Why Use 'Description' Classes? The need for description classes is common in many domain models. The need for description classes is common in sales, product, and service domains. It is also common in manufacturing, which requires a description of a manufactured thing that is distinct from the thing itself Figure. Descriptions about other things. The * means a multiplicity of "many." It indicates that one Product Description may describe many (*) Items.



When Are Description Classes Useful?

Add a description class (for example, Product Description) when:

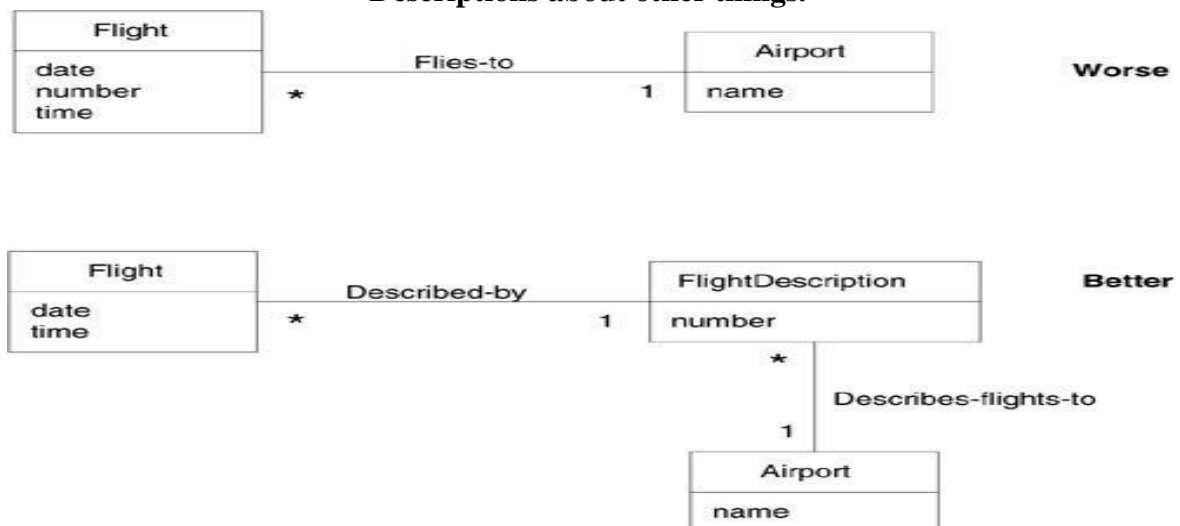
- There needs to be a description about an item or service, independent of the current existence of any examples of those items or services.
- Deleting instances of things they describe (for example, Item) results in a loss of information that needs to be maintained, but was incorrectly associated with the deleted thing.
- It reduces redundant or duplicated information.

Example: Descriptions in the Airline Domain

As another example, consider an airline company that suffers a fatal crash of one of its planes. Assume that all the flights are cancelled for six months pending completion of an investigation. Also assume that when flights are cancelled, their corresponding Flight software objects are deleted from computer memory. Therefore, after the crash, all Flight software objects are deleted.

If the only record of what airport a flight goes to is in the Flight software instances, which represent specific flights for a particular date and time, then there is no longer a record of what flight routes the airline has. The problem can be solved, both from a purely conceptual perspective in a domain model and from a software perspective in the software designs, with a FlightDescription that describes a flight and its route, even when a particular flight is not scheduled in following figure

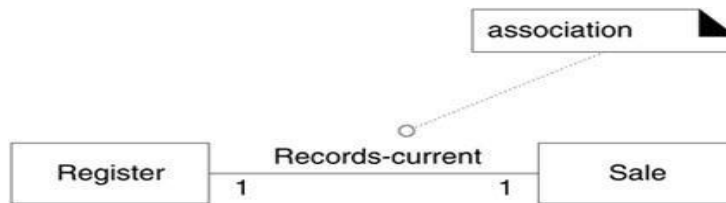
Descriptions about other things.



Note that the prior example is about a service (a flight) rather than a good. Descriptions of services or service plans are commonly needed.

Associations

An **association** is a relationship between classes (more precisely, instances of those classes) that indicates some meaningful and interesting connection (see Figure)



In the UML, associations are defined as "the semantic relationship between two or more classifiers that involve connections among their instances."

Include the following associations in a domain model:

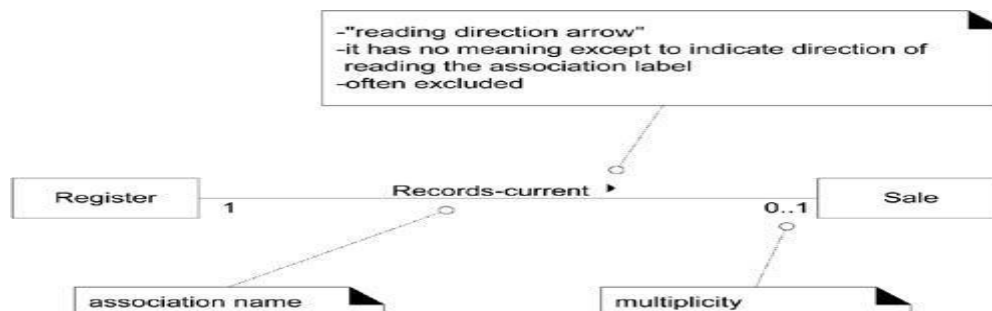
- Associations for which knowledge of the relationship needs to be preserved for some duration ("need-to-remember" associations).
- Associations derived from the Common Associations List.

Guideline 1. Avoid Adding Many Associations

- We need to avoid adding too many associations to a domain model. In a graph with n nodes, there can be $(n \cdot (n-1))/2$ associations to other nodes—a potentially very large number. A domain model with 20 classes could have 190 associations lines!
- During domain modeling, an association is not a statement about data flows, database foreign key relationships, instance variables, or object connections in a software solution; it is a statement that a relationship is meaningful in a purely conceptual perspective—in the real domain.

Applying UML: Association Notation

An association is represented as a line between classes with a capitalized association name. See Figure



The UML notation for associations.

The ends of an association may contain a multiplicity expression indicating the numerical relationship between instances of the classes.

The association is inherently bidirectional, meaning that from instances of either class, logical traversal to the other is possible. This traversal is purely abstract; it is not a statement about connections between software entities.

An optional "reading direction arrow" indicates the direction to read the association name; it does not indicate direction of visibility or navigation. If the arrow is not present, the convention is to read the association from left to right or top to bottom.

Guideline 2: To Name an Association in UML

Name an association based on a ClassName-VerbPhrase - ClassName format where the verb phrase creates a sequence that is readable and meaningful. Simple association names such as "Has" or "Uses" are usually poor, as they seldom enhance our understanding of the domain.

For example,

- Sale Paid-by CashPayment
 - bad example (doesn't enhance meaning): Sale Uses CashPayment
- Player Is-on Square
 - bad example (doesn't enhance meaning): Player Has Square

Association names should start with a capital letter, since an association represents a classifier of links between instances; in the UML, classifiers should start with a capital letter.

Applying UML: Roles

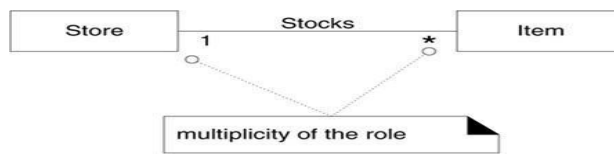
Each end of an association is called a **role**. Roles may optionally have:

- multiplicity expression
- name
- navigability

Applying UML: Multiplicity

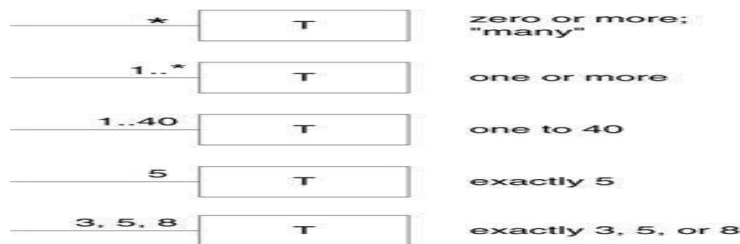
Multiplicity defines how many instances of a class A can be associated with one instance of a class B

Multiplicity on an association.



For example, a single instance of a Store can be associated with "many" (zero or more, indicated by the *) Item instances.

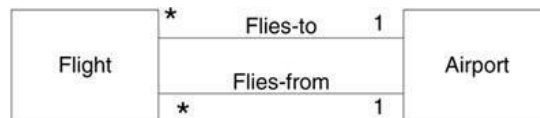
Multiplicity values.



Applying UML: Multiple Associations Between Two Classes

The domain of the airline is the relationships between a Flight and an Airport. The flying-to and flying-from associations are distinctly different relationships, which should be shown separately.

Multiple associations.



Guideline 3 : To Find Associations with a Common Associations List

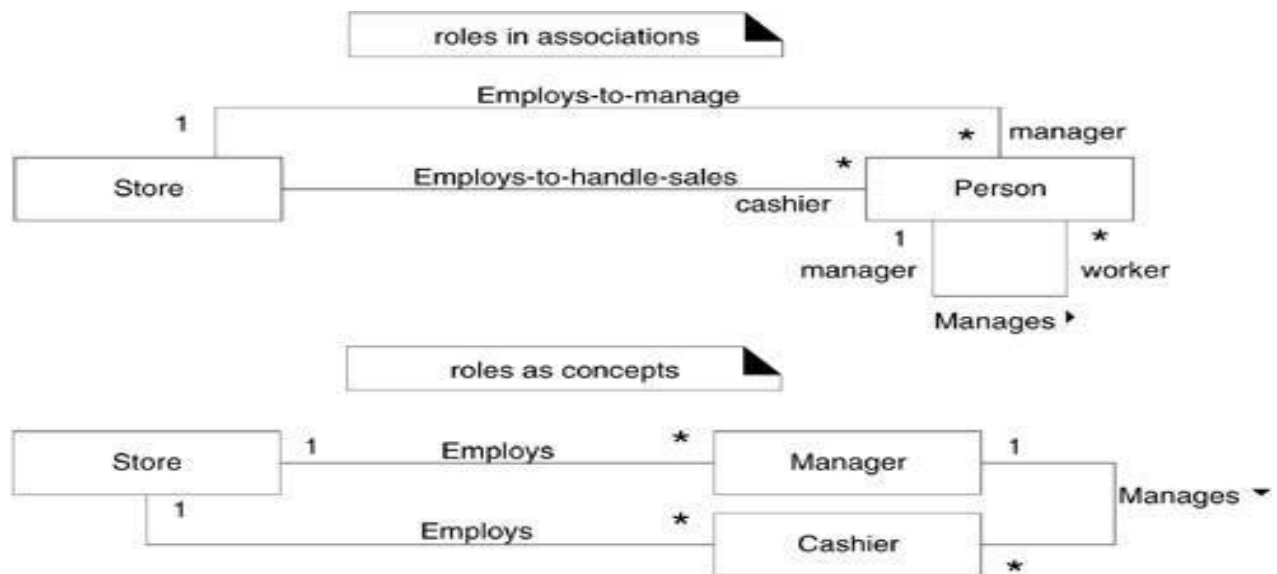
Start the addition of associations by using the list in Table . It contains common categories that are worth considering, especially for business information systems. Examples are drawn from the 1) POS, 2) Monopoly, and 3) airline reservation domains.

Table - Common Associations List.

Category	Examples
A is a transaction related to another transaction B	CashPaymentSale CancellationReservation
A is a line item of a transaction B	SalesLineItemSale

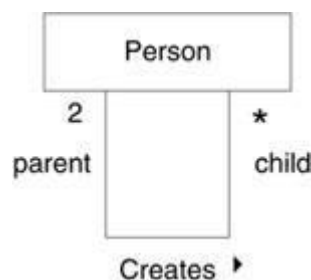
Category	Examples
A is a product or service for a transaction (or line item) B	ItemSalesLineItem(or Sale)FlightReservation
A is a role related to a transaction B	CustomerPayment PassengerTicket
A is a physical or logical part of B	DrawerRegister SquareBoard SeatAirplane
A is physically or logically contained in/on B	RegisterStore, ItemShelf SquareBoard PassengerAirplane
A is a description for B	ProductDescriptionItem FlightDescriptionFlight
A is known / logged / recorded / reported / captured in B	SaleRegister PieceSquare ReservationFlightManifest
A is a member of B	CashierStore PlayerMonopolyGame PilotAirline
A is an organizational subunit of B	DepartmentStore MaintenanceAirline
A uses or manages or owns B	CashierRegister PlayerPiece PilotAirplane
A is next to B	SalesLineItemSalesLineItem SquareSquare CityCity

Roles as Concepts versus Roles in Associations :In a domain model, a real-world role especially a human role may be modeled in a number of ways, such as a discrete concept, or expressed as a role in an association. For example, the role of cashier and manager may be expressed in at least the two ways illustrated in Fig



Qualified Associations (Refer pg no 9)

Reflexive Associations : A concept may have an association to itself; this is known as a reflexive association

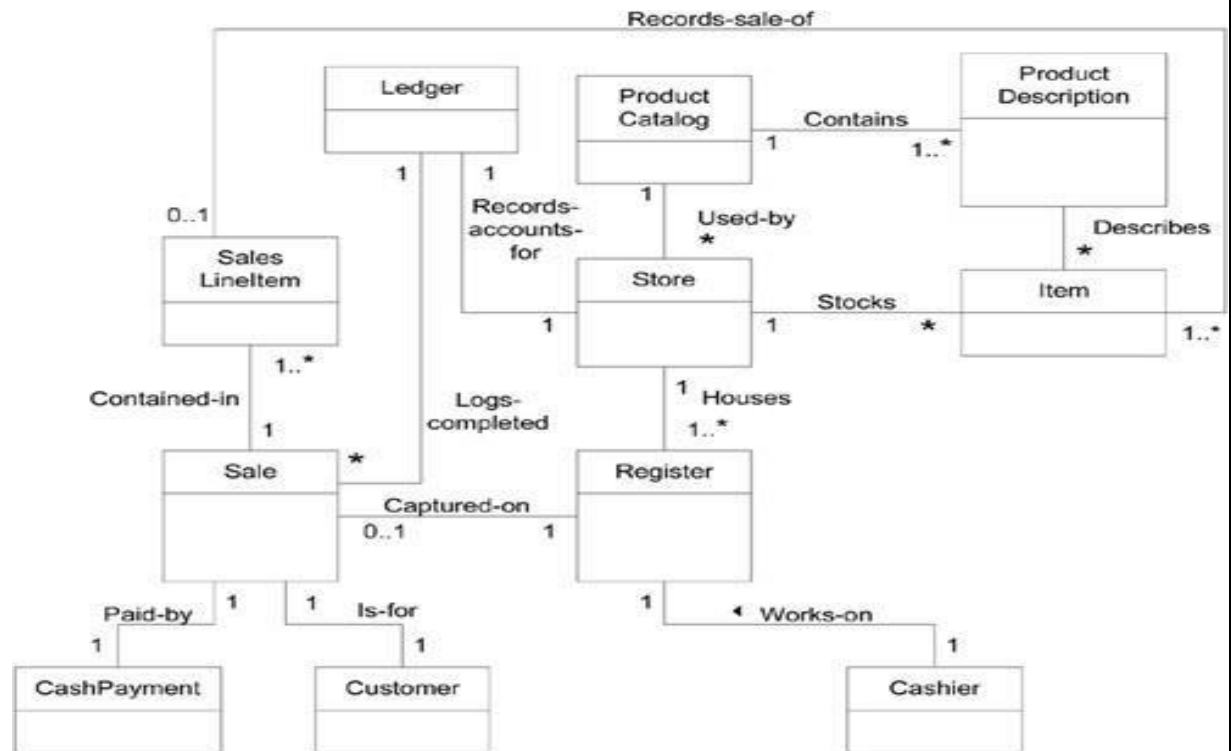


Example: Associations in the Domain Models

Case Study: NextGen POS : The domain model in Figure shows a set of conceptual classes and associations that are candidates for our POS domain model. The associations are primarily derived from the "need-to-remember" criteria of these iteration requirements, and the Common Association List. For example:

- Transactions related to another transaction Sale Paid-by CashPayment.
- Line items of a transaction Sale Contains SalesLineItem.
- Product for a transaction (or line item) SalesLineItem Records-sale-of Item.

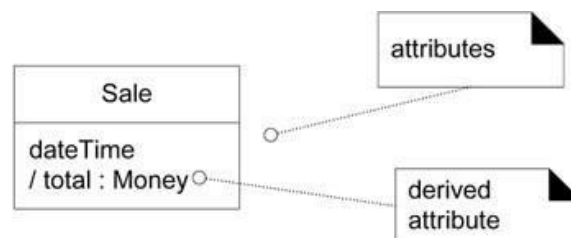
NextGen POS partial domain model.



Attributes : An **attribute** is a logical data value of an object. Include attributes that the requirements (for example, use cases) suggest or imply a need to remember information. For example, a receipt (which reports the information of a sale) in the Process Sale use case normally includes Therefore,

- Sale needs a dateTime attribute.
- Store needs a name and address.
- Cashier needs an ID.

Applying UML- Attribute Notation : Attributes are shown in the second compartment of the class box . Their type and other information may optionally be shown.



Class and attributes.

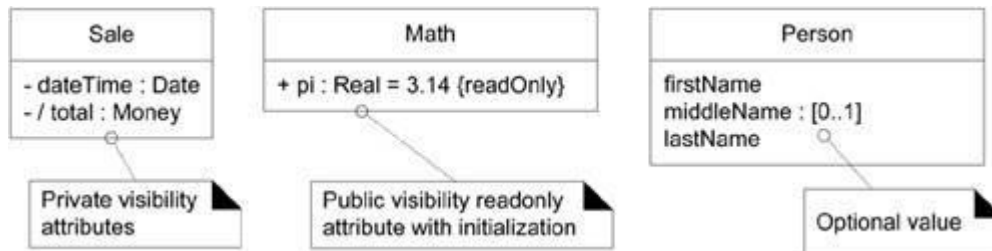
More Notation

The full syntax for an attribute in the UML is:

visibility name : type multiplicity = default {property-string}

Some common examples are shown in Fig

Attribute notation in UML.



`{readOnly}` is probably the most common property string for attributes.

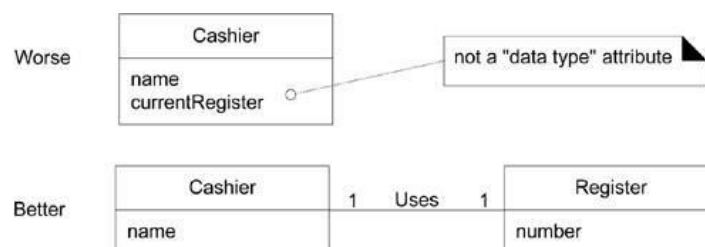
Multiplicity can be used to indicate the optional presence of a value, or the number of objects that can fill a (collection) attribute.

Derived Attributes : When we want to communicate that 1) this is a noteworthy attribute, but 2) it is derivable, we use the UML convention: a `/` symbol before the attribute name.

Guideline 1 : Suitable Attribute Types - Focus on Data Type Attributes in the Domain Model

Most attribute types should be what are often thought of as "primitive" data types, such as numbers and Booleans. For example, the current Register attribute in the Cashier class in Figure is undesirable because its type is meant to be a Register, which is not a simple data type (such as Number or String).

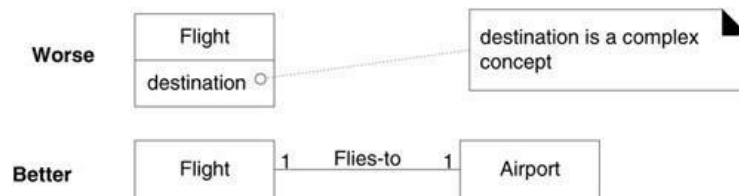
Relate with associations, not attributes



Guideline : The attributes in a domain model should preferably be data types. Very common data types include: Boolean, Date (or DateTime), Number, Character, String (Text), Time. Other common types include: Address, Color, Geometrics (Point, Rectangle), Phone Number, Social Security Number, Universal Product Code (UPC), SKU, ZIP or postal codes, enumerated types

A common confusion is modeling a complex domain concept as an attribute. To illustrate, a destination airport is not really a string; it is a complex thing that occupies many square kilometers of space. Therefore, Flight should be related to Airport via an association, not with an attribute, as shown in Fig.

Don't show complex concepts as attributes; use associations.



Guideline : Relate conceptual classes with an association, not with an attribute.

Data Types

Attributes in the domain model should generally be data types; informally these are "primitive" types such as number, boolean, character, string, and enumerations (such as Size = {small, large}).

For example, it is not (usually) meaningful to distinguish between:

- Separate instances of the Integer 5.
- Separate instances of the String 'cat'.
- Separate instance of the Date "Nov. 13, 1990".

Guideline 1 : When to define New Data type Classes ?

Guidelines for modeling data types

Represent what may initially be considered a number or string as a new data type class in the domain model if:

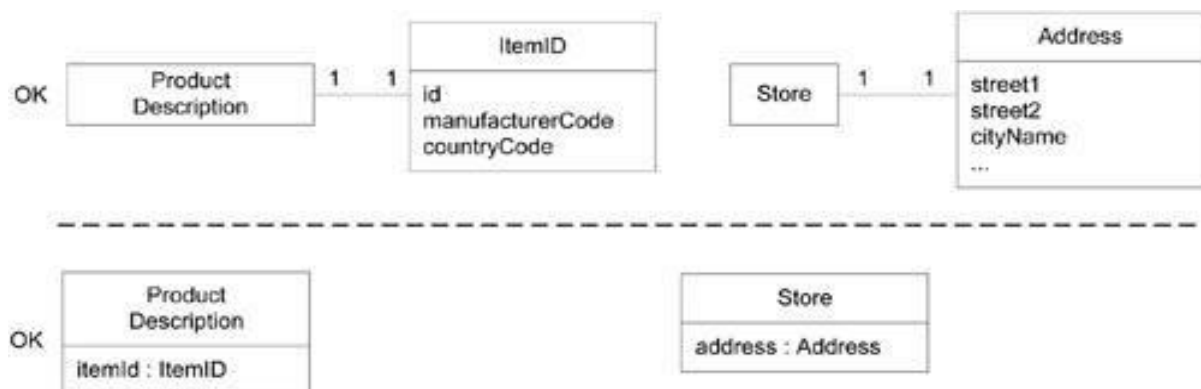
- It is composed of separate sections. –Ex phone number, name of person
- There are operations associated with it, such as parsing or validation. - social security number
- It has other attributes. - promotional price could have a start (effective) date and end date
- It is a quantity with a unit. - payment amount has a unit of currency
- It is an abstraction of one or more types with some of these qualities.

Item identifier in the sales domain is a generalization of types such as Universal Product Code (UPC) and European Article Number (EAN)

Applying these guidelines to the POS domain model attributes yields the following analysis:

- The item identifier is an abstraction of various common coding schemes, including UPC-A, UPC-E, and the family of EAN schemes. These numeric coding schemes have subparts identifying the manufacturer, product, country (for EAN), and a check-sum digit for validation. Therefore, there should be a data type ItemID class, because it satisfies many of the guidelines above.
- The price and amount attributes should be a data type Money class because they are quantities in a unit of currency.
- The address attribute should be a data type Address class because it has separate sections.

Applying UML: Where to Illustrate These Data Type Classes?



Two ways to indicate a data type property of an object.

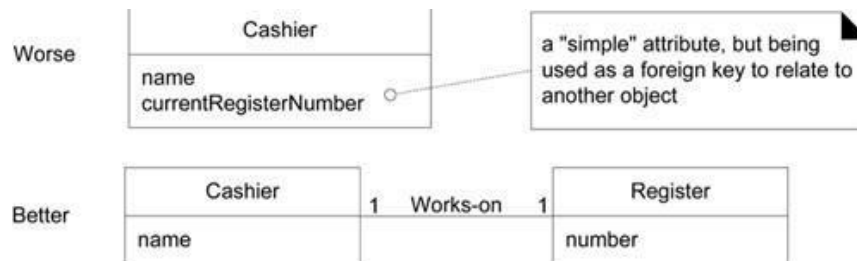
Should the ItemID class be shown as a separate class in a domain model?. Since ItemID is a data type (unique identity of instances is not used for equality testing), it may be shown only in the attribute compartment of the class box, as shown in above Figure. On the other hand, if ItemID is a new type with its own attributes and associations, showing it as a conceptual class in its own box may be informative.

Guideline 2 : No Attributes Representing Foreign Keys

In Following Fig the `currentRegisterNumber` attribute in the `Cashier` class is undesirable because its purpose is to relate the `Cashier` to a `Register` object. The

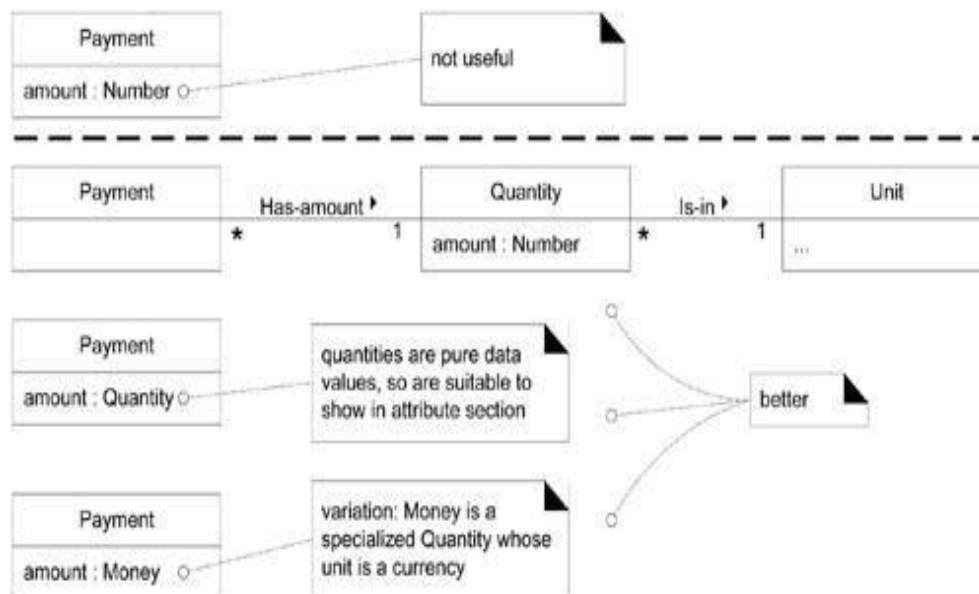
better way to express that a Cashier uses a Register is with an association, not with a foreign key attribute.

Do not use attributes as foreign keys.



Guideline 3 : Modeling Quantities and Units

Most numeric quantities should not be represented as plain numbers. Consider price or weight. These are quantities with associated units, and it is common to require knowledge of the unit to support conversions.



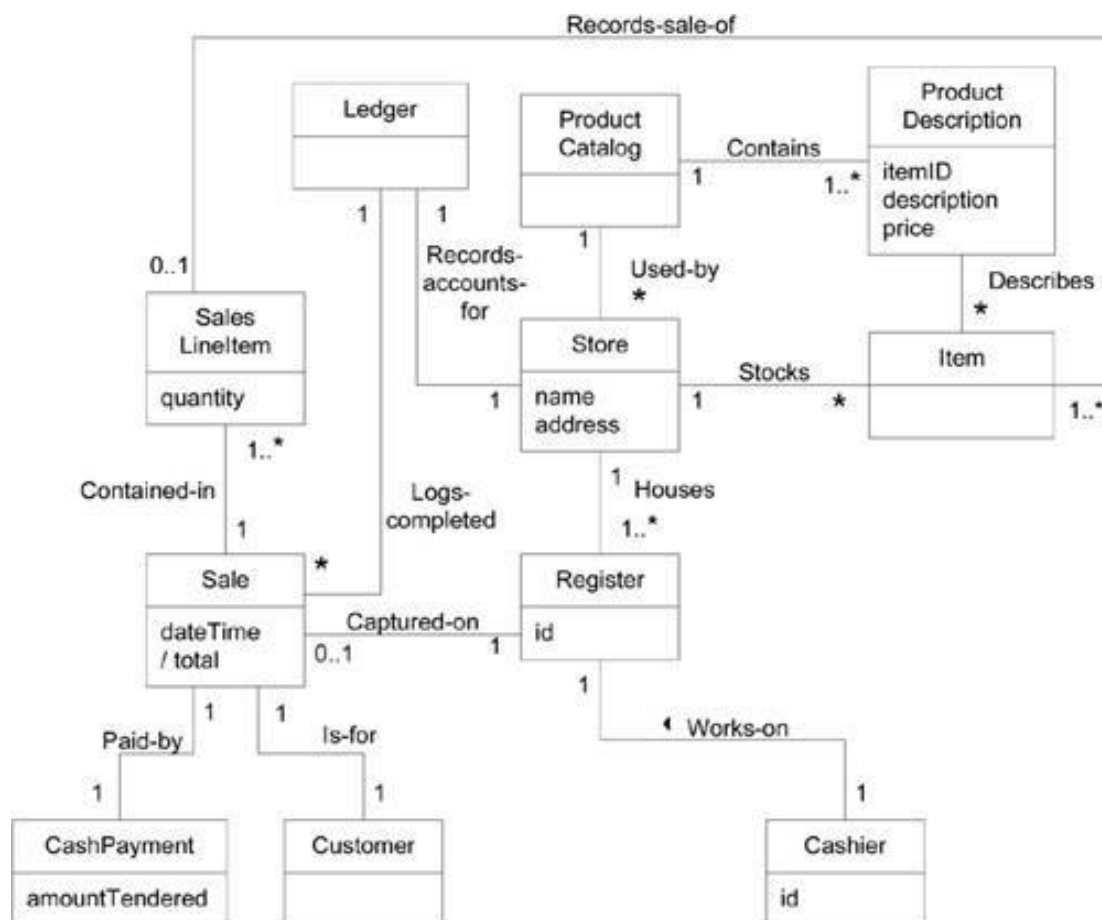
Modeling quantities.

Example: Attributes in the Domain Models -Case Study: NextGen POS

See following Fig. The attributes chosen reflect the information requirements for this iteration the Process Sale cash-only scenarios of this iteration. For example:

CashPayment	amountTendered To determine if sufficient payment was provided, and to calculate change, an amount (also known as "amount tendered") must be captured.
-------------	--

CashPayment	amountTendered To determine if sufficient payment was provided, and to calculate change, an amount (also known as "amount tendered") must be captured.
Product-Description	description To show the description on a display or receipt. itemId To look up a ProductDescription. price To calculate the sales total, and show the line item price.
Sale	dateTime A receipt normally shows date and time of sale, and this is useful for sales analysis.
SalesLineItem	quantity To record the quantity entered, when there is more than one item in a line item sale (for example, five packages of tofu).
Store	address, name The receipt requires the name and address of the store.



NextGen POS partial domain model.

DOMAIN MODEL REFINEMENT

OBJECTIVES

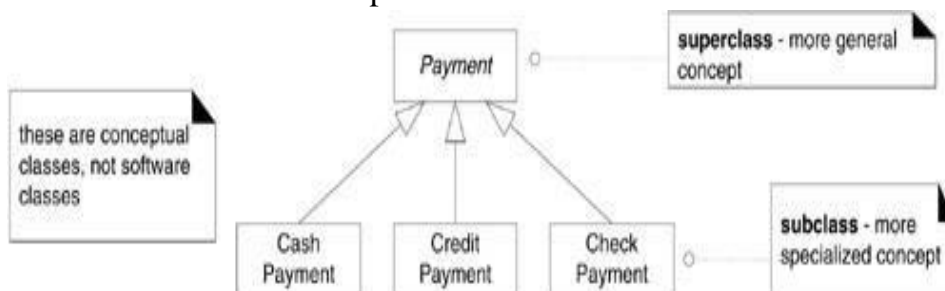
- Refine the domain model with generalizations, specializations, association classes, time intervals, composition, and packages.
- Generalization and specialization are fundamental concepts in domain modeling that support an economy of expression;
- Association classes capture information about an association itself.
- Time intervals capture the important concept that some business objects are valid for a limited time.
- Packages are a way to organize large domain models into smaller units.

Concepts Category List : This Table shows some concepts being considered in this iteration.

Category	Examples
physical or tangible objects	CreditCard, Check
Transactions	CashPayment, CreditPayment, CheckPayment
other computer or electro-mechanical systems external to our system	CreditAuthorizationService, CheckAuthorizationService
abstract noun concepts	
Organizations	CreditAuthorizationService, CheckAuthorizationService
records of finance, work, contracts, legal matters	AccountsReceivable

Generalization

The concepts CashPayment, CreditPayment, and CheckPayment are all very similar. In this situation, it is possible (and useful) to organize them (as in following Figure) into a generalization-specialization class hierarchy (or simply **class hierarchy**) in which the **super class** Payment represents a more general concept, and the **subclasses** more specialized ones.



Generalization-specialization hierarchy.

Generalization is the activity of identifying commonality among concepts and defining superclass (general concept) and subclass (specialized concept) relationships. Identifying a superclass and subclasses is of value in a domain model because their presence allows us to understand concepts in more general, refined and abstract terms.

Guideline : Identify domain superclasses and subclasses relevant to the current iteration, and illustrate them in the Domain Model.



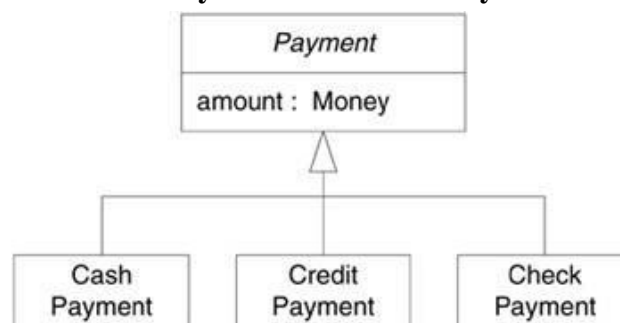
Class hierarchy with separate and shared arrow notations.

Defining Conceptual Superclasses and Subclasses :

Definition : A conceptual super class definition is more general or encompassing than a subclass definition.

For example, consider the superclass Payment and its subclasses (CashPayment, and so on). Assume the definition of Payment is that it represents the transaction of transferring money (not necessarily cash) for a purchase from one party to another, and that all payments have an amount of money transferred. The model corresponding to this is shown in following Figure.

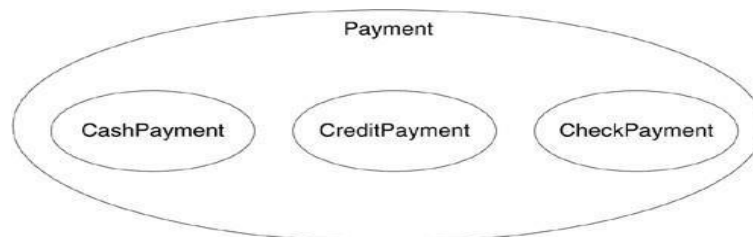
Payment class hierarchy.



A Credit Payment is a transfer of money via a credit institution which needs to be authorized. My definition of Payment encompasses and is more general than my definition of Credit Payment.

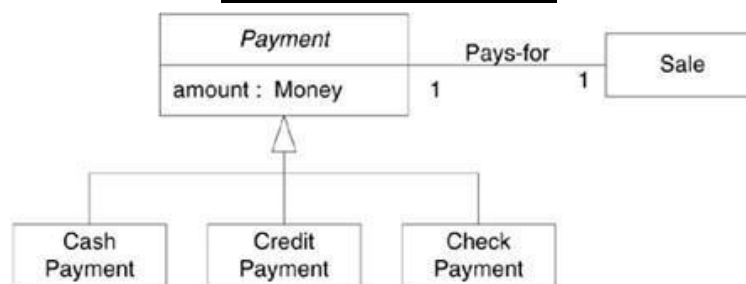
Definition : All members of a conceptual subclass set are members of their superclass set. For example, in terms of set membership, all instances of the set CreditPayment are also members of the set Payment. In a Venn diagram, this is shown as in following Fig

Venn diagram of set relationships.



Conceptual Subclass Definition Conformance : When a class hierarchy is created, statements about superclasses that apply to subclasses are made. For example, the following Figure states that all Payments have an amount and are associated with a Sale.

Subclass conformance.



Guideline: 100% Rule

100% of the conceptual superclass's definition should be applicable to the subclass. The subclass must conform to 100% of the superclass's:

- attributes
- associations

Conceptual Subclass Set Conformance : A conceptual subclass should be a member of the set of the superclass. Thus, CreditPayment should be a member of the set of Payments.

Guideline: Is-a Rule

All the members of a subclass set must be members of their superclass set.

In natural language, this can usually be informally tested by forming the statement: Subclass is a Superclass.

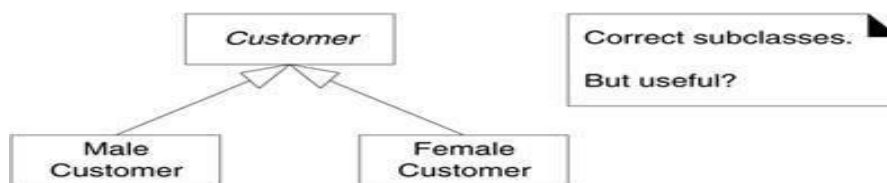
Guideline :Correct Conceptual Subclass

A potential subclass should conform to the:

- 100% Rule (definition conformance)
- Is-a Rule (set membership conformance)

When to Define a Conceptual Subclass?

Definition: A conceptual class partition is a division of a conceptual class into disjoint subclasses. For example, in the POS domain, Customer may be correctly partitioned (or subclassed) into MaleCustomer and FemaleCustomer. But is it relevant or useful to show this in our model (see following figure)? This partition is not useful for our domain; the next section explains why



Legal conceptual class partition, but is it useful in our domain

Motivations to Partition a Conceptual Class into Subclasses

Create a conceptual subclass of a superclass when:

1. The subclass has additional attributes of interest.
2. The subclass has additional associations of interest.
3. The subclass concept is operated on, handled, reacted to, or manipulated differently than the superclass or other subclasses, in ways that are of interest.
4. The subclass concept represents an animate thing (for example, animal, robot) that behaves differently than the superclass or other subclasses, in ways that are of interest.

Based on the above criteria, it is not compelling to partition Customer into the subclasses MaleCustomer and FemaleCustomer because they have no additional attributes or associations, are not operated on (treated) differently, and do not behave differently in ways that are of interest. This table shows some examples of class partitions from the domain of payments and other areas, using these criteria

Example subclass partitions

Conceptual Subclass Motivation	Examples
The subclass has additional attributes of interest.	Payments not applicable. Library Book, subclass of LoanableResource, has an ISBN attribute.

Conceptual Subclass Motivation	Examples
The subclass has additional associations of interest.	Payments CreditPayment, subclass of Payment, is associated with a CreditCard. Library Video, subclass of LoanableResource, is associated with Director.
The subclass concept is operated upon, handled, reacted to, or manipulated differently than the superclass or other subclasses, in ways that are of interest.	Payments CreditPayment, subclass of Payment, is handled differently than other kinds of payments in how it is authorized. Library Software, subclass of LoanableResource, requires a deposit before it may be loaned.
The subclass concept represents an animate thing (for example, animal, robot) that behaves differently than the superclass or other subclasses, in ways that are of interest.	Payments not applicable. Library not applicable. Market Research MaleHuman, subclass of Human, behaves differently than FemaleHuman with respect to shopping habits.

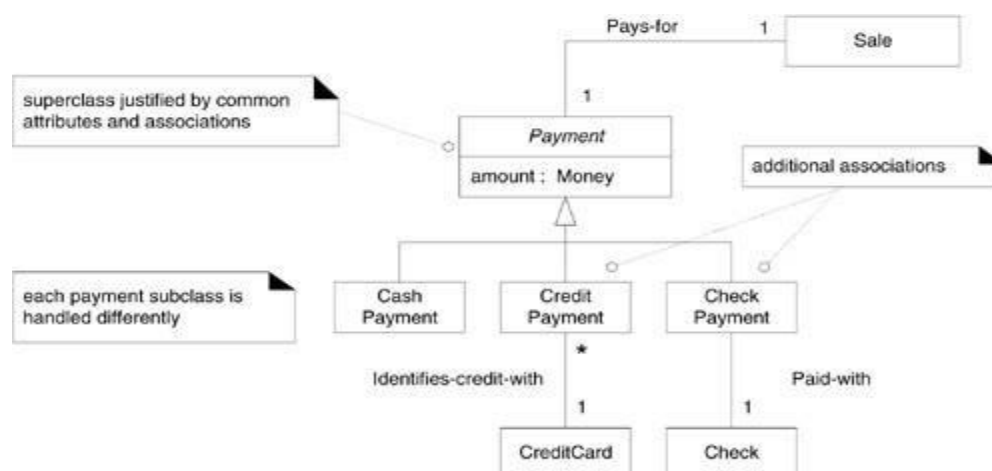
When to Define a Conceptual Superclass?

Motivations to generalize and define a superclass: Guideline

Create a superclass in a generalization relationship to subclasses when:

- The potential conceptual subclasses represent variations of a similar concept.
- The subclasses will conform to the 100% and Is-a rules.
- All subclasses have the same attribute that can be factored out and expressed in the superclass.
- All subclasses have the same association that can be factored out and related to the superclass.

NextGen POS Conceptual Class Hierarchies

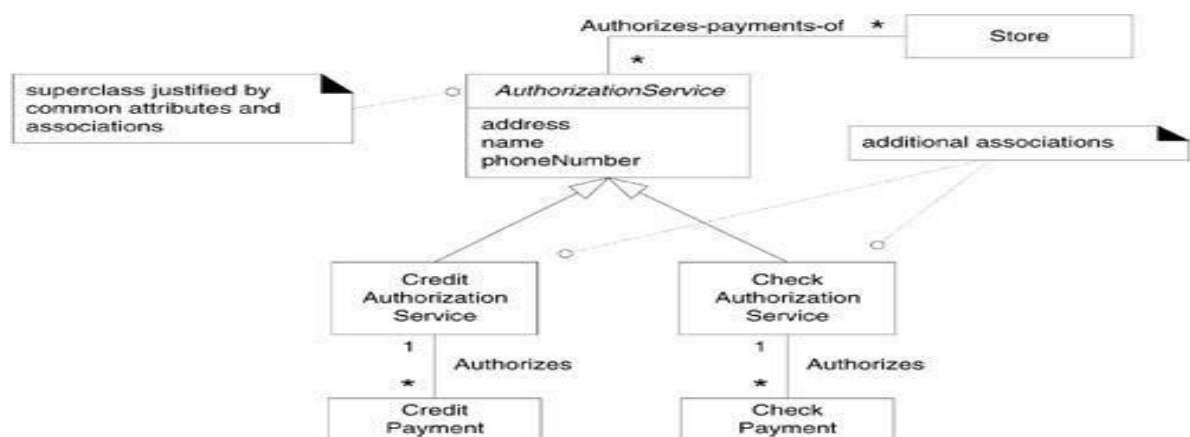


Justifying Payment subclasses.

Payment Classes : Based on the above criteria for partitioning the Payment class, it is useful to create a class hierarchy of various kinds of payments. The justification for the superclass and subclasses is shown in Figure .

Authorization Service Classes : Credit and check authorization services are variations on a similar concept, and have common attributes of interest. This leads to the class hierarchy in following Figure.

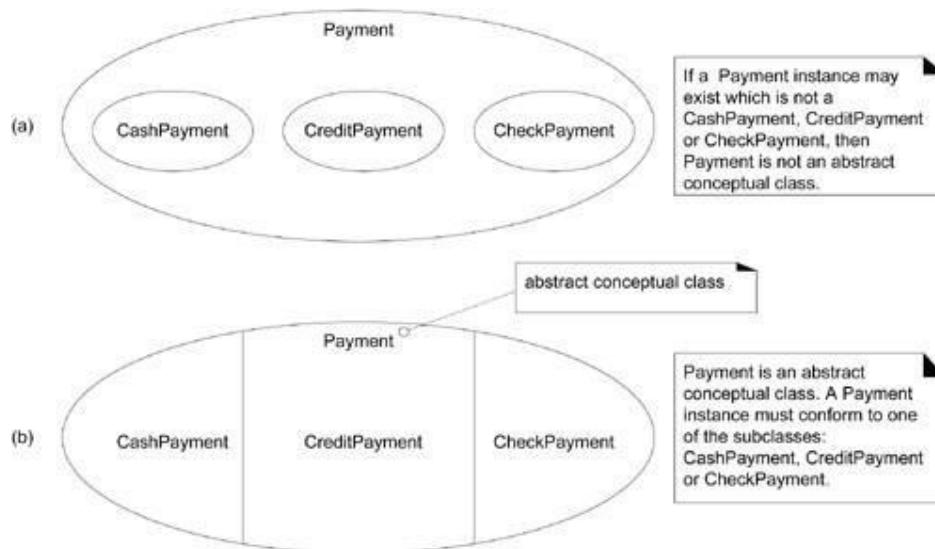
Justifying the AuthorizationService hierarchy



Abstract Conceptual Classes

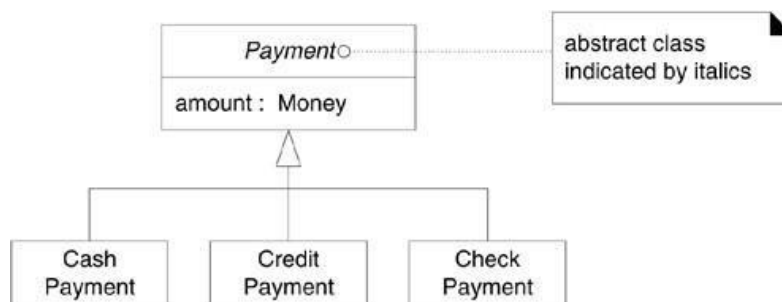
Definition : If every member of a class C must also be a member of a subclass, then class C is called an abstract conceptual class. For example, assume that every Payment instance must more specifically be an instance of the subclass CreditPayment, CashPayment, or CheckPayment. This is illustrated in the Venn diagram of Figure (b). Since every Payment member is also a member of a subclass, Payment is an abstract conceptual class by definition.

Abstract conceptual classes.



Abstract Class Notation in the UML : To review, the UML provides a notation to indicate abstract classes the class name is italicized

Abstract class notation.



Guideline : Identify abstract classes and illustrate them with an italicized name in the Domain Model, or use the {abstract} keyword.

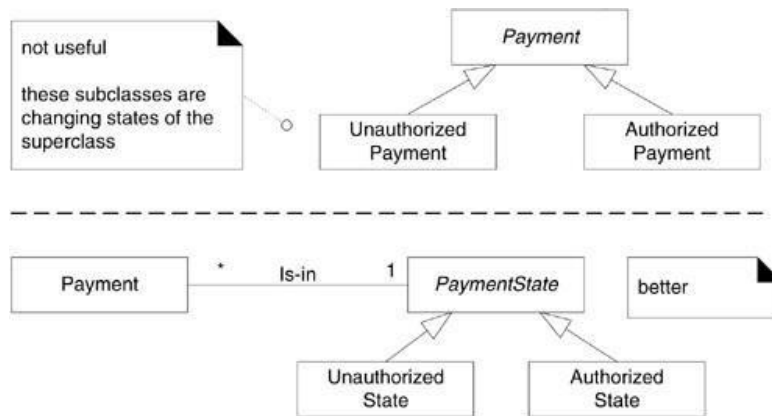
Modeling Changing States

Assume that a payment can either be in an unauthorized or authorized state, and it is meaningful to show this in the domain model. As shown in Figure , one modeling approach is to define subclasses of Payment: Unauthorized Payment and Authorized Payment.

Guideline : Do not model the states of a concept X as subclasses of X. Rather, either:

- Define a state hierarchy and associate the states with X, or
- Ignore showing the states of a concept in the domain model; show the states in state diagrams instead.

Modeling changing states.



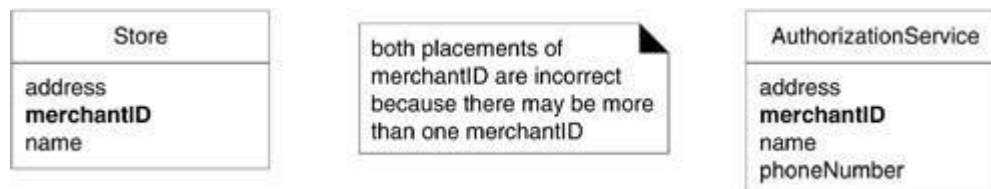
Association Classes

The following domain requirements set the stage for association classes:

- Authorization services assign a merchant ID to each store for identification during communications.
- A payment authorization request from the store to an authorization service needs the merchant ID that identifies the store to the service.
- Furthermore, a store has a different merchant ID for each service.

Placing merchantID in Store is incorrect because a Store can have more than one value for merchantID. The same is true with placing it in AuthorizationService (see Figure).

Inappropriate use of an attribute.



Guideline : In a domain model, if a class C can simultaneously have many values for the same kind of attribute A, do not place attribute A in C. Place attribute A in another class that is associated with C.

For example:

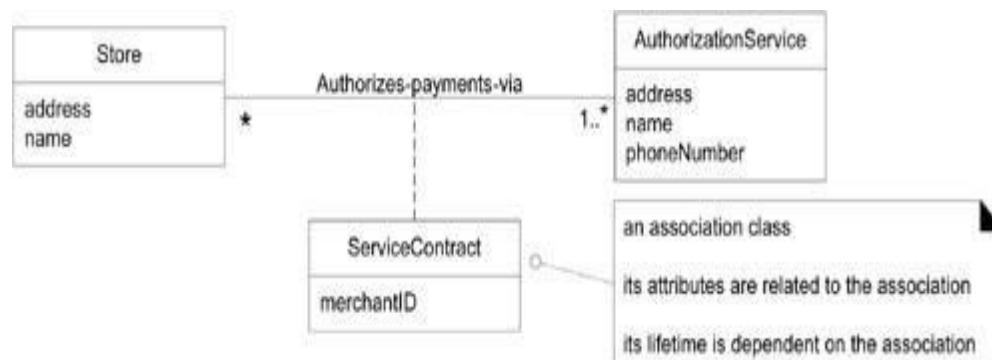
- A Person may have many phone numbers. Place phone number in another class, such as PhoneNumber or ContactInformation, and associate many of these to Person.

First attempt at modeling the merchantID problem.



The fact that both Store and AuthorizationService are related to ServiceContract is a clue that it is dependent on the relationship between the two. The merchantID may be thought of as an attribute related to the association between Store and AuthorizationService.

This leads to the notion of an association class, in which we can add features to the association itself. ServiceContract may be modeled as an association class related to the association between Store and AuthorizationService.



An association class

Guideline : Clues that an association class might be useful in a domain model:

- An attribute is related to an association.
- Instances of the association class have a lifetime dependency on the association.
- There is a many-to-many association between two concepts and information associated with the association itself.

AGGREGATION AND COMPOSITION (Refer Pg No: 7)

How to Identify Composition : Guideline

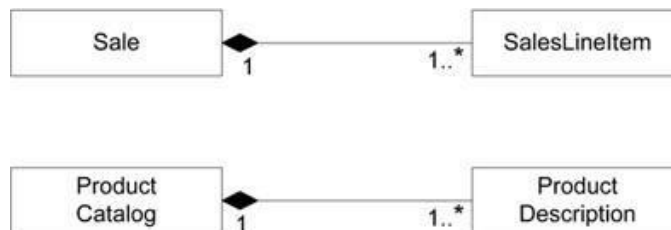
Consider showing composition when:

- The lifetime of the part is bound within the lifetime of the composite there is a create-delete dependency of the part on the whole.
- There is an obvious whole-part physical or logical assembly.
- Some properties of the composite propagate to the parts, such as the location.
- Operations applied to the composite propagate to the parts, such as destruction, movement, and recording.

Composition in the NextGen Domain Model

In the POS domain, the SalesLineItems may be considered a part of a composite Sale;

Aggregation in the point-of-sale application.



SYSTEM SEQUENCE DIAGRAMS

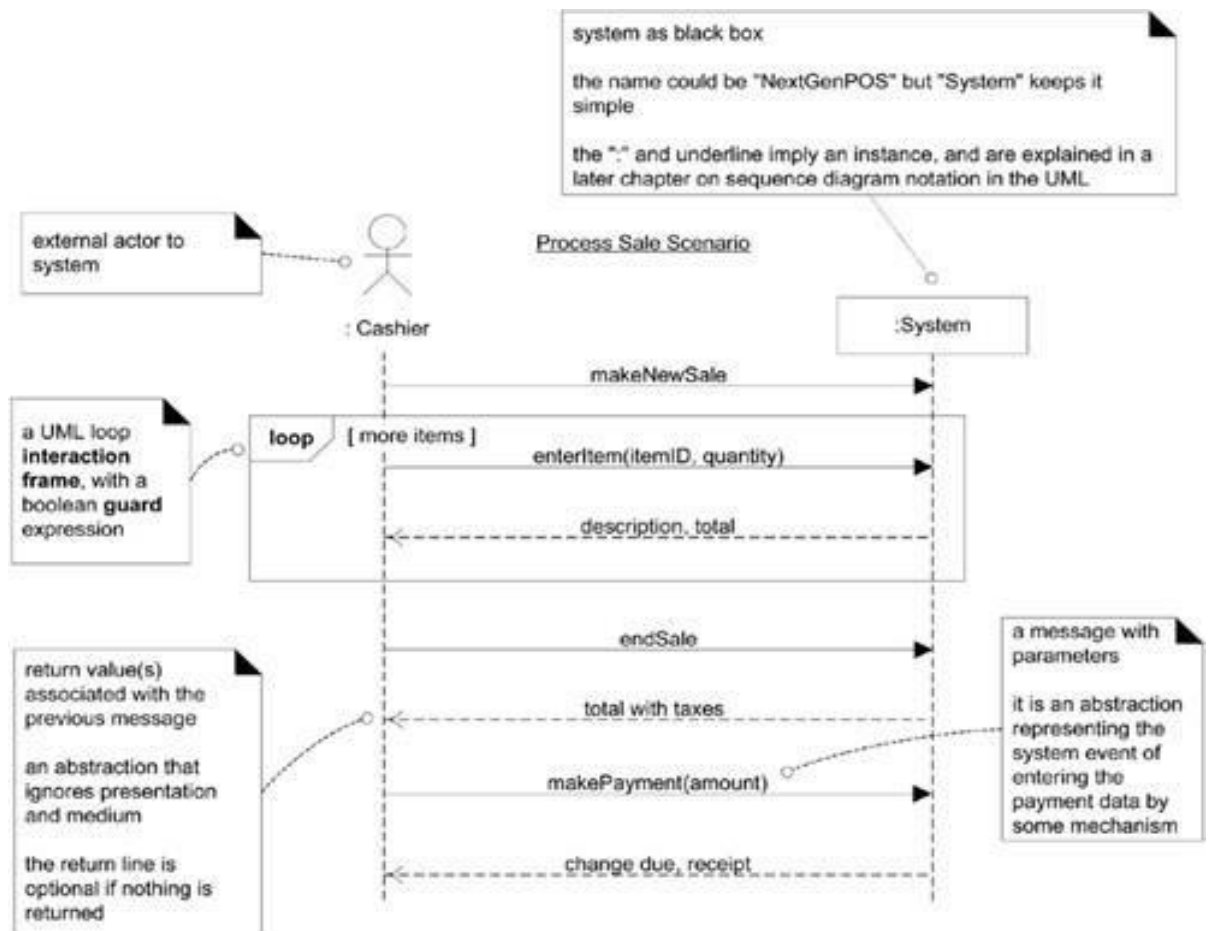
Use cases describe how external actors interact with the software system we are interested in creating. During this interaction an actor generates system events to a system, usually requesting some system operation to handle the event.

For example, when a cashier enters an item's ID, the cashier is requesting the POS system to record that item's sale (the enterItem event). That event initiates an operation upon the system. The use case text implies the enterItem event, and the SSD makes it concrete and explicit.

A system sequence diagram is a picture that shows, for one particular scenario of a use case, the events that external actors generate their order, and inter-system events. All systems are treated as a black box.

Guideline : Draw an SSD for a main success scenario of each use case, and frequent or complex alternative scenarios.

SSD for a Process Sale scenario.



Why Draw an SSD?

A software system reacts to three things:

- 1) external events from actors (humans or computers),
- 2) timer events,
- 3) faults or exceptions (which are often from external sources).

Therefore, it is useful to know what, precisely, are the external input events the system events. They are an important part of analyzing system behavior.

System behavior is a description of what a system does, without explaining how it does it. One part of that description is a system sequence diagram.

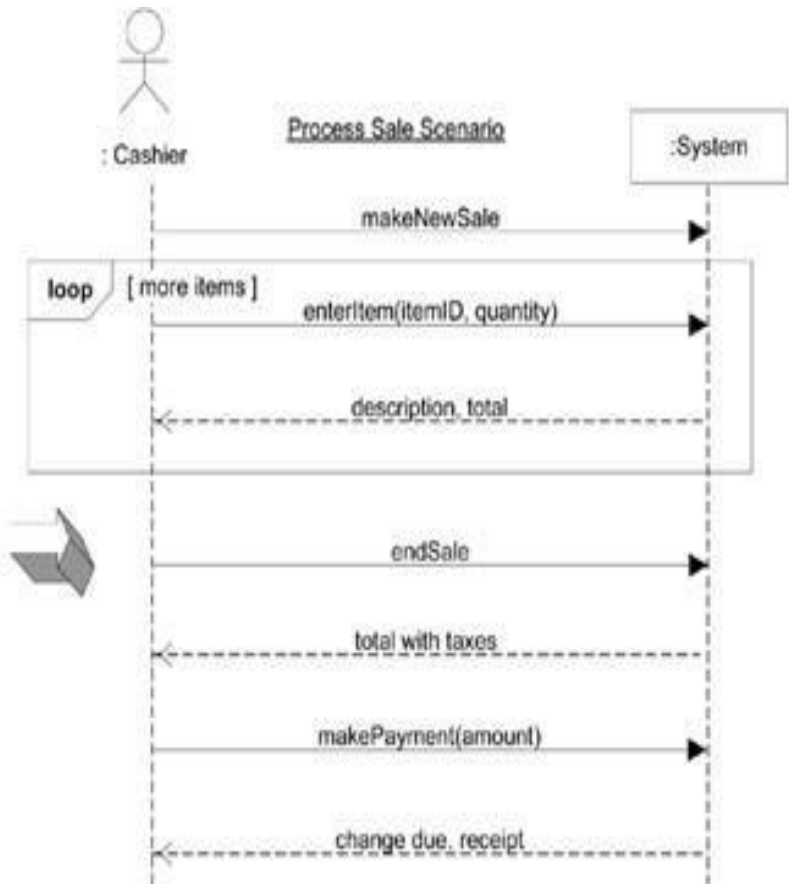
RELATIONSHIP BETWEEN SSDS AND USE CASES

An SSD shows system events for one scenario of a use case, therefore it is generated from inspection of a use case (see Figure below).

SSDs are derived from use cases; they show one scenario.

Simple cash-only Process Sale scenario:

1. Customer arrives at a POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total.
Cashier repeats steps 3-4 until indicates done.
5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.



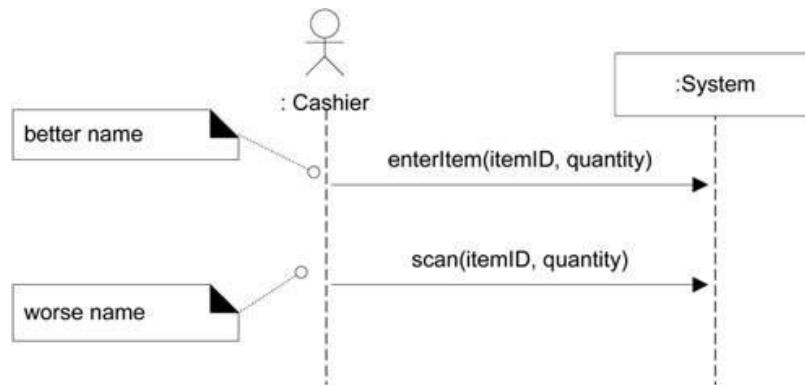
How to Name System Events and Operations?

Which is better, scan(itemID) or enterItem(itemID)?

System events should be expressed at the abstract level of intention rather than in terms of the physical input device.

Thus "enterItem" is better than "scan" (that is, laser scan) because it captures the intent of the operation while remaining abstract and noncommittal with respect to design choices about what interface is used to capture the system event.

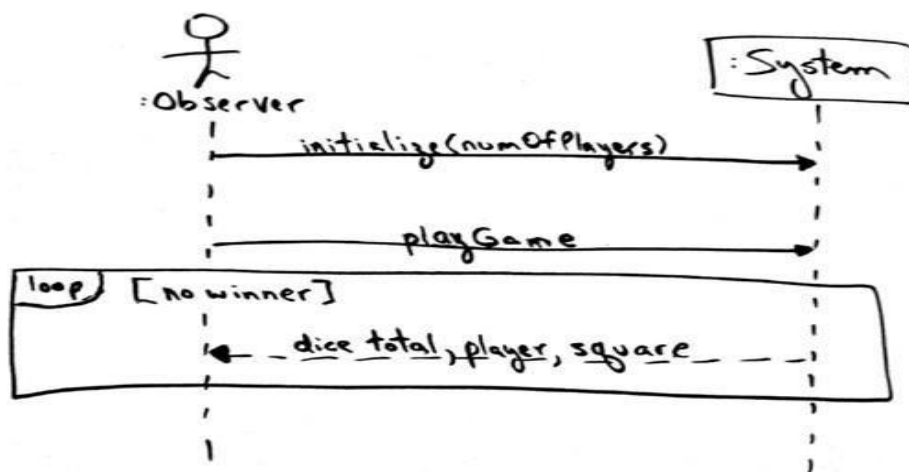
Choose event and operation names at an abstract level.



Example: Monopoly SSD

The Play Monopoly Game use case is simple, as is the main scenario. The observing person initializes with the number of players, and then requests the simulation of play, watching a trace of the output until there is a winner.

SSD for a Play Monopoly Game scenario.



Process:

Draw SSDs only for the scenarios chosen for the next iteration. Don't create SSDs for all scenarios, unless you are using an estimation technique that requires identification of all system operations.

WHEN TO USE CLASS DIAGRAMS

Class diagram is a static diagram and it is used to model the static view of a system. The static view describes the vocabulary of the system.

Class diagram is also considered as the foundation for component and deployment diagrams. Class diagrams are not only used to visualize the static view of the system

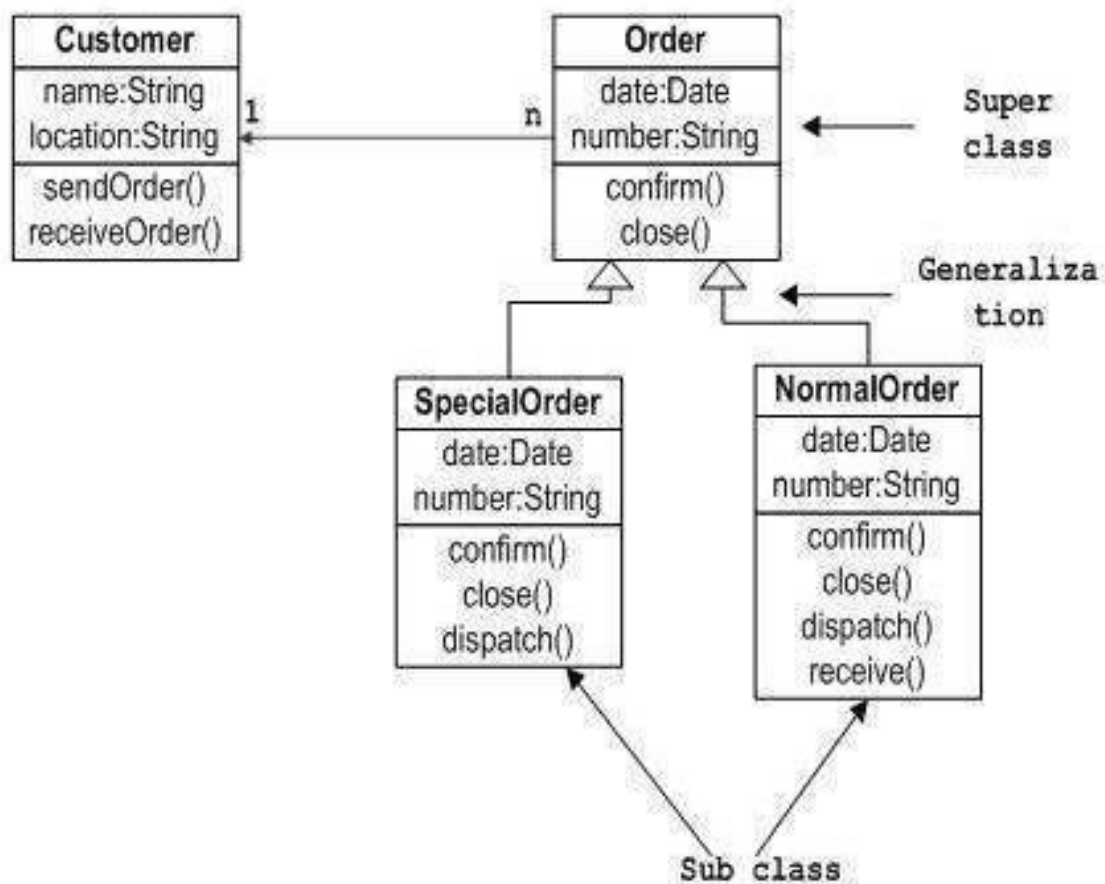
but they are also used to construct the executable code for forward and reverse engineering of any system.

Class diagram clearly shows the mapping with object-oriented languages such as Java, C++, etc. From practical experience, class diagram is generally used for construction purpose.

Class Diagrams are used for –

- Describing the static view of the system.
- Showing the collaboration among the elements of the static view.
- Describing the functionalities performed by the system.
- Construction of software applications using object oriented languages.
- Forward and reverse engineering.

Ex: Order Processing System



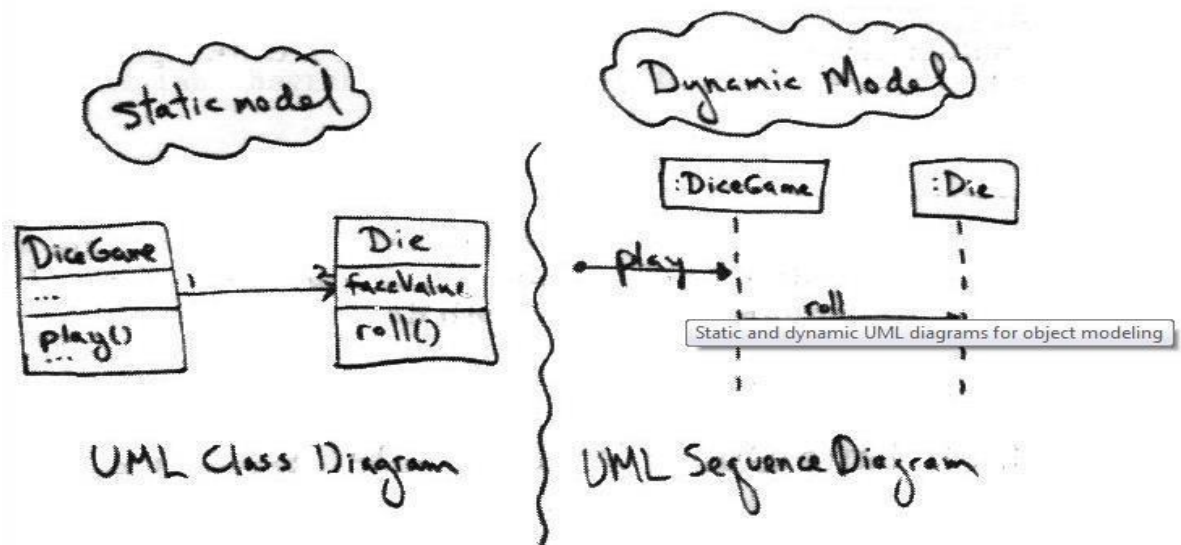
UNIT III - DYNAMIC AND IMPLEMENTATION UML DIAGRAMS

Dynamic Diagrams – UML interaction diagrams – System sequence diagram – Collaboration diagram – When to use Communication Diagrams – State machine diagram and Modeling –When to use State Diagrams – Activity diagram – When to use activity diagrams
Implementation Diagrams – UML package diagram – When to use package diagrams – Component and Deployment Diagrams – When to use Component and Deployment diagrams

DYNAMIC DIAGRAMS

There are two kinds of object models: dynamic and static. **Dynamic models**, such as UML interaction diagrams (sequence diagrams or communication diagrams), State chart diagram, Activity diagram, help design the logic, the behavior of the code or the method bodies. They tend to be the more interesting, difficult, important diagrams to create. **Static models**, such as UML class diagrams, help design the definition of packages, class names, attributes, and method signatures (but not method bodies).

Static and dynamic UML diagrams for object modeling



The main behavior or dynamic diagrams in UML are

- Interaction diagrams are:
 - Sequence diagrams
 - Collaboration diagrams
- State chart diagrams
- Activity diagrams

UML INTERACTION DIAGRAMS

The UML includes interaction diagrams to illustrate how objects interact via messages. They are used for dynamic object modeling. There are two common types:

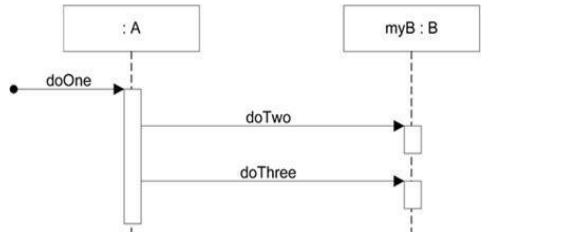
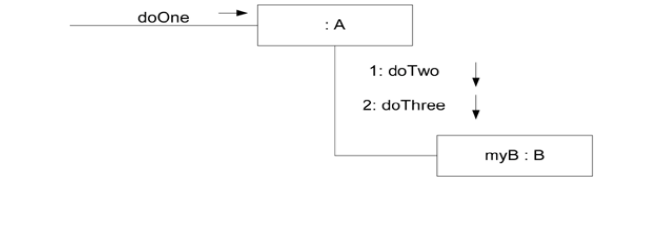
- Sequence Diagram
- Communication Diagram

Ex: Sequence and Communication Diagrams

```
public class A
{private B myB = new B();
```

```
public void doOne()
{ myB.doTwo();
  myB.doThree();
}}
```

Class A has a method named doOne and an attribute of type B. Also, that class B has methods named doTwo and doThree.

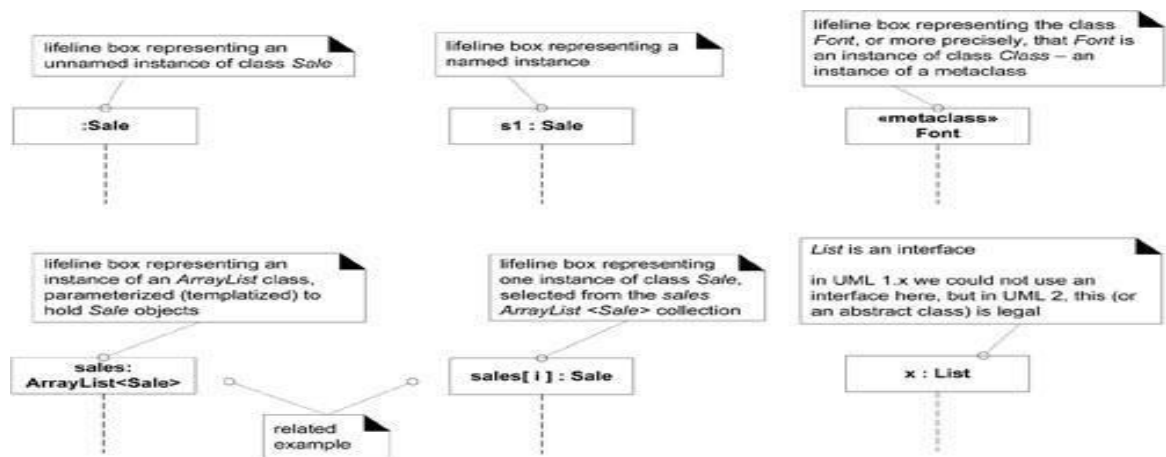
Sequence Diagram	Communication Diagram
Sequence diagrams illustrate interactions in a kind of fence format, in which each new object is added to the right	illustrate object interactions in a graph or network format, in which objects can be placed anywhere on the diagram
	

Strengths and Weaknesses of Sequence vs. Communication Diagrams

Type	Strengths	Weaknesses
sequence	clearly shows sequence or time ordering of messages large set of detailed notation options	forced to extend to the right when adding new objects; consumes horizontal space fewer notation options
communication	space economical flexibility to add new objects in two dimensions	more difficult to see sequence of messages

Common UML Interaction Diagram Notation

Lifeline boxes to show participants in interactions.



Basic Message Expression Syntax

The UML has a standard syntax for these message expressions:

return = message(parameter : parameterType) : returnType

Parentheses are usually excluded if there are no parameters, though still legal.

For example:

initialize(code)

initialize

d = getProductDescription(id)

d = getProductDescription(id:ItemID)

d = getProductDescription(id:ItemID) : ProductDescription

Singleton Objects

There is only one instance of a class instantiated never two. it is a "singleton" instance. In a UML interaction diagram (sequence or communication), such an object is marked with a '1' in the upper right corner of the lifeline box.

Singletons in interaction diagrams.



Basic Sequence Diagram Notation

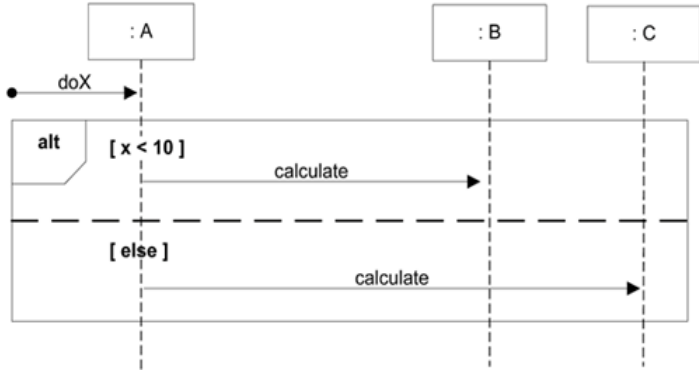
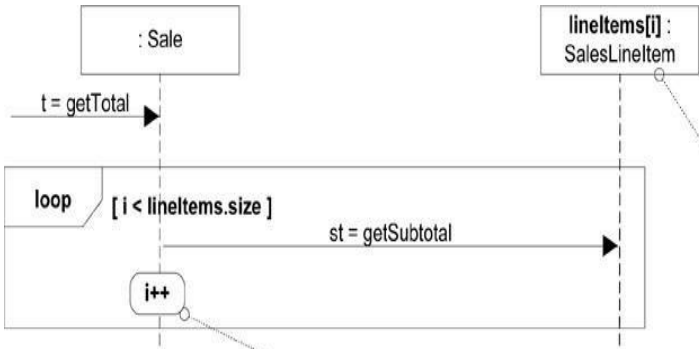
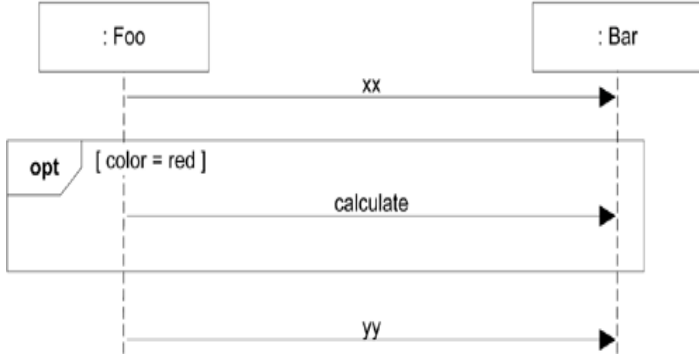
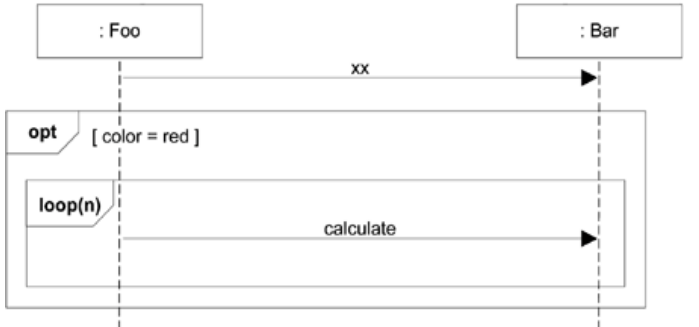
Name	Symbol	Description
Lifeline Boxes and Lifelines , Messages	<p>a found message whose sender will not be specified</p> <p>execution specification bar indicates focus of control</p> <p>typical synchronous message shown with a filled-arrow line</p>	<p>All UML examples show the lifeline as dashed (because of UML 1 influence), in fact the UML 2 specification says it may be solid or dashed.</p> <p>Found message the sender will not be specified, is not known, or that the message is coming from a random source.</p>

Reply or Return Msgs	<pre> sequenceDiagram participant Register as :Register participant Sale as :Sale Register->>Register: doX Register->>Sale: d1 = getDate Sale-->>Register: aDate </pre>	<p>There are two ways to show the return result from a message:</p> <ol style="list-style-type: none"> 1. Using the message syntax <code>returnVar = message(parameter)</code>. 2. Using a reply (or return) message line at the end of an activation bar.
Messages to "self" or "this"	<pre> sequenceDiagram participant Register as :Register Register->>Register: doX activate Register Register->>Register: clear deactivate Register </pre>	<p>message being sent from an object to itself by using a nested activation bar</p>
Creation of Instances and Object Destruction	<pre> sequenceDiagram participant Sale as :Sale participant Payment as :Payment Sale-->>Payment: create(cashTendered) activate Payment Sale->>Payment: «destroy» destroy Payment </pre>	<p>It is desirable to show explicit destruction of an object. The UML lifeline notation provides a way to express this destruction</p>

Frames

To support conditional and looping constructs (among many other things), the UML uses frames. Frames are regions or fragments of the diagrams; they have an operator or label (such as loop) and a guard (conditional clause).

Frame Operator	Meaning
alt	Alternative fragment for mutual exclusion conditional logic expressed in the guards.
loop	Loop fragment while guard is true. Can also write loop(n) to indicate looping n times. There is discussion that the specification will be enhanced to define a FOR loop, such as loop(i, 1, 10)
opt	Optional fragment that executes if guard is true.
par	Parallel fragments that execute in parallel.
region	Critical region within which only one thread can run.

Alt Frame	 <pre> sequenceDiagram participant A as : A participant B as : B participant C as : C A->>A: doX alt [x < 10] A->>B: calculate else [else] A->>C: calculate end </pre>	<p>Alternative fragment for mutual exclusion conditional logic expressed in the guards.</p>
Loop Frame	 <pre> sequenceDiagram participant Sale as : Sale participant LinItem as :lineltems[i] : SalesLinItem Sale->>Sale: t = getTotal loop [i < lineltems.size] Sale->>LinItem: st = getSubtotal Note over Sale: i++ end </pre>	<p>Loop fragment while guard is true. Can also write loop(n) to indicate looping n times. the specification will be enhanced to define a FOR loop, such as loop(i, 1, 10)</p>
Opt Frame	 <pre> sequenceDiagram participant Foo as : Foo participant Bar as : Bar Foo->>Bar: xx opt [color = red] Foo->>Bar: calculate end Foo->>Bar: yy </pre>	<p>Optional fragment that executes if guard is true.</p>
Nesting of frames	 <pre> sequenceDiagram participant Foo as : Foo participant Bar as : Bar Foo->>Bar: xx opt [color = red] loop(n) Foo->>Bar: calculate end end </pre>	<p>Frames can be nested</p>

Relating Interaction Diagrams		<p>An interaction occurrence (also called an interaction use) is a reference to an interaction within another interaction. It is useful, for example, when you want to simplify a diagram and factor out a portion into another diagram, or there is a reusable interaction occurrence.</p>
to Invoke Static (or Class) Methods		<p>The classes Class and Type are metaclasses, which means their instances are themselves classes. A specific class, such as class Calendar, is itself an instance of class Class. Thus, class Calendar is an instance of a metaclass.</p>
Asynchronous and Synchronous Calls		<p>An asynchronous message call does not wait for a response. They are used in multi-threaded environments such as .NET and Java.</p> <p>The UML notation for asynchronous calls is a stick arrow message; regular synchronous (blocking) calls are shown with a filled arrow.</p>

Basic Communication Diagram Notation

Name	Symbol	Description
Links , Messages		<p>A link is a connection path between two objects; it indicates some form of navigation and visibility between the objects</p> <p>Each message between objects is represented with a message expression , direction of the message , message Number</p>
Messages to "self" or "this"		<p>A message can be sent from an object to itself.</p> <p>This is illustrated by a link to itself, with messages flowing along the link.</p>
Creation of Instances		<p>The message may be annotated with a UML stereotype, like so: «create». The create message may include parameters, indicating the passing of initial values.</p>
Message Number Sequencing		<p>The first message is not numbered. Thus, msg1 is unnumbered.</p> <p>The order and nesting of subsequent messages is shown with a legal numbering scheme in which nested messages have a number appended to them.</p>
Conditional Messages		<p>A conditional message by following a sequence number with a conditional clause in square brackets, similar to an iteration clause. The message is only sent if the clause evaluates to true.</p>
Mutually Exclusive Conditional Paths		<p>modify the sequence expressions with a conditional path letter. The first letter used is a by convention.</p> <p>either 1a or 1b could execute after msg1. Both are sequence number 1 since either could be the first internal message.</p>

Iteration or Looping		a simple * can be used
Messages to a Classes to Invoke Static (Class) Methods		Meta class stereotype is used to represent static method call
Asynchronous and Synchronous Calls		asynchronous calls are shown with a stick arrow; synchronous calls with a filled arrow.

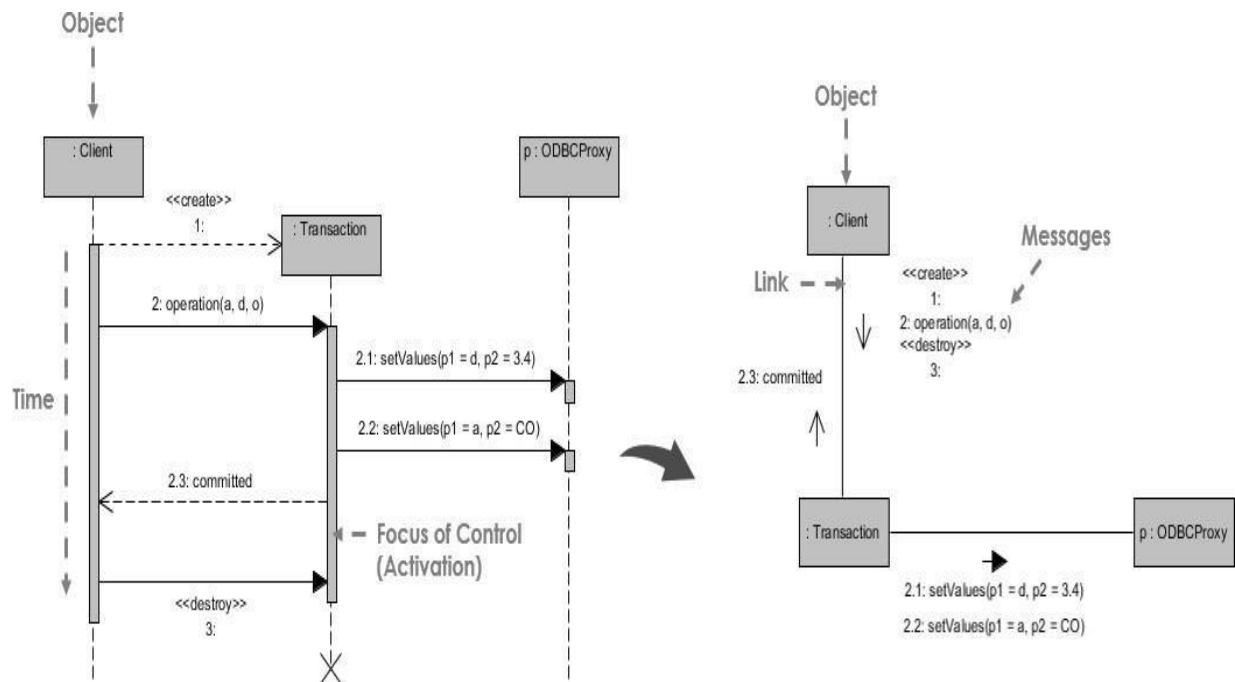
WHEN TO USE COMMUNICATION DIAGRAMS

The purpose of interaction diagrams is to visualize the interactive behavior of the system. Visualizing the interaction is a difficult task. Hence, the solution is to use different types of models to capture the different aspects of the interaction.

Sequence and collaboration diagrams are used to capture the dynamic nature but from a different angle. The purpose of interaction diagram is –

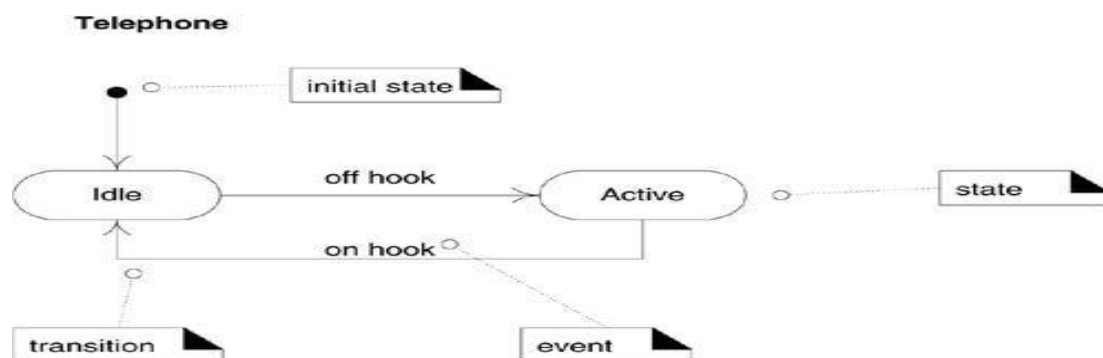
- To capture the dynamic behavior of a system.
- Model message passing between objects or roles that deliver the functionalities of use cases and operations
- To describe the structural organization of the objects.
- To describe the interaction among objects. Support the identification of objects (hence classes), and their attributes (parameters of message) and operations (messages) that participate in use cases

Sequence diagram vs Communication diagram Example



UML STATE MACHINE DIAGRAMS AND MODELLING

A UML state machine diagram illustrates the interesting events and states of an object, and the behavior of an object in reaction to an event. Transitions are shown as arrows, labeled with their event. States are shown in rounded rectangles. It is common to include an initial pseudo-state, which automatically transitions to another state when the instance is created.



State machine diagram for a telephone

A state machine diagram shows the lifecycle of an object: what events it experiences, its transitions, and the states it is in between these events. Therefore, we can create a state machine diagram that describes the lifecycle of an object at arbitrarily simple or complex levels of detail, depending on our needs.

Definitions: Events, States, and Transitions

An **event** is a significant or noteworthy occurrence. For example:

- A telephone receiver is taken off the hook.

A **state** is the condition of an object at a moment in time the time between events. For example:

- A telephone is in the state of being "idle" after the receiver is placed on the hook and until it is taken off the hook.

A **transition** is a relationship between two states that indicates that when an event occurs, the object moves from the prior state to the subsequent state. For example:

- When the event "off hook" occurs, transition the telephone from the "idle" to "active" state.

Guidelines : To Apply State Machine Diagrams:

Object can be classified into

- 1) **State-Independent Object** - If an object always responds the same way to an event, then it is considered state-independent (or modeless) with respect to that event. The object is state-independent with respect to that message.
- 2) **State-Dependent Objects** - State-dependent objects react differently to events depending on their state or mode.

Guideline : Consider state machines for state-dependent objects with complex behavior, not for state-independent objects . For example, a telephone is very state-dependent. The phone's reaction to pushing a particular button (generating an event) depends on the current mode of the phoneoff hook, engaged, in a configuration subsystem, and so forth.

Modeling State-Dependent Objects : state machines are applied in two ways:

1. To model the behavior of a **complex reactive object** in response to events.
2. To model **legal sequences of operations** protocol or language specifications.
 - This approach may be considered a specialization of #1, if the "object" is a language, protocol, or process. A formal grammar for a context-free language is a kind of state machine.

1. Complex Reactive Objects

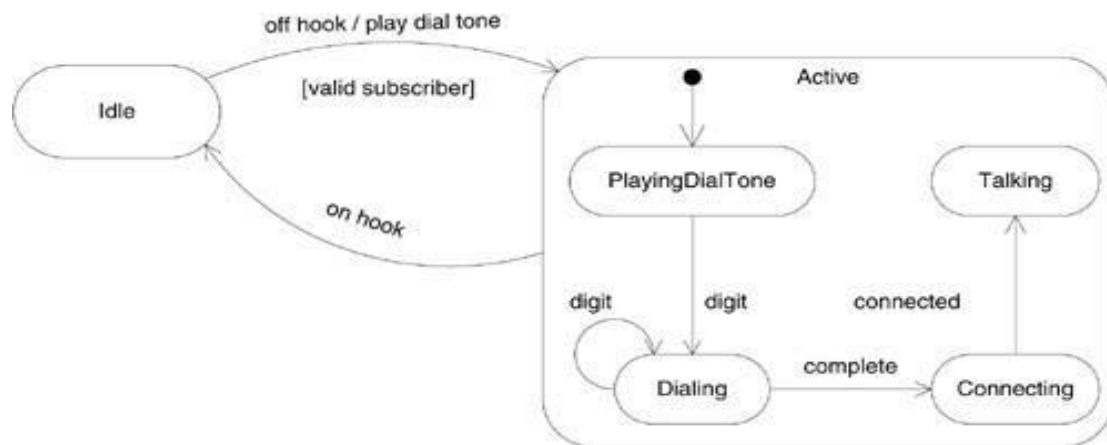
- a) **Physical Devices controlled by software**
 - Phone, car, microwave oven: They have complex and rich reactions to events, and the reaction depends upon their current mode.
- b) **Transactions and related Business Objects**
 - How does a business object (a sale, order, payment) react to an event? For example, what should happen to an Order if a cancel event occurs? And understanding all the events and states that a Package can go

through in the shipping business can help with design, validation, and process improvement.

- c) **Role Mutators** : These are objects that change their role.
 - o A Person changing roles from being a civilian to a veteran. Each role is represented by a state.

Example 1: Physical Devices / Nested States – Telephone Object

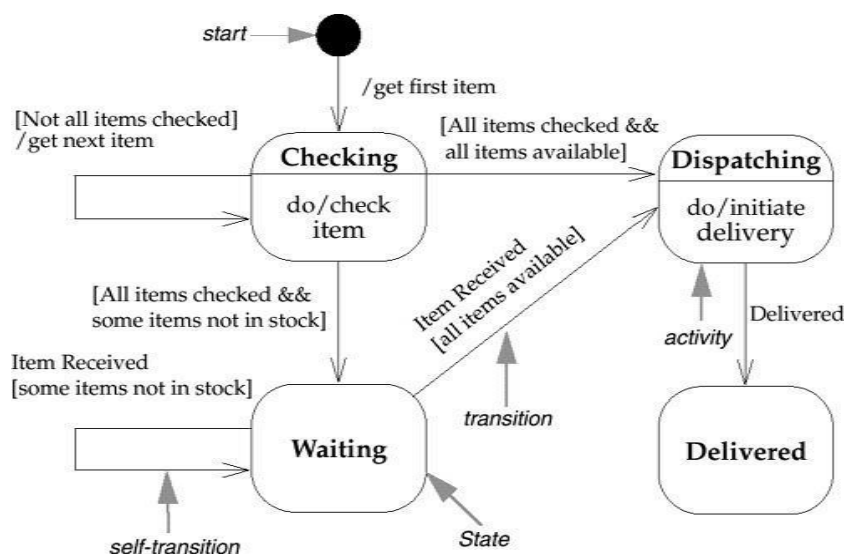
A state allows nesting to contain substates; a substate inherits the transitions of its superstate (the enclosing state). It may be graphically shown by nesting them in a superstate box..



For example, when a transition to the Active state occurs, creation and transition into the PlayingDialTone substate occurs. No matter what substate the object is in, if the on hook event related to the Active superstate occurs, a transition to the Idle state occurs.

Example 2: Transactions and related Business Objects

Order Processing System – Order Object



2) Protocols and Legal Sequences

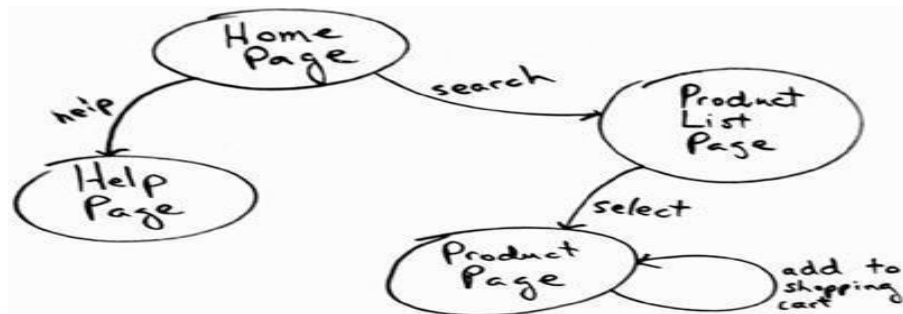
a) Communication Protocols

- TCP, and new protocols, can be easily and clearly understood with a state machine diagram. For example, a TCP "close" request should be ignored if the protocol handler is already in the "closed" state.

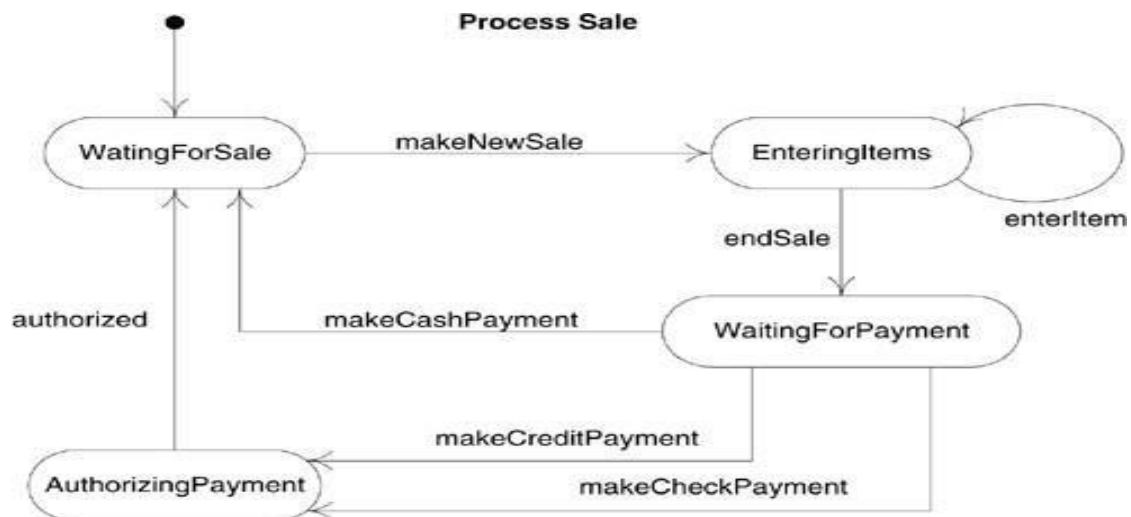
b) UI Page/Window Flow or Navigation

When doing UI modeling, it can be useful to understand the legal sequence between Web pages or windows;

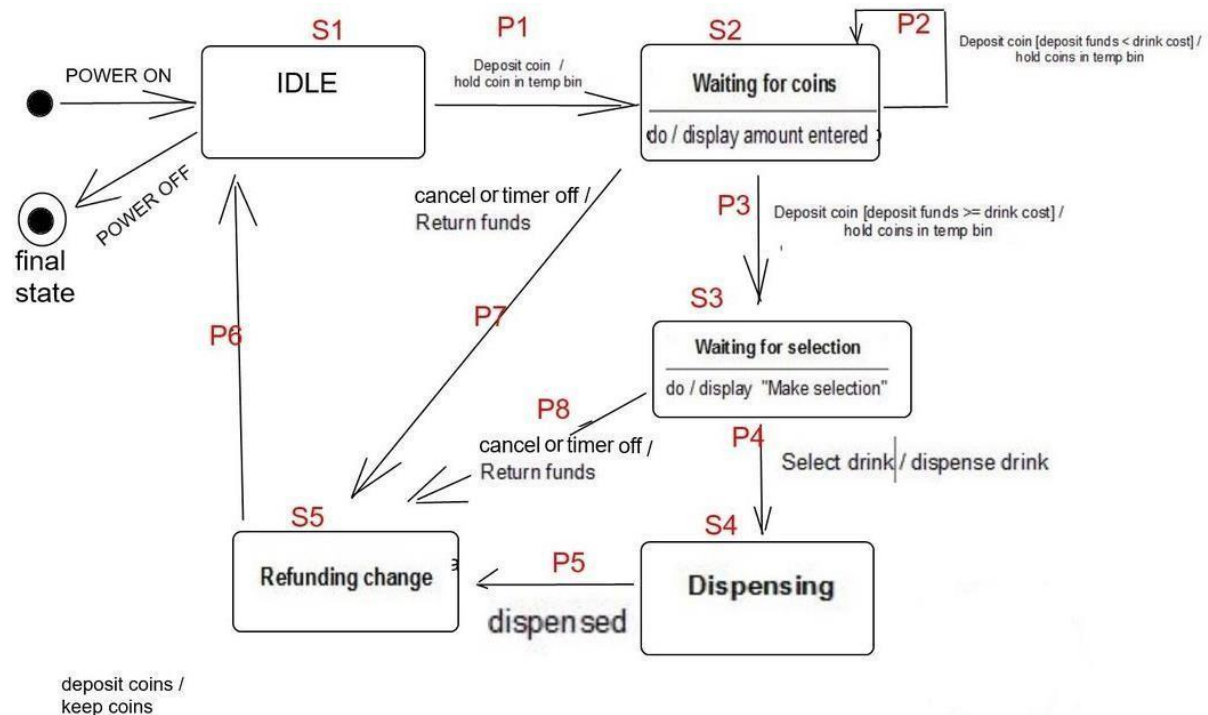
Applying a state machine to Web page navigation modeling.



- ### c) UI Flow Controllers or Sessions
- These are usually server-side objects representing an ongoing session or conversations with a client. For example, a Web application that remembers the state of the session with a Web client and controls the transitions to new Web pages, or the modified display of the current Web page, based upon the state of the session and the next operation that is received.
- ### d) Use Case System Operations
- Do you recall the system operations for Process Sale: makeNewSale, enterItem etc. These should arrive in a legal order; for example, endSale should only come after one or more enterItem operations.



STATE DIAGRAM : COFFEE VENDING MACHINE



WHEN TO USE STATE DIAGRAM

State chart diagram is one of the five UML diagrams used to model the dynamic nature of a system. They define different states of an object during its lifetime and these states are changed by events. State chart diagrams are useful to model the reactive systems. Reactive systems can be defined as a system that responds to external or internal events.

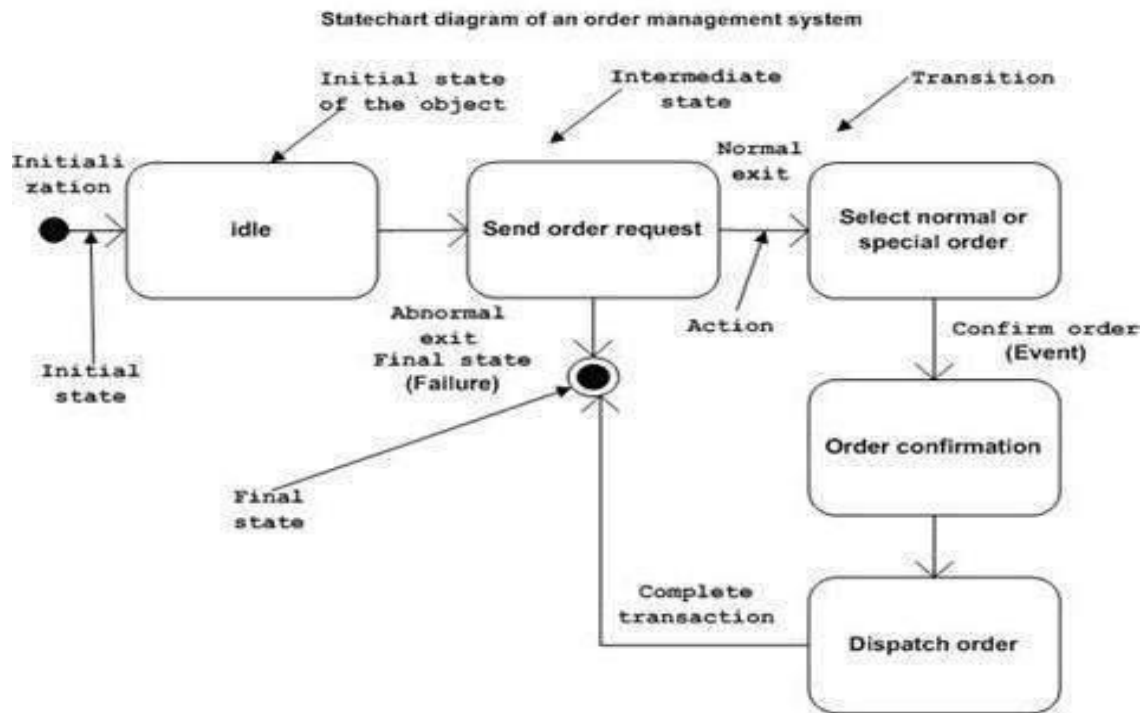
State chart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. The most important purpose of State chart diagram is to model lifetime of an object from creation to termination.

State chart diagrams are also used for forward and reverse engineering of a system. However, the main purpose is to model the reactive system.

The main usage can be described as –

- To model the object states of a system.
- To model the reactive system. Reactive system consists of reactive objects.
- To identify the events responsible for state changes.
- Forward and reverse engineering.

Example:



UML ACTIVITY DIAGRAMS





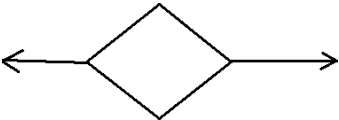
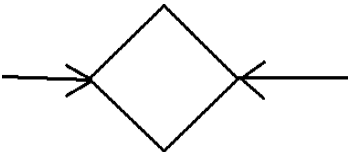
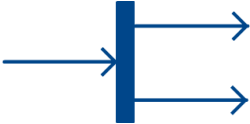
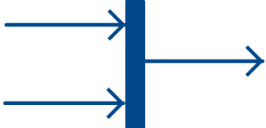

A UML activity diagram shows sequential and parallel activities in a process. They are useful for modeling business processes, workflows, data flows, and complex algorithms.

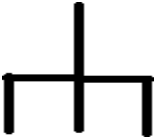


Purpose: To understand & Communicate the structure & dynamics of the organization in which a system is to be deployed.

Elements:

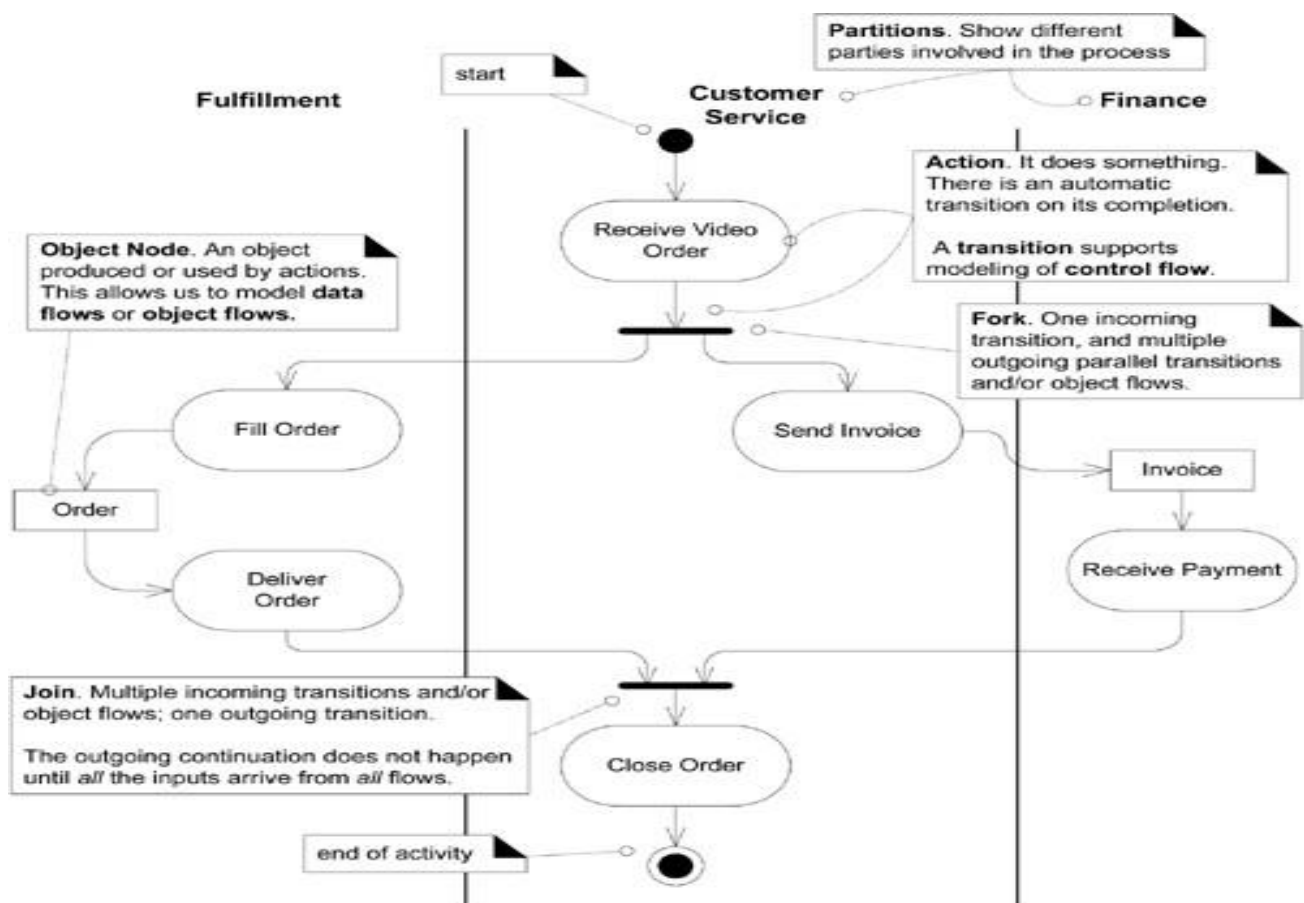
Start , end, activity ,action, object , fork ,join, decision , merge, time signal , rake , accept signal , swim lane

Symbol	Name	Use
	Start/ Initial Node	Used to represent the starting point or the initial state of an activity
	Activity / Action State	Used to represent the activities of the process

	Action	Used to represent the executable sub-areas of an activity
	Object	Used to represent the object
	Control Flow / Edge	Used to represent the flow of control from one action to the other
	Object Flow / Control Edge	Used to represent the path of objects moving through the activity
	Decision Node	Used to represent a conditional branch point with one input and multiple outputs
	Merge Node	Used to represent the merging of flows. It has several inputs, but one output.
	Fork	Used to represent a flow that may branch into two or more parallel flows
	Merge	Used to represent a flow that may branch into two or more parallel flows
	Signal Receipt	Used to represent that the signal is received

	rake	Further can be expanded into sub activity diagram
	swim lane	Divides the diagram into vertical zones
	Time Signal	Timing condition can be specified

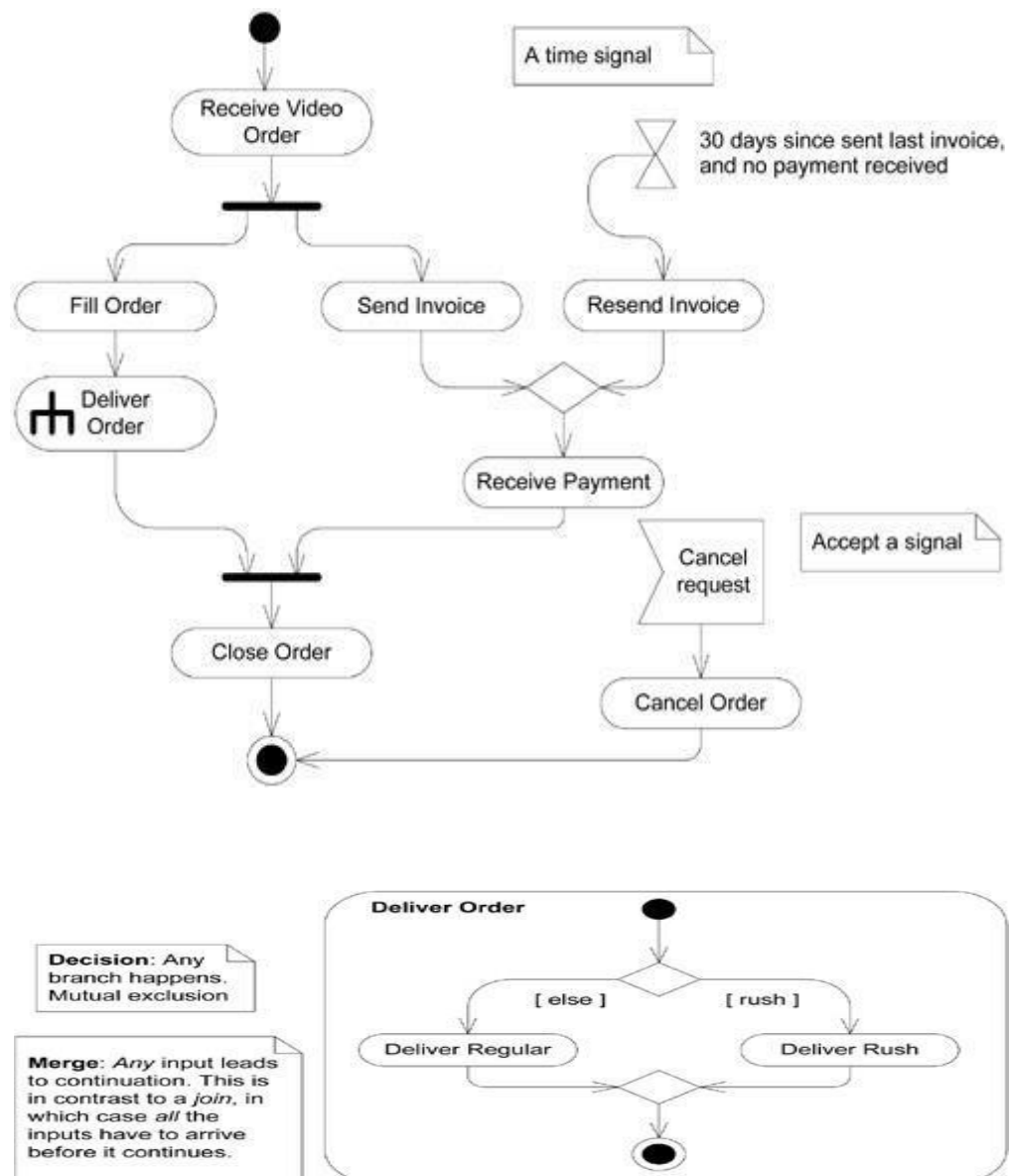
Example: Showing usage of symbols



Example

This diagram shows a sequence of actions, some of which may be parallel. Two points to remember :

- once an action is finished, there is an automatic outgoing transition
- the diagram can show both control flow and data flow



Guideline to Apply Activity Diagrams

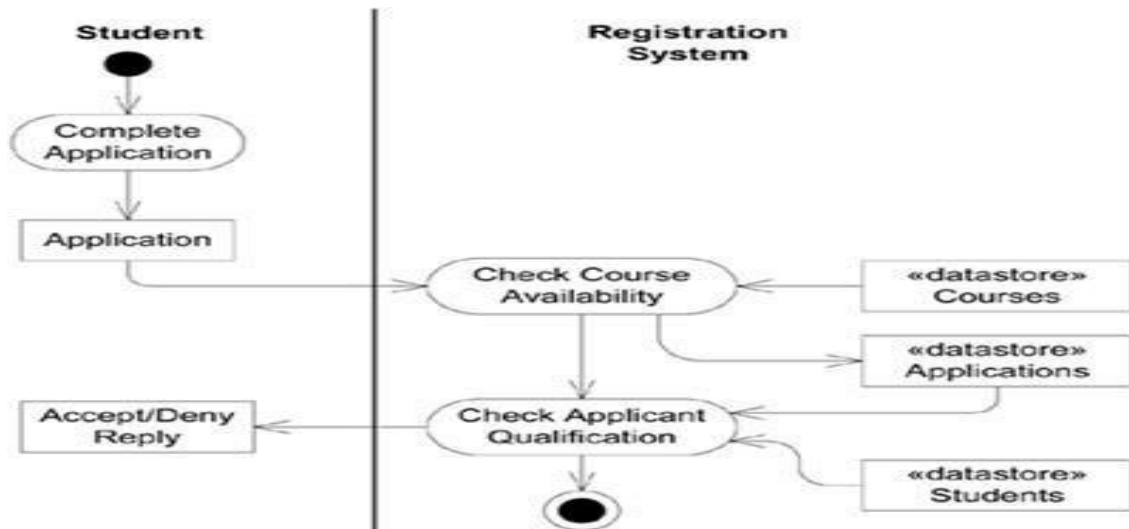
1. Business Process Modeling

Client uses activity diagrams to understand their current complex business processes by visualizing them. The partitions are useful to see the multiple parties and parallel actions involved in the shipping process, and the object nodes illustrate what's moving around.

Ex: Borrow books return books usecase of Library Information system

2. Data Flow Modeling

data flow diagrams (DFD) became a popular way to visualize the major steps and data involved in software system processes. This is not the same as business process modeling; rather, DFDs were usually used to show data flows in a computer system, although they could in theory be applied to business process modeling.



3. Concurrent Programming and Parallel Algorithm Modeling

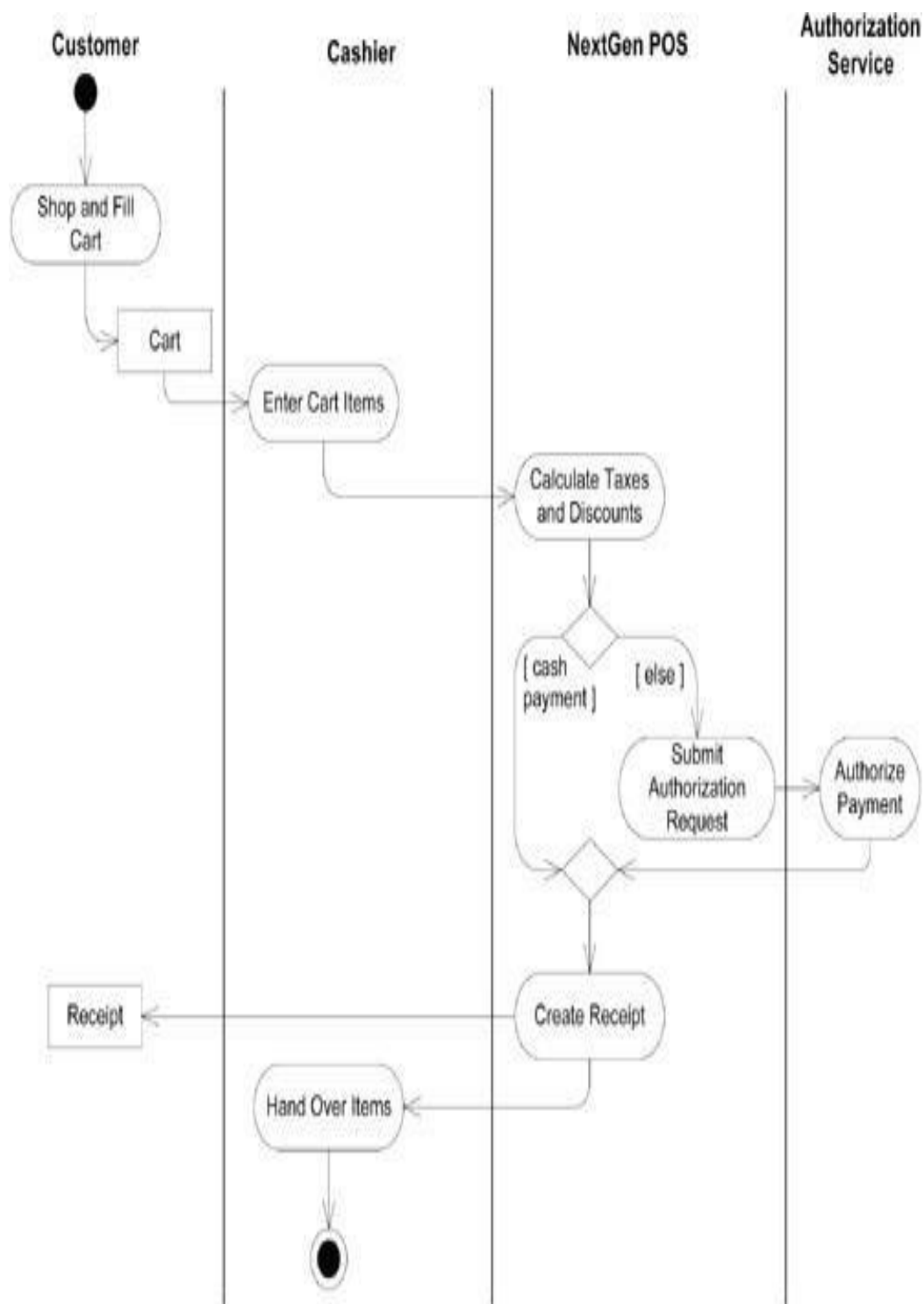
Parallel algorithms in concurrent programming problems involve multiple partitions, and fork and join behavior. The UML activity diagram partitions can be used to represent different operating system threads or processes. The object nodes can be used to model the shared objects and data. Forking can be used to model the creation and parallel execution of multiple threads or processes, one per partition.

4. Guidelines

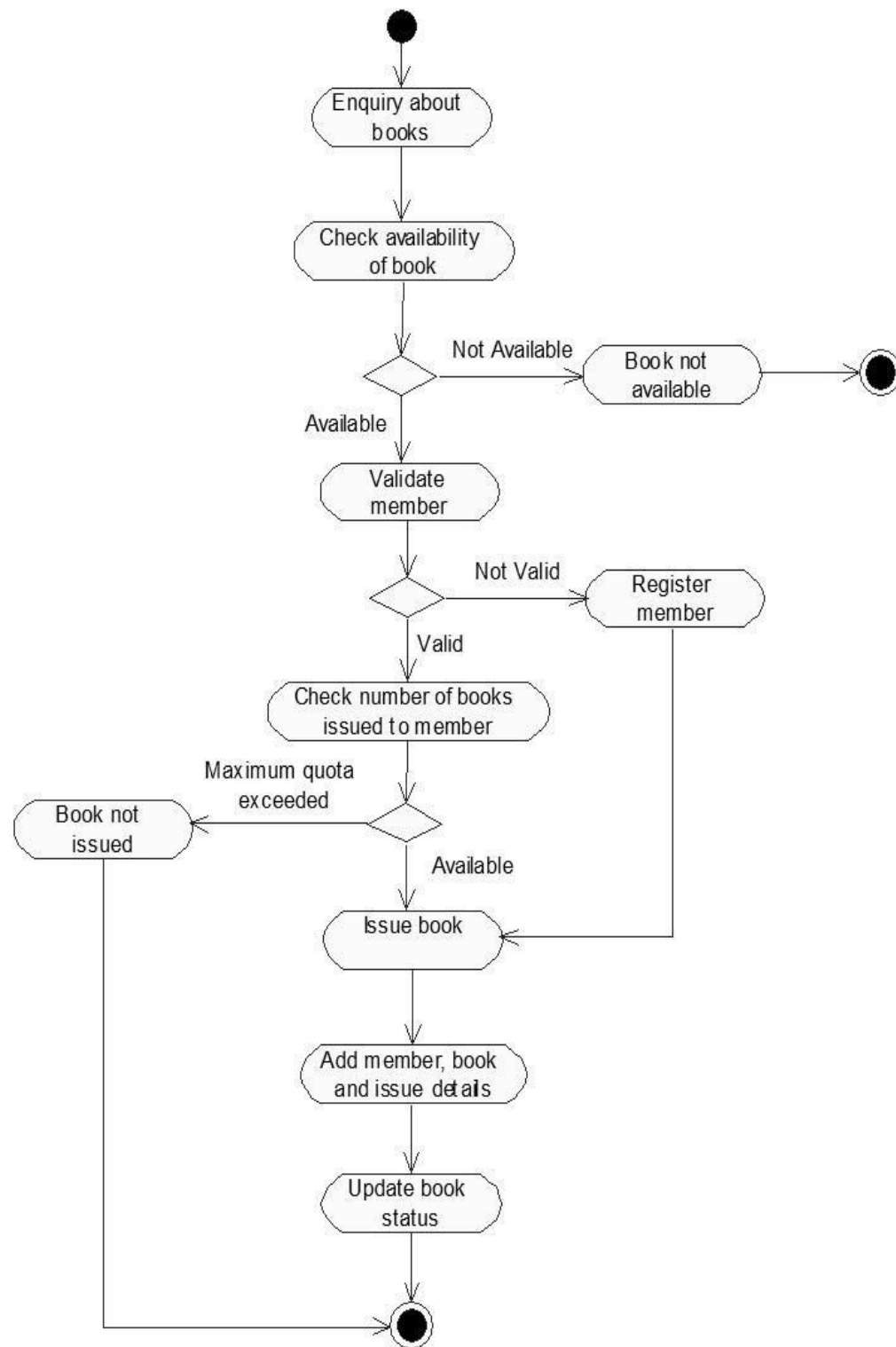
- If modeling a business process, take advantage of the "rake" notation and sub-activity diagrams. On the first overview "level 0" diagram, keep all the actions at a very high level of abstraction, so that the diagram is short and sweet. Expand the details in sub-diagrams at the "level 1" level, and perhaps even more at the "level 2" level, and so forth.

Example: NextGen Activity Diagram

The partial model in Figure illustrates applying the UML to the Process Sale use case process.



Activity Diagram Ex1: Borrow Books (Library Information System)



WHEN TO USE ACTIVITY DIAGRAMS

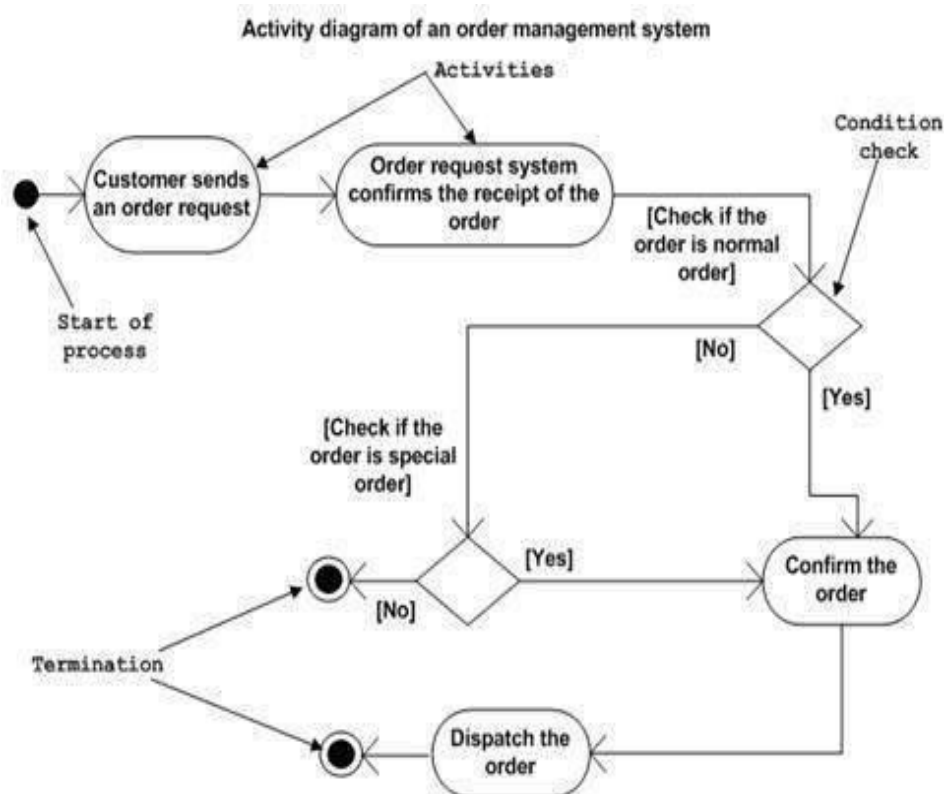
Activity diagram is suitable for modeling the activity flow of the system. An application can have multiple systems. Activity diagram also captures these systems and describes the flow from one system to another. This specific usage is not available in other diagrams. These systems can be database, external queues, or any other system.

Activity diagram gives high level view of a system. This high level view is mainly for business users or any other person who is not a technical person. This diagram is used to model the activities of business requirements. The diagram has more impact on business understanding rather than on implementation details.

Activity diagram can be used for –

- Modeling work flow by using activities.
- Modeling business requirements.
- High level understanding of the system's functionalities.
- Investigating business requirements at a later stage.

Example



UML PACKAGE DIAGRAMS

UML package diagrams are often used to illustrate the logical architecture of a system-the layers, subsystems, packages . A layer can be modeled as a UML package; It is part of the Design Model and also be summarized as a view in the Software Architecture Document.

Logical Architecture is the large scale organization of the software classes into packages subsystem and layers It is called logical architecture because there's no decision about how these elements are deployed across different operating system processes or across physical computers in a network.

Layer is a coarse grained grouping of classes , packages, or subsystems that has cohesive responsibility for major aspect of the system .

There are 2 types of Layers.

- 1) Higher Layer (Contain more application specific services ex: UI layer)
- 2) Lower layer (Contain more generalized services ex: Technical Services layer)

Higher Layer calls upon services of lower layer , but vice versa is not .

Typically layers in the Object Oriented System has 7 standard layers. The important layers are

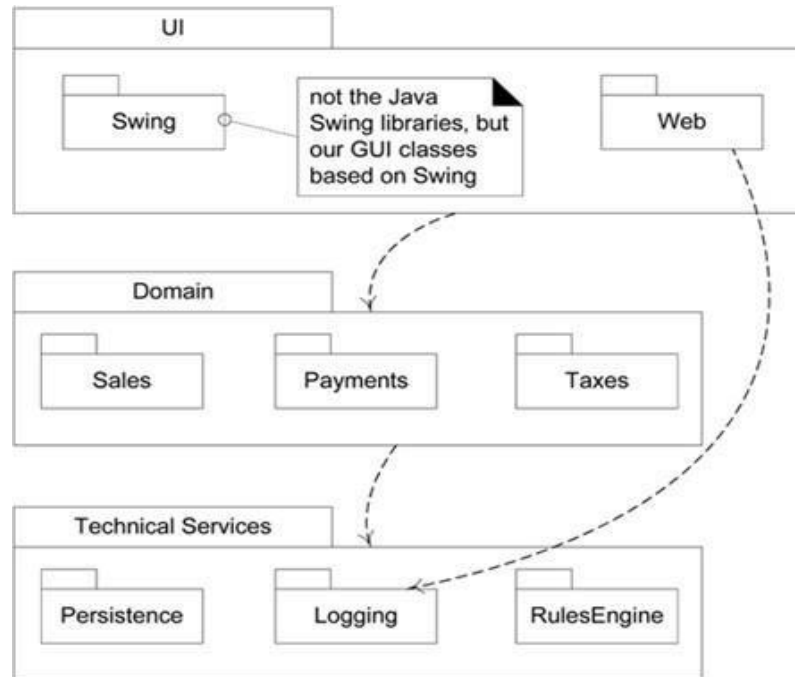
- **User Interface** – Has various I/O formats & forms.
- **Application Logic and Domain Objects** - software objects representing domain concepts (for example, a software class Sale) that fulfill application requirements, such as calculating a sale total.
- **Technical Services** general purpose objects and subsystems that provide supporting technical services, such as interfacing with a database or error logging.

Architecture Types

strict layered architecture ⑦ a layer only calls upon the services of the layer directly below it. This design is common in network protocol stacks, but not in information systems

Relaxed layered architecture ⑦ a higher layer calls upon several lower layers. For example, the UI layer may call upon its directly subordinate application logic layer, and also upon elements of a lower technical service layer, for logging and so forth.

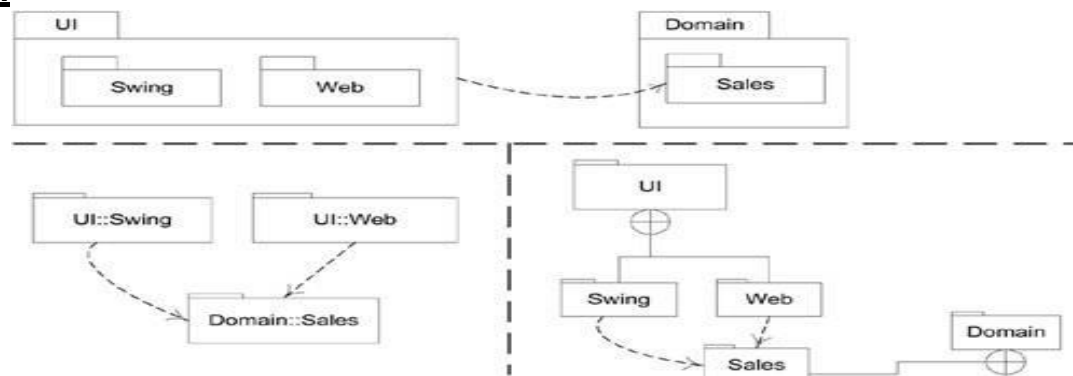
Layers shown with UML package diagram notation.



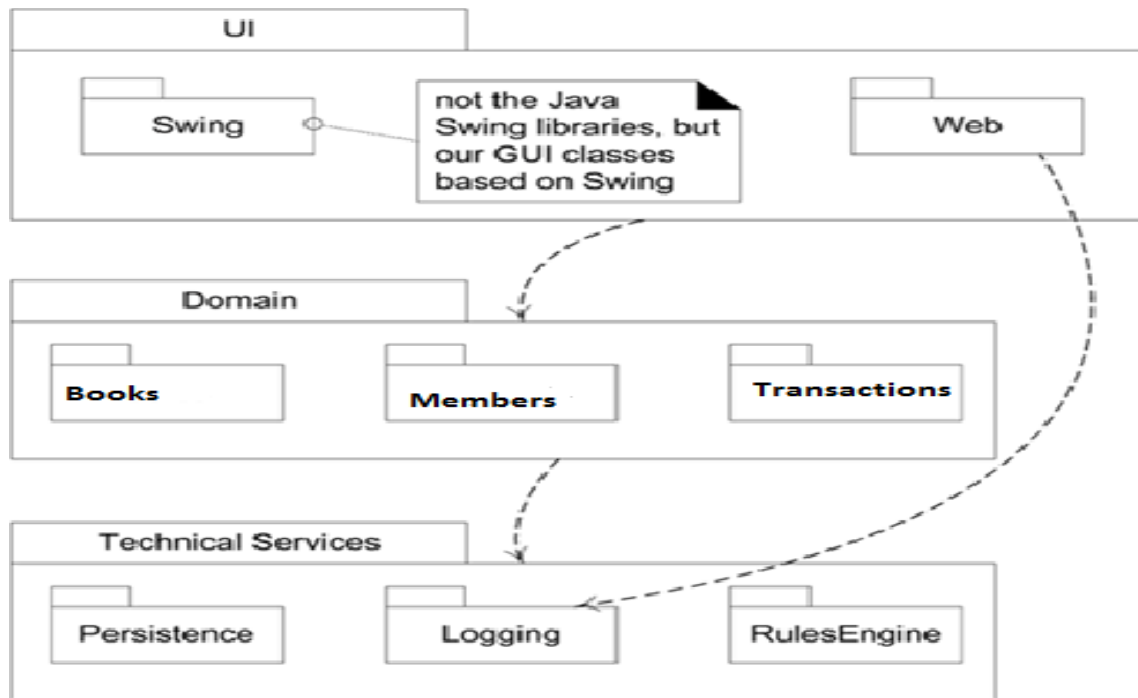
Elements

Name	Symbol	Description
Package		package can group anything: classes, other packages, use cases
Dependency		depended-on package
Fully qualified Name	java::util::Date	To represents a namespace (outer package named "java" with a nested package named "util" with a Date class)

Ex:



Package Diagram : Library Information System



WHEN TO USE PACKAGE DIAGRAMS

1. It is used in large scale systems to picture dependencies between major elements in the system.
2. Package diagrams represent a compile time grouping mechanism.

UML DEPLOYMENT AND COMPONENT DIAGRAMS

Deployment Diagrams : A deployment diagram shows the assignment of concrete software artifacts (such as executable files) to computational nodes (something with processing services). It shows the deployment of software elements to the physical architecture and the communication (usually on a network) between physical elements

Two Nodes of Deployment Diagram :

- 1) **Device Node :** This is a Physical Computing Resource representing a computer with memory or mobile.


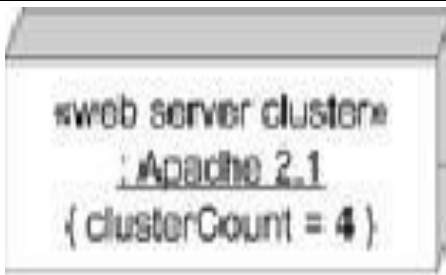

- 2) **Execution Environment Node :**

This is a software computing resource that runs within an outer node (such as a computer) and which itself provides a service to host and execute other executable software elements. **For example:**

- an operating system (OS) is software that hosts and executes programs
- a virtual machine (VM, such as the Java or .NET VM) hosts and executes programs

- a database engine (such as PostgreSQL) receives SQL program requests and executes them, and hosts/executes internal stored procedures (written in Java or a proprietary language)
- a Web browser hosts and executes JavaScript, Java applets, Flash, and other executable technologies
- a workflow engine
- a servlet container or EJB container

Elements:

Name	Symbol	Description
Device Node :		Physical Computing Resource
Execution Environment Node	 { OS = Linux } { JVM = sun Hot Spot 2.0}	a software computing resource
Communication path		Connection between nodes with protocol name
Artifact:		Name of the project file

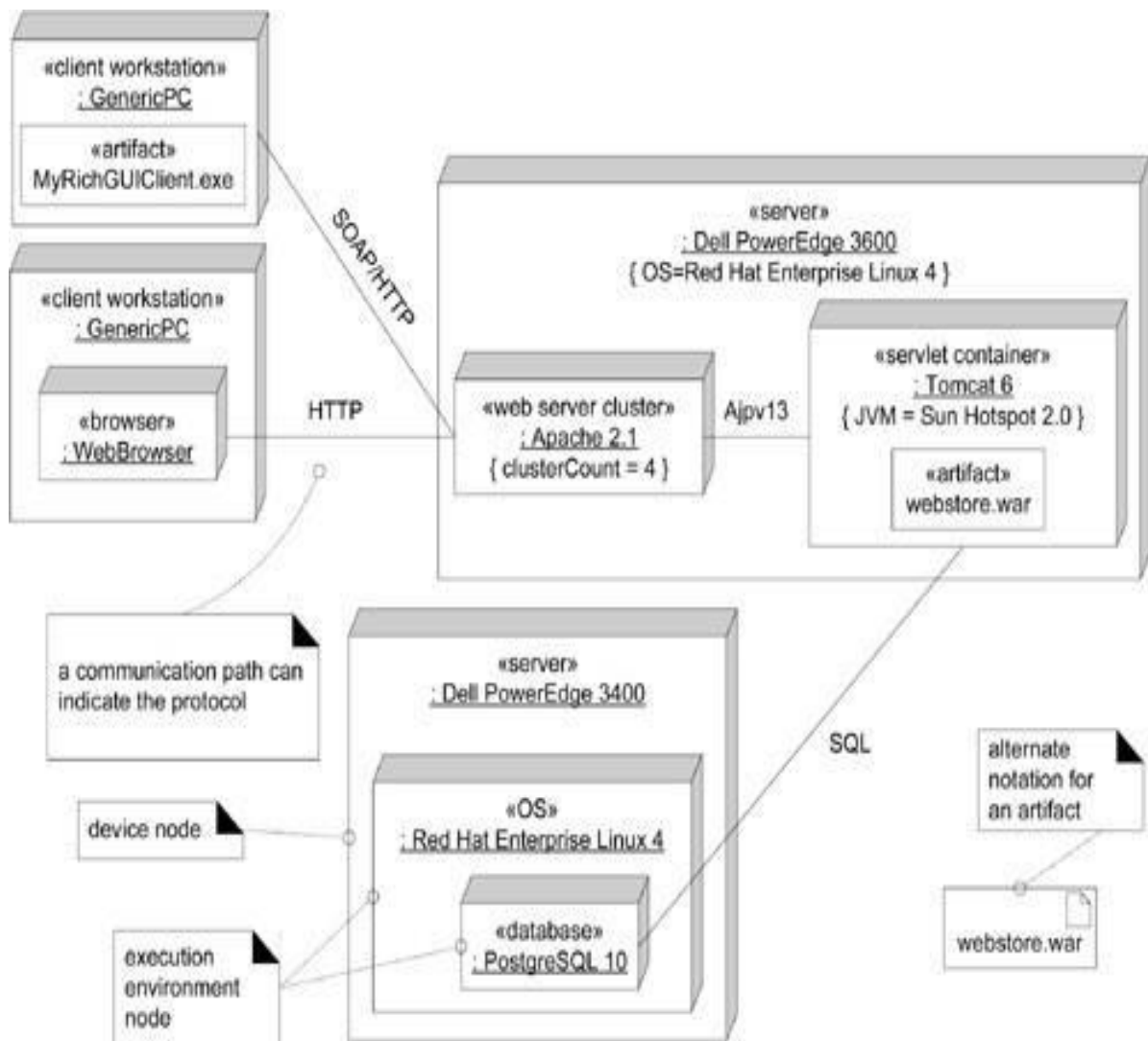
- As the UML specification suggests, many node types may show stereotypes, such as «server», «OS», «database», or «browser», but these are not official predefined UML stereotypes.
- Note that a device node or EEN may contain another EEN. For example, a virtual machine within an OS within a computer.
- A particular EEN can be implied, or not shown, or indicated informally with a UML property string; for example, { OS=Linux }.
- The normal connection between nodes is a communication path, which may be labeled with the protocol. These usually indicate the network connections.

- A node may contain and show an artifact a concrete physical element, usually a file. This includes executables such as JARs, assemblies, .exe files, and scripts. It also includes data files such as XML, HTML, and so forth.

Deployment Diagram Ex: Next Generation POS System

In the diagram ,there are 2 servers namely DellpowerEdge 3600 with RedHat Linux OS , Tomcat 6, Apache 2.1 are software computing resources , in tomcat server webstore.war file is loaded. In the other server DellpowerEdge 3400 with RedHat Linux OS, database PostgreSQL 10 are software computing resources.

There are 2 client nodes connected to server via HTTP & SOAP /HTTP protocol. In first client exe file is shown as artifact. In the other client , web base application is enabled by browser .



COMPONENT DIAGRAMS

A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. A component serves as a type, whose conformance is defined by these provided and required interfaces.

Features of UML component


- 1) It has interfaces
- 2) it is modular, self-contained and replaceable.

The second point implies that a component tends to have little or no dependency on other external elements . it is a relatively stand-alone module.

Example : A good analogy for software component modeling is a home entertainment system; we expect to be able to easily replace the DVD player or speakers. They are modular, self-contained, replaceable, and work via standard interfaces.

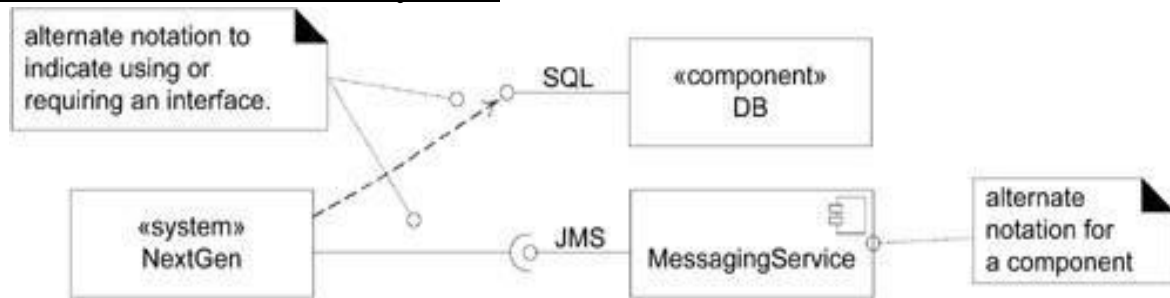
For example, at a large-grained level, a SQL database engine can be modeled as a component; any database that understands the same version of SQL and supports the same transaction semantics can be substituted. At a finer level, any solution that implements the standard Java Message Service API can be used or replaced in a system.

Elements:

Name	Symbol	Description
Component with Provided Interface		stand-alone module.
Dependency		System getting services from the component

Guideline : Component-based modeling is suitable for relatively large-scale elements, because it is difficult to think about or design for many small, fine-grained replaceable parts.

Ex: Next Generation POS system.



WHEN TO USE COMPONENT AND DEPLOYMENT DIAGRAMS

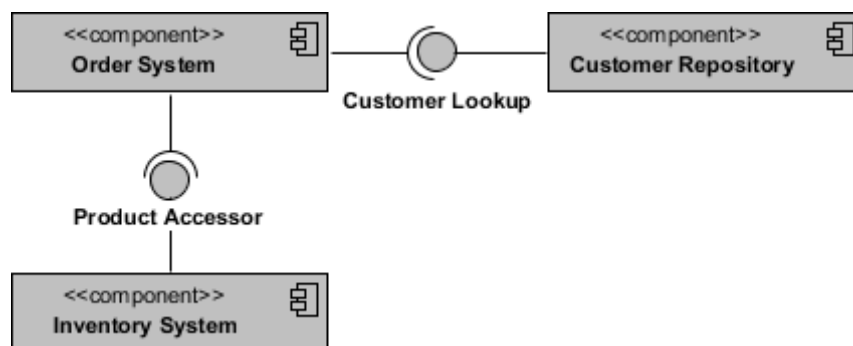
Component diagrams are used to visualize the static implementation view of a system. Component diagrams are special type of UML diagrams used for different purposes.

Component diagrams can be used to –

- Model the components of a system.
- Model the database schema.
- Model the executables of an application.
- Model the system's source code.

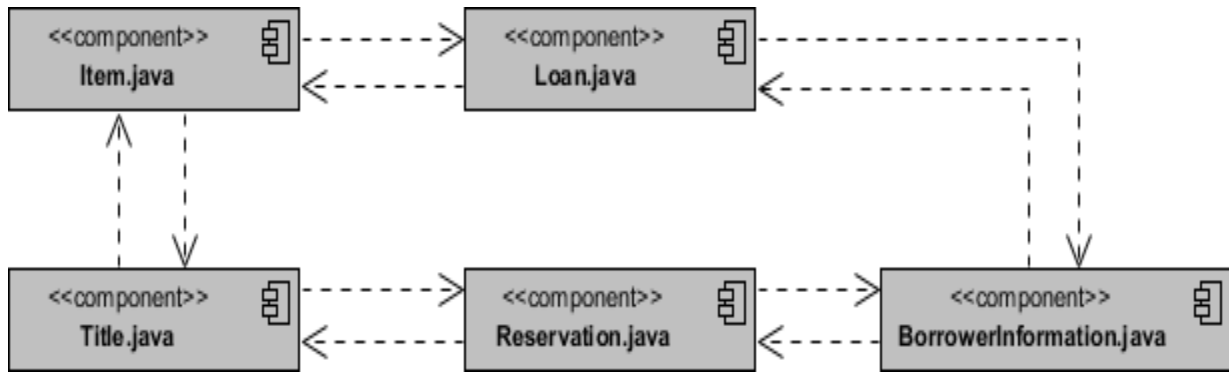
Model the components of a system

Ex: Order System



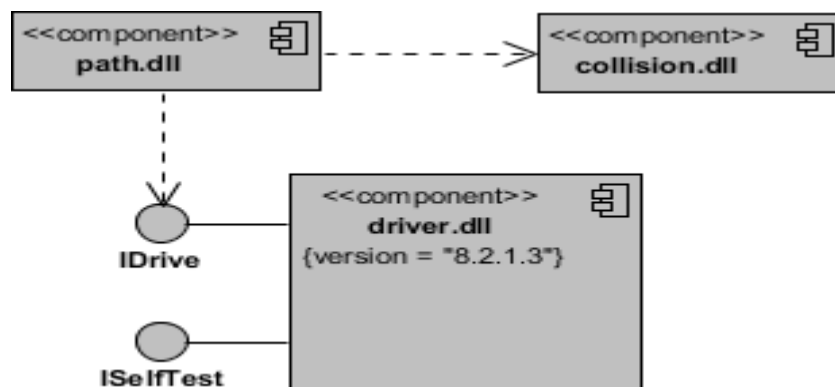
Modeling Source Code

- Either by forward or reverse engineering, identify the set of source code files of interest and model them as components stereotyped as files.
- For larger systems, use packages to show groups of source code files.
- Consider exposing a tagged value indicating such information as the version number of the source code file, its author, and the date it was last changed. Use tools to manage the value of this tag.
- Model the compilation dependencies among these files using dependencies. Again, use tools to help generate and manage these dependencies.



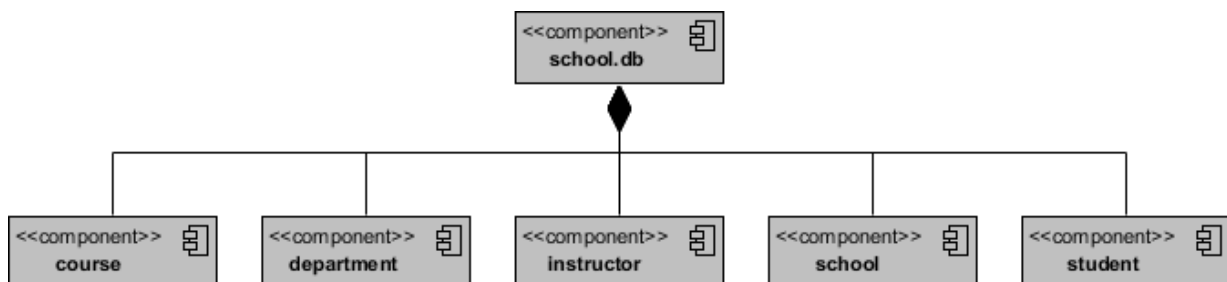
Modeling an Executable Release

- Identify the set of components to model.
- Consider the stereotype of each component in this set. find a small number of different kinds of components (such as executables, libraries, tables, files, and documents).
- For each component in this set, consider its relationship to its neighbors. It shows only dependencies among the comp



Modeling a Physical Database

- Identify the classes in the model that represent the logical database schema.
- Select a strategy for mapping these classes to tables.
- To visualize, specify, construct, and document your mapping, create a component diagram that contains components stereotyped as tables.
- Where possible, use tools to help you transform your logical design into a physical design.



UNIT IV - DESIGN PATTERNS

GRASP: Designing objects with responsibilities – Creator – Information expert – Low Coupling – High Cohesion – Controller Design Patterns – creational – factory method – structural – Bridge – Adapter – behavioural – Strategy – observer –Applying GoF design patterns – Mapping design to code

4.1 INTRODUCTION

PANIMALAR ENGINEERING COLLEGE

- One way to describe object design “After identifying your requirements and creating a domain model, then add methods to the software classes, and define the messaging between the objects to fulfill the requirements”
- Not really- Answer the following questions:
 - What methods belong to where?
 - How the objects should interact?
- GRASP as Methodical Approach to Learning Basic Object Design

UML versus Design Principles

- The UML is simply a standard visual modeling language, knowing its details doesn't teach you how to think in objects – that is the theme of this course
- The critical design tool for software development is a mind well educated in design principles. It is not the UML or any other technology

Object Design

After the requirements identification, add the methods to the classes and define the message between the objects. The designing of object starts with

- **Inputs**
- **Activities**
- **Outputs**

Inputs to object design

- Use case model
- Domain Model
- System Sequence Diagrams
- Operation Contracts

- Supplementary Specification

Activities of object design

- Dynamic and static modeling (draw both interaction and complementary class diagrams)
- Applying various OOD principles
 - GRASP- General Responsibility Assignment Software Patterns
 - GoF (Gang of Four) design patterns
 - Responsibility-Driven Design (RDD)

Outputs of object design

- Modeling for the difficult part of the design that we wished to explore before coding
- Specifically for object design,
 - UML Interaction diagrams
 - Class diagrams
 - Package diagrams
- UI sketches and prototypes
- Database models Report sketches and prototypes

4.2 RDD

- RDD is a general metaphor for thinking about OO software design.
- Thinking of software objects as having responsibilities an abstraction of what they do. Responsibility means a contract or obligation of a classifier.
- RDD is a general metaphor for thinking about object oriented design.

Responsibilities are related to the obligation of an object in terms of its behavior

- what an object should know?
- what an object should do?

Responsibilities is of two types : Doing , Knowing

1. Knowing responsibilities:

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

2. Doing responsibilities:

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

What's the Connection Between Responsibilities, GRASP, and UML Diagrams?

assigning responsibilities to objects while coding or while modeling. Within the UML, drawing interaction diagrams becomes the occasion for considering these responsibilities.

Responsibilities are implemented using methods

What are Patterns?

A [pattern](#) is a named description of a problem and solution that can be applied to new contexts; ideally, a pattern advises us on how to apply its solution in varying circumstances and considers the forces and trade-offs.

Many patterns, given a specific category of problem, guide the assignment of responsibilities to objects.

Example : The format is

Pattern Name : Information Expert

Problem : What is a basic principle by which to assign responsibilities to objects?

Solution : Assign a responsibility to the class that has the information needed to fulfill it.

4.3 APPLYING GRASP TO OBJECT DESIGN

GRASP stands for “**General Responsibility Assignment Software Patterns**”

- It is a Learning Aid for OO Design with Responsibilities. This approach to understanding and using design principles is based on patterns of assigning responsibilities.
- We can apply the GRASP principles while drawing UML interaction diagrams, and also while coding where we deciding on responsibly assignments.
- **GRASP** defines **nine basic OO design principles or basic building blocks** in design.
- They are
 1. Information Expert
 2. Creator
 3. Controller
 4. Low Coupling
 5. High Cohesion
 6. Polymorphism
 7. Pure Fabrication
 8. Indirection
 9. Protected Variations.

All these patterns answer some software problem, and in almost every case these problems are common to almost every software development project.

PATTERN/ PRINCIPLE	DESCRIPTION
Information Expert	A general principle of object design and responsibility assignment? Assign a responsibility to the information expert – the class that has the information necessary to fulfill the responsibility.
Creator	Who creates? (Note that Factory is a common alternate solution.) Assign class B the responsibility to create an instance of class A if one of these is true: <ol style="list-style-type: none"> 1. B contains A 2. B aggregates A 3. B has the initializing data for A 4. B records A 5. B closely uses A
Controller	What first object beyond the UI layer receives and coordinates (“controls”) a system operation? Assign the responsibility to an object representing one of these choices: <ol style="list-style-type: none"> 1. Represents the overall “system,” a “root object,” a device that the software is running within, or a major subsystem (these are all variations of a façade controller). 2. Represents a use case scenario within which the system operation occurs (a use-case or session controller)
Low coupling (evaluative)	How to reduce the impact of change? Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.
High Cohesion (evaluative)	How to keep objects focused, understandable, and manageable, and as a side-effect, support low Coupling? Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives.
Polymorphism	Who is responsible when behavior varies by type? When related alternatives or behaviors vary by type (class), assign responsibility for the behavior – using polymorphic operations – to the types for which the behavior varies.
Pure Fabrication	Who is responsible when you are desperate, and do not want to violate high cohesion and low coupling? Assign a highly cohesive set of responsibilities to an artificial or convenience “behavior” class that does not represent a problem domain concept – something made up, in order to support high cohesion, low coupling, and reuse.

Indirection	How to assign responsibilities to avoid direct coupling? Assign the responsibilities to an intermediate object to mediate between other components or services, so that they are not directly coupled.
--------------------	---

Protected Variations	<p>How to assign responsibility to objects, subsystems and systems so that the variations or instability in these elements do not have an undesirable impact on other elements?</p> <p>Identify points of predicted variation or instability; assign responsibilities to create a stable “interface” around them.</p>
-----------------------------	---

4.3.1 Creator

Problem

Who should be responsible for creating a new instance of some class?

One of the most common activities in object oriented system is creation of objects. General principle is applied for the assignment of creation responsibilities.

Design supports :

- 1) low coupling
- 2) increased clarity
- 3) encapsulation
- 4) reusability.

Solution

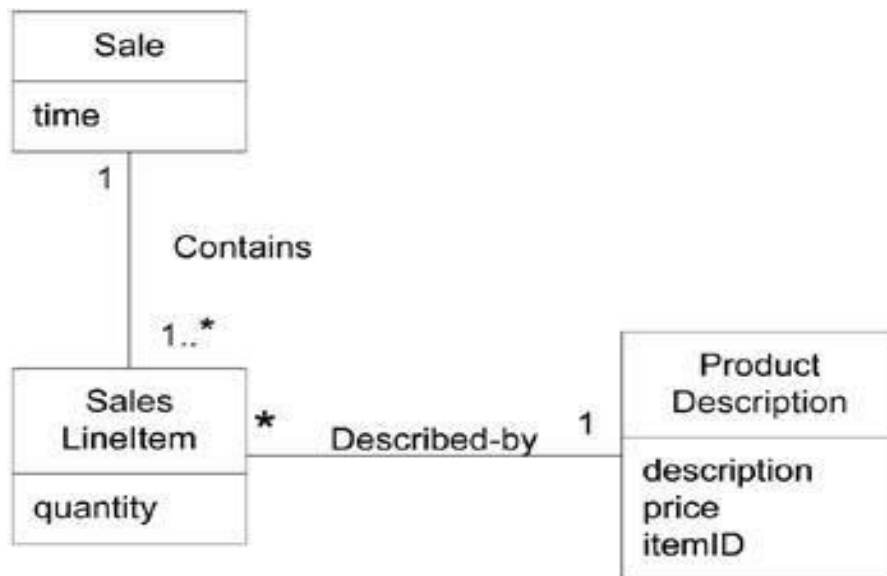
Assign class B the responsibility to create an instance of class A if one of these is true

- B "contains" or compositely aggregates A.
- B records A.
- B closely uses A.
- B is an expert while creating A (B passes the initializing data for A that is passed to A when created.)

B is a creator of A objects. If more than one option applies, usually prefer a class B which aggregates or contains class A.

Example:

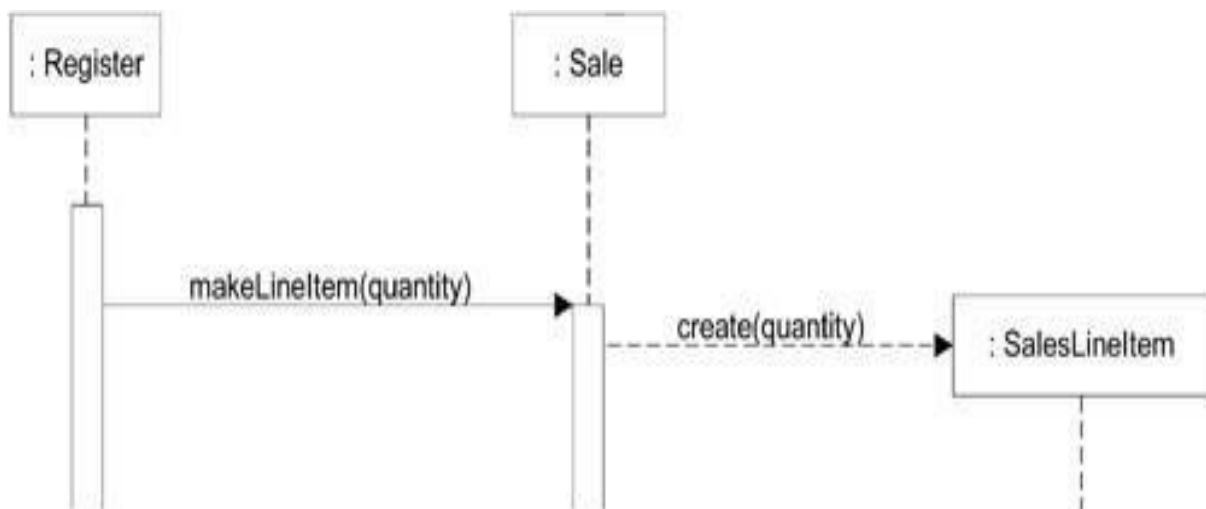
In the NextGen POS application, who should be responsible for creating a SalesLineItem instance? By Creator, we should look for a class that aggregates, contains, and so on, SalesLineItem instances. Consider the partial domain model in [Figure](#)



Partial Domain Model

Here “Sale “ takes the responsibility of creating ‘SalesLineItem’ instance . Since sale contains many ‘SalesLineItem’ objects.

The assignment of responsibilities requires that a ‘makeLineitem’ must also be defined in ‘Sale’.



Creating a SalesLineItem

Creator guides the assigning of responsibilities related to the creation of objects, a very common task. The basic intent of the Creator pattern is to find a creator that needs to be connected to the created object in any event. Choosing it as the creator supports low coupling.

All very common relationships between classes in a class diagram are,

- Composite aggregate part
- Container **contains** Content
- Recorder **records**

Creator suggests that the enclosing container or recorder class is a good candidate for the responsibility of creating the thing contained or recorded.

Example:- ‘Payment’ instance while creation initialized with ‘sale’ total. ‘Sale’ is a candidate creator of ‘Payment’.

Contradictions:

Based upon some external property value, creation requires significant complexity like,

- Recycled instances for performances.
- Creating an instance from one of a family of similar classes based upon some external property value, etc.

In such cases we go for helper class called

- **Concrete Factory, and**
- **Abstract Factory**

Benefits of the creator

- Low coupling is supported which implies Lower maintenance and higher opportunities for reuse

4.3.2 INFORMATION EXPERT (OR EXPERT)

Problem

What is the general principle of assigning responsibilities to objects?

During Object Design, when the interactions between objects are defined, we make choices about the assignment of responsibilities to software classes. This makes the software easier to

- Maintain
- Understand and
- Extend

Solution

Assign a responsibility to the information expert, the class that has the *information* necessary to fulfill the responsibility.

Example:-

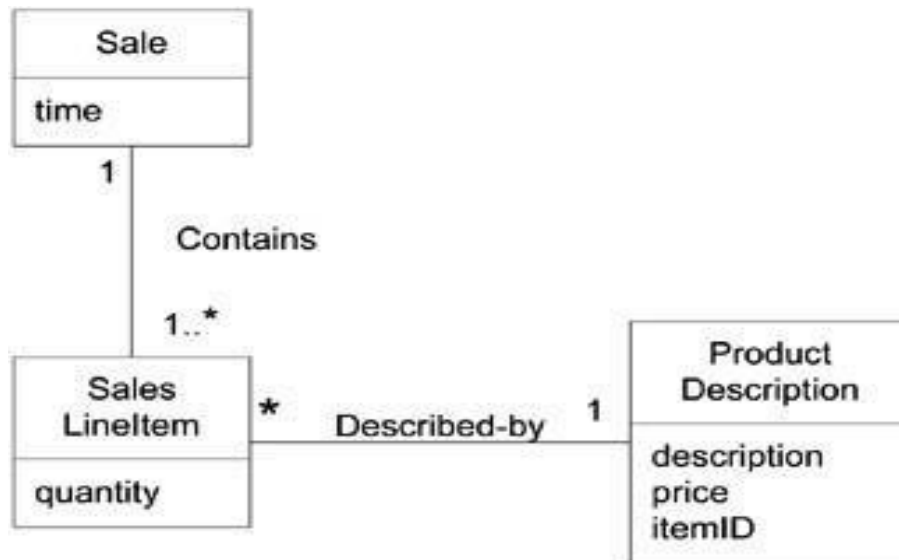
NextGEN POS Application

Some class needs to know the grand total of a sale. Start assigning responsibilities by clearly stating the responsibility.

1. Look at relevant classes in the Design Model if available,
2. Otherwise, look in the Domain Model

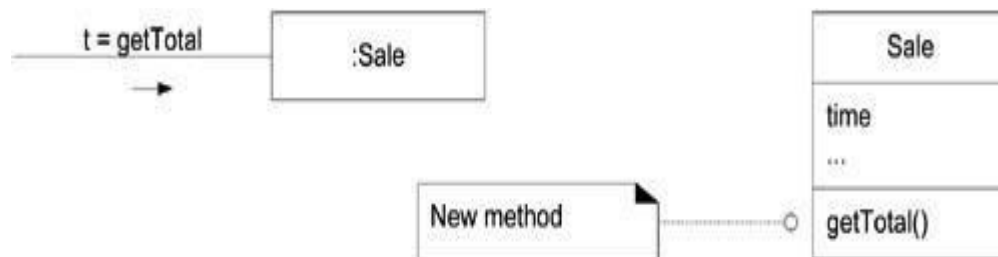
Example:-

If the design work has been just started, then look into the domain model, the real-world “sale”. In design model, software class ‘sale’ is added with the responsibility for getting total with the method ‘getTotal’.



Partial domain model for association of sale

After adding the `getTotal()`, the partial interaction and class diagrams given as :



To determine expert using the above the **SalesLineItem** should determine subtotal.

- **SalesLineItem Quantity**
- **ProductDescription Price**

By information expert using the above the **Sales LineItem** should determine subtotal. This is done by **Sale** sending `get Subtotal` messages to each **Sales LineItem** and sum the results.

UML diagram illustrating a sequence diagram for a `Sale` object. The diagram shows a message `t = getTotal()` sent to the `: Sale` object. This is followed by a loop `1 * st = getSubtotal()` that iterates over all elements of the `lineItems` collection (of type `SalesLineItem`). A note explains that this notation implies iterating over all elements of a collection. A `New method` box is also shown, indicating a new method `getSubtotal()` is being added to the `SalesLineItem` class.

```
sequenceDiagram
    participant Sale as : Sale
    participant SalesLineItem as SalesLineItem
    Note over Sale: t = getTotal()
    Note over Sale: 1 * st = getSubtotal()
    Note over SalesLineItem: lineItems[i] : SalesLineItem
    Note over SalesLineItem: New method
    Note over SalesLineItem: getSubtotal()
```

Design Class	Responsibility
ProductDescription	knows product price

- The Information Expert is frequently used in the assignment of responsibilities
- Experts express the common "intuition" that objects do things related to the information they have.
- Partial information experts will collaborate in the task.
- For example:- Sales total problem experts will collaborates in the task.
- Information expert thus has real world analogy.
- Information experts are basic guiding principle used continuously in object design.

Contradictions

Solution suggested by Expert is undesirable, usually because of problems in coupling and cohesion.

To overcome this,

- Keep application logic in one place(like domain software objects)
- Keep database objects in another place(separate persistence services subsystem.
- Supporting a separation of major concerns improves coupling and cohesion in a design.

Benefits

- Information encapsulation is maintained since objects use their own information to fulfill tasks.
- High cohesion is usually supported

4.3.3 LOW COUPLING

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.

An element with low (or weak) coupling is not dependent on too many other elements.

Types of coupling

Low coupling or weak coupling	High coupling or strong coupling
An element if does not depend on too many other elements like classes, subsystems and systems it is having low coupling.	<p>A class with high coupling relies on many other classes.</p> <p>The Problem of high coupling are</p> <ul style="list-style-type: none"> • Forced local changes because of changes in related classes. • Harder to understand in isolation. • Harder to reuse because its use requires the additional presence of the classes on which it is dependent

Problem :

How to support low dependency, low change impact, and increased reuse?

Solution :

Assign a responsibility so that coupling remains low. Use this principle to evaluate alternatives.

Example:-

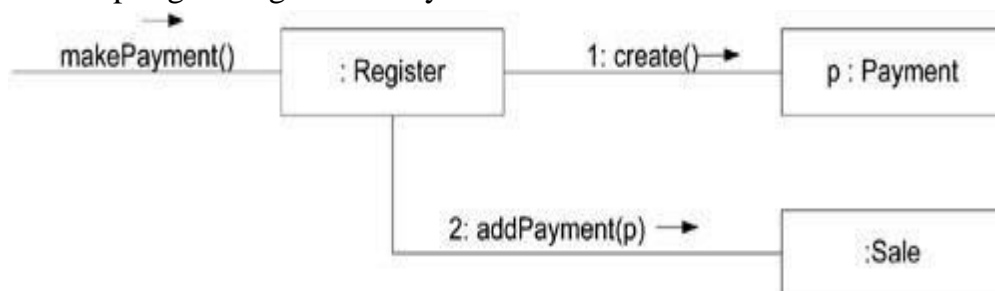


NextGen case Study

We have to create payment instance and associate it with sale.

DESIGN 1: *Suggested by creator*

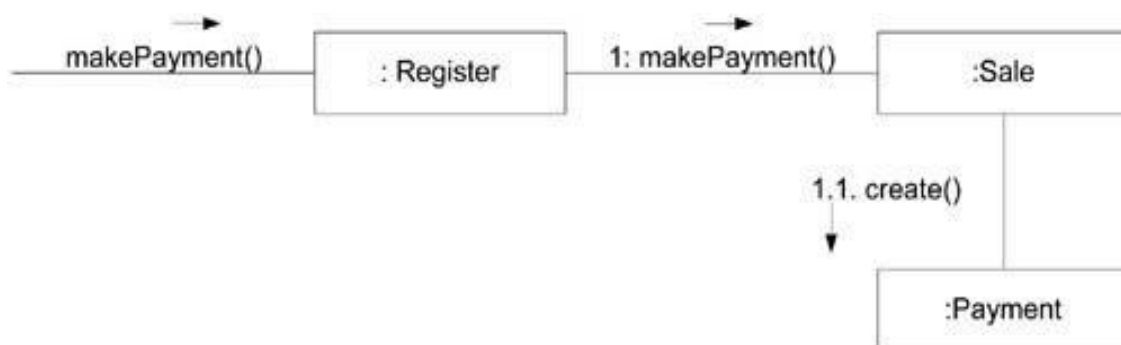
- 1) The Register creates the Payment and
- 2) It adds coupling of Register to Payment.



Register creates Payment

DESIGN 2: *Suggested by low coupling*

- 1) The Sale does the creation of a Payment and
- 2) It does not increase the coupling.



Sales creates Payment

Low coupling is an evaluation principle for evaluating all designs decisions. In object-oriented languages such as C++, Java, and C#, common forms of coupling from TypeX to TypeY include the following:

- TypeX has an attribute (data member or instance variable) that refers to a TypeY instance, or TypeY itself.
- A TypeX object calls on services of a TypeY object.
- TypeX has a method that references an instance of TypeY, or TypeY itself, by any means. These typically include a parameter or local variable of type TypeY, or the object returned from a message being an instance of TypeY.
- TypeX is a direct or indirect subclass of TypeY.
- TypeY is an interface, and TypeX implements that interface.

Contradictions

High coupling to stable elements and to pervasive elements is a problem. For example, a J2EE application can safely couple itself to the Java libraries

Benefits

- Not affected by changes in other components
- Simple to understand in isolation
- Convenient to reuse

4.3.4 CONTROLLER

Problem

What first object beyond the UI layer receives and coordinates ("controls") a system operation?

System operations were first explored during the analysis of SSD. These are the major input events upon our system.

Example:-

- 1) When a cashier using a POS terminal presses the "End Sale" button indicating "sale has ended".
- 2) When a writer using a word processor presses the "spell check" button to perform checking of spelling.

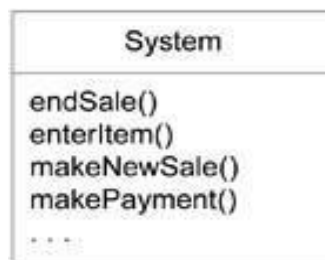
A **controller** is the first object beyond the User Interface (UI) layer that is responsible for receiving or handling a system operation message.

Solution

Assign the responsibility to one of the following

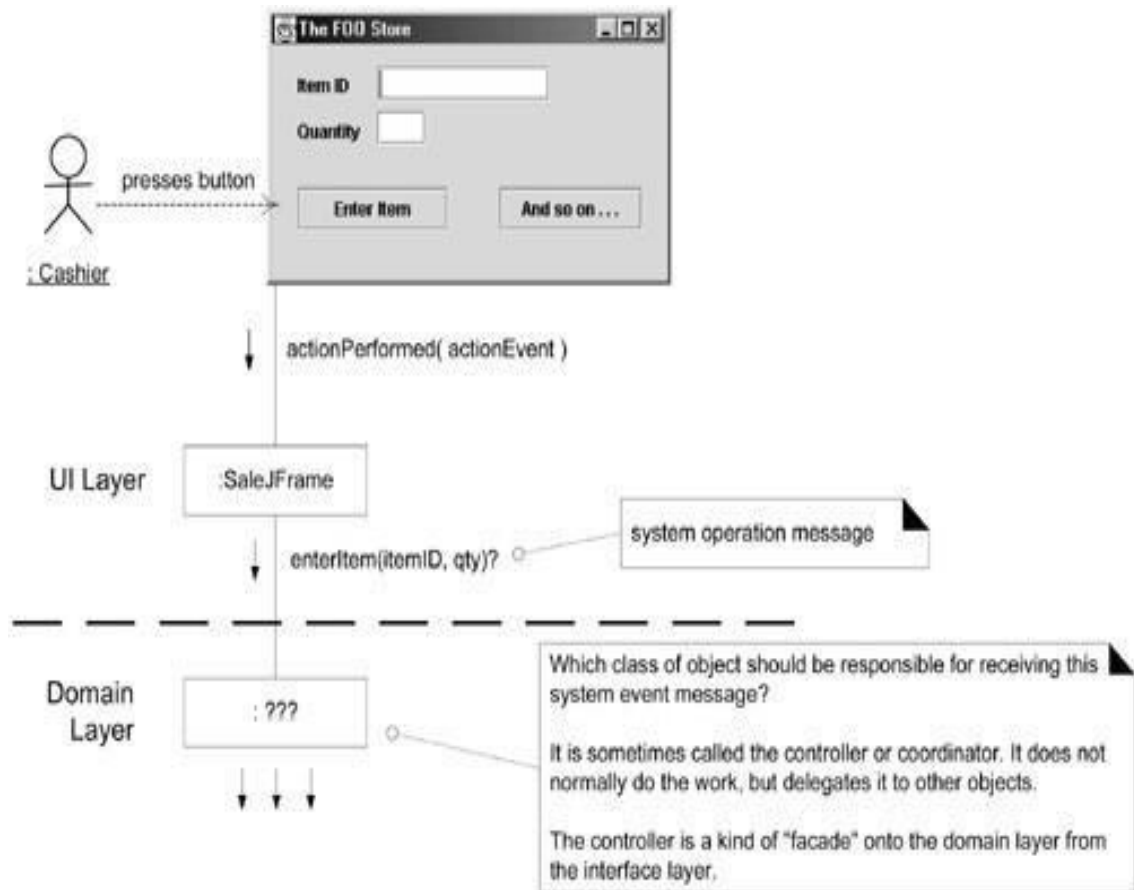
- Represents the overall "system," a "root object,"
 - These are all variations of a facade controller.
- Represents a use case scenario called
 - <UseCaseName>Handler,
 - <UseCaseName>Coordinator, or
 - <UseCaseName>Session
 - Use the same controller class for all system events in the same use case scenario.
 - Informally, a session is an instance of a conversation with an actor. Sessions can be of any length but are often organized in terms of use cases (use case sessions)

Example: NextGen application contains several system operations.



Some system operations of NextGen POS Application.

During the design the responsibility of system operations Is done by the controller.

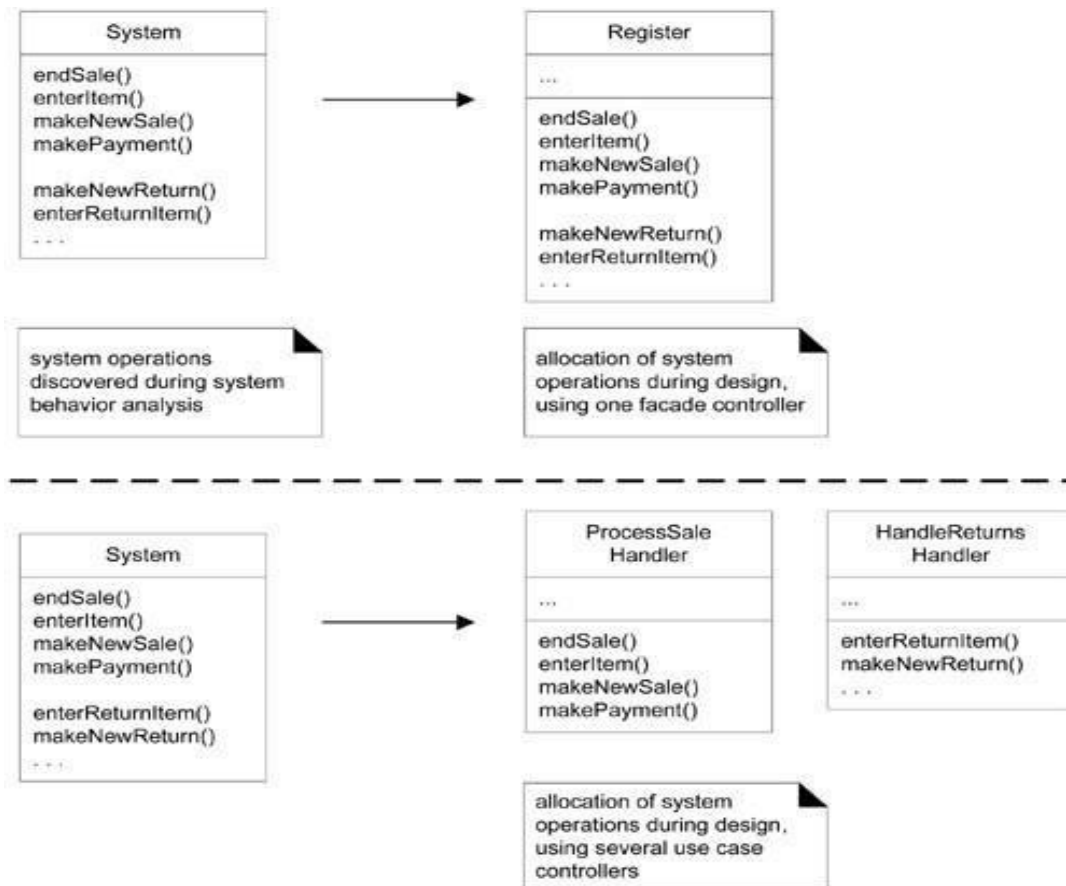


The controller pattern some choices are,

Represents the overall "system," "root object," device, or subsystem.	Register, POSSystem	
Represents a receiver or handler of all system events of a use case scenario.	ProcessSaleHandler, ProcessSaleSession	

A controller should assign other objects the work that needs to be done. It coordinates or controls the activity. Same controller class can be used for all system events to maintain information about the state of use case. A common defect in the design of controllers is it suffers from bad cohesion.

The system operations identified during system behavior analysis are assigned to one or more controller classes like,



Allocation of system operations

Controller

The Facade controller representing the overall system, device, or a subsystem. facade controller representing the overall system, device, or a subsystem.

Facade controllers

- 1) Facade controllers are suitable when there are not "too many" system events,
- 2) When the user interface (UI) cannot redirect system event messages to alternating controllers, such as in a message-processing system.

Use case controller

A use case controller is a good choice when there are many system events across different processes; it factors their handling into manageable separate classes and also provides a basis for knowing and reasoning about the state of the current scenario in progress.

Guideline

Normally, a controller should delegate to other objects the work that needs to be done; it coordinates or controls the activity. It does not do much work itself.

Benefits

1. Increased potential for reuse and pluggable interfaces .

2 Opportunity to reason about the state of the use case.

Implementation

The code has

- Process JFrame window referring to domain controller object – Register.
- Define handler for button click.
- Show key message – sending enterItem message to the controller.

Code

```
public class ProcessSaleJFrame extends JFrame
{
    // the window has a reference to the 'controller' domain object
(1) private Register register;

    // the window is passed the register, on creation
    public ProcessSaleJFrame(Register _register)
    {
        register = _register;
    }

    // this button is clicked to perform the
    // system operation "enterItem"
    private JButton BTN_ENTER_ITEM;
    -----
    -----
    -----

(2) BTN_ENTER_ITEM.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            // utility class

            ---
            ---
            ---

(3) register.enterItem(id, qty);
        }
    } ); // end of the addActionListener call
    return BTN_ENTER_ITEM;
} // end of method
// ...
} // end of class
```

Bloated Controllers

Issues and Solutions

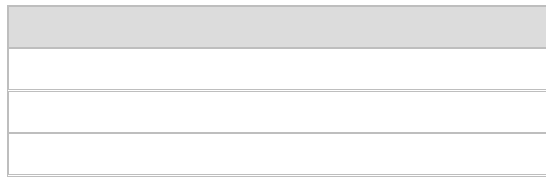
Poorly designed, a controller class will have low cohesion unfocused and handling too many areas of responsibility; this is called a bloated controller.

Signs of bloating are:

- There is only a single controller class receiving all system events in the system, and there are many of them. This sometimes happens if a facade controller is chosen.
- The controller itself performs many of the tasks necessary to fulfill the system event, without delegating the work. This usually involves a violation of Information Expert and High Cohesion.
- A controller has many attributes, and it maintains significant information about the system or domain, which should have been distributed to other objects, or it duplicates information found elsewhere.

Among the **Cures for a bloated controller** are these two:

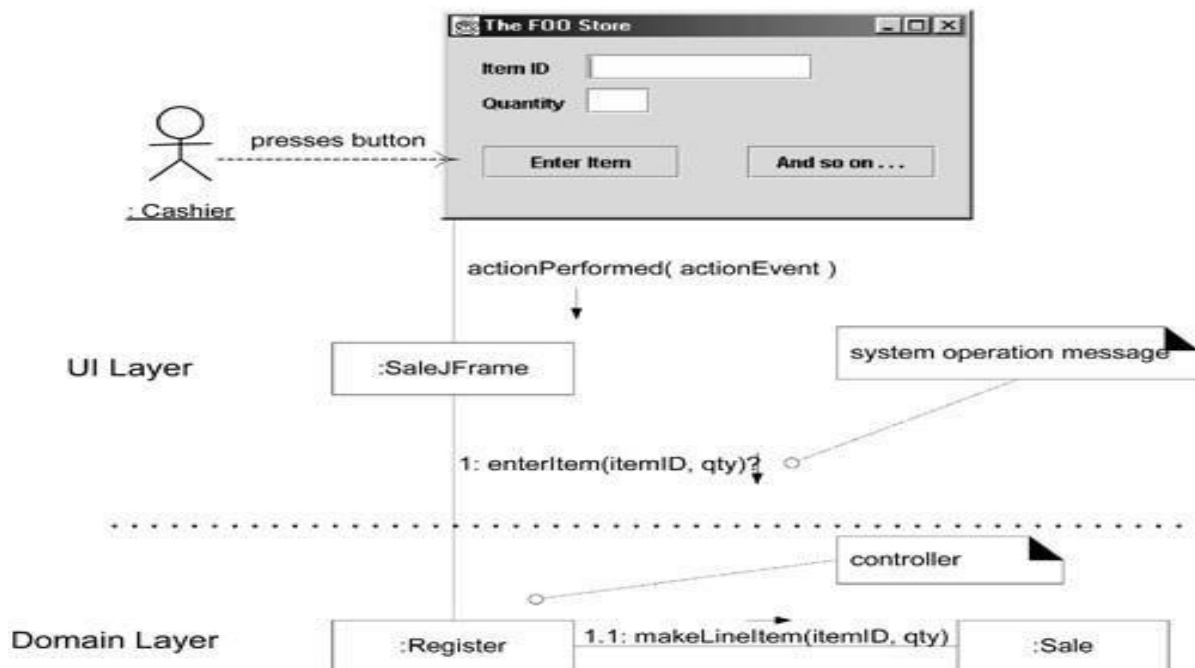
1. Add more controllers a system does not have to need only one. For example, consider an application with many system events, such as an airline reservation system.



2. Design the controller so that it primarily delegates the fulfillment of each system operation responsibility to other objects.

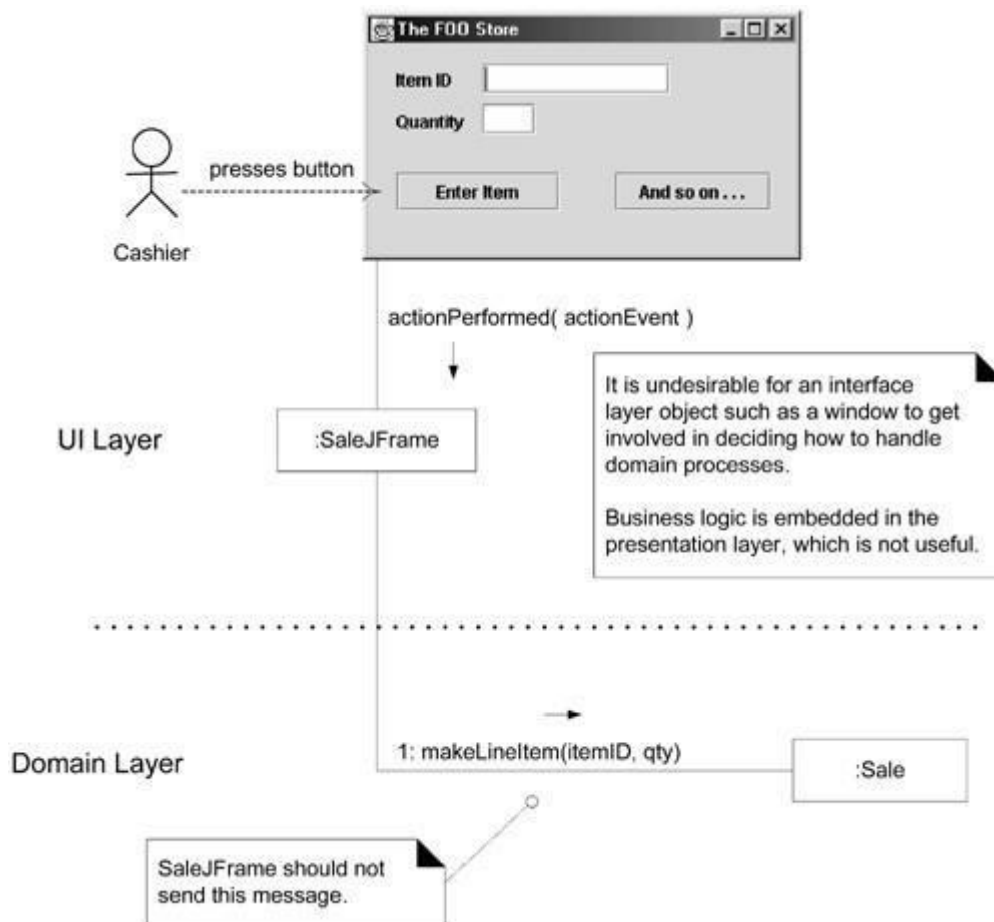
UI Layer Does Not Handle System Events

An important corollary of the Controller pattern is that UI objects (for example, window objects) and the UI layer should not have responsibility for handling system events. Assume the NextGen application has a window that displays sale information and captures cashier operations.



Desirable coupling of UI layer to domain layer

Assigning the responsibility for system operations to objects in the application or domain layer by using the Controller pattern rather than the UI layer can increase reuse potential. If a UI layer object (like the SaleJFrame) handles a system operation that represents part of a business process, then business process logic would be contained in an interface (for example, window-like) object; the opportunity for reuse of the business logic then diminishes because of its coupling to a particular interface and application.



Less desirable coupling of interface layer to domain layer

4.3.5 HIGH COHESION

Problem

How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?

Cohesion (or more specifically, functional cohesion) is a measure of how strongly related and focused the responsibilities of an element are. An element with highly related responsibilities that does not do a tremendous amount of work has high cohesion. These elements include classes, subsystems, and so on.

Solution

Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives.

A class with low cohesion does many unrelated things or does too much work. Such classes are undesirable; they suffer from the following problems:

- hard to comprehend
- hard to reuse
- hard to maintain
- delicate; constantly affected by change

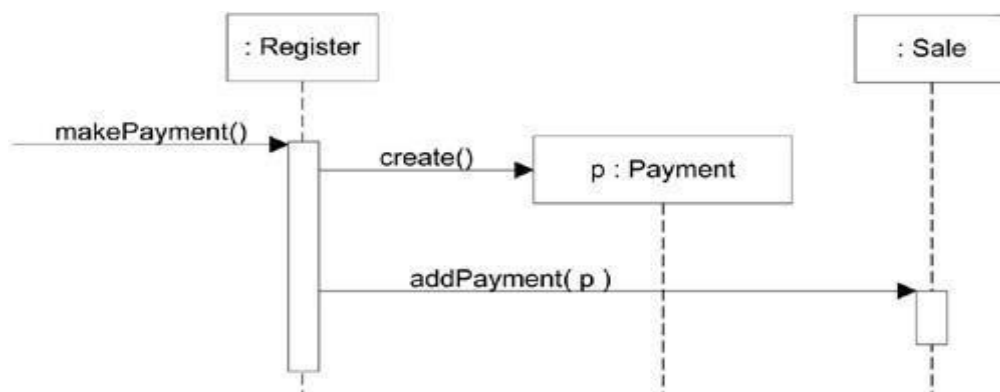
Low cohesion classes often represent a very "large grain" of abstraction or have taken on responsibilities that should have been delegated to other objects.

Example

Create a payment instance and associate it with sale

DESIGN 1

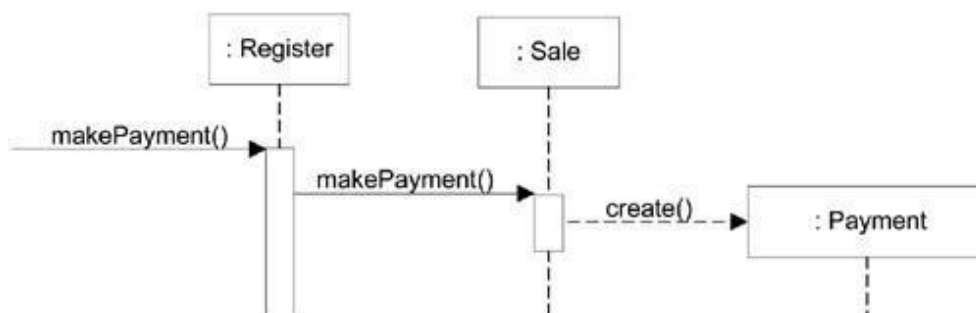
- Register records a Payment in the real-world domain, the Creator pattern suggests Register as a candidate for creating the Payment.
- The Register instance could then send an addPayment message to the Sale, passing along the new Payment as a parameter.



Register creates Payment

DESIGN 2

The second design delegates the payment creation responsibility to the Sale supports higher cohesion in the Register.



Sale creates Payment

- In the second design, payment creation is the responsibility of sale.
 - a. It is highly desirable because it supports High Cohesion & Low Coupling

Scenarios of varying degrees of functional cohesion

1. **Very low cohesion**: A class is solely responsible for many things in very different functional areas.

Ex : Assume the existence of a class called RDB-RPC-Interface which is completely responsible for interacting with relational databases and for handling remote procedure calls. These are two vastly different functional areas, and each requires lots of supporting code.

2. **Low cohesion**: A class has sole responsibility for a complex task in one functional area.

Ex: Assume the existence of a class called RDBInterface which is completely responsible for interacting with relational databases. The methods of the class are all related, but there are lots of them, and a tremendous amount of supporting code; there may be hundreds or thousands of methods.

3. **High cohesion**: A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks.

Ex: Assume the existence of a class called RDBInterface that is only partially responsible for interacting with relational databases. It interacts with a dozen other classes related to RDB access in order to retrieve and save objects.

4. **Moderate cohesion**: A class has lightweight and sole responsibilities in a few different areas that are logically related to the class concept but not to each other.

Ex: Assume the existence of a class called Company that is completely responsible for (a) knowing its employees and (b) knowing its financial information. These two areas are not strongly related to each other, although both are logically related to the concept of a company.

Rule of thumb

A class with high cohesion has a relatively small number of methods, with highly related functionality, and does not do too much work. It collaborates with other objects to share the effort if the task is large.

- Easy to maintain
- Understand and
- Reuse

Modular Design

Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules. Modular design creates methods and classes with single purpose, clarity and high cohesion.

Lower cohesion is had in

- Grouping responsibilities or code into one class or component.
- Distributed server objects.

Benefits

- Clarity and ease of comprehension of the design is increased.
- Maintenance and enhancements are simplified.
- Low coupling is often supported.
- Reuse of fine-grained, highly related functionality is increased because a cohesive class can be used for a very specific purpose.

4.4 APPLYING GOF DESIGN PATTERNS

In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development.

These authors are collectively known as **Gang of Four (GOF)**. According to these authors design patterns are primarily based on the following principles of object orientated design.

- Program to an interface not an implementation
- Favor object composition over inheritance

Pattern & Description

There are 23 design patterns which can be classified in three categories:

Creational Patterns : These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.

Structural Patterns : These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

Behavioral Patterns : These design patterns are specifically concerned with communication between objects.

The 23 design patterns are listed below:

		Purpose		
		Creational	Structural	Behavioral
Type	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

4.4.1 Creational Patterns

- Make the system independent of how objects are created composed and represented
- Abstract the instantiation process
 - Hide how instances of these classes are created and assembled
 - Hide references to concrete classes used in the system
- Govern the what, when, who , how object creation

1. **Abstract Factory:** Creates an instance of several families of classes. Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
2. **Builder:** Separates object construction from its representation. Separate the construction of a complex object from its representation so that the same construction processes can create different representations.
3. **Factory Method:** Creates an instance of several derived classes. Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

4. **Prototype:** A fully initialized instance to be copied or cloned. Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
5. **Singleton:** A class of which only a single instance can exist. Ensure a class only has one instance, and provide a global point of access to it.

FACTORY

Name	Factory
Problem:	Who should be responsible for creating objects when there are special considerations, such as complex creation logic, a desire to separate the creation responsibilities for better cohesion, and so forth?
Solution: (advice)	Create a Pure Fabrication object called a Factory that handles the creation.

This is also called as

- Simple Factory or
- Concrete Factory.

This pattern is not a GoF design pattern, but extremely widespread. It is also a simplification of the GoF Abstract Factory pattern. The adapter raises a new problem in the design,

- Who create adapters?
- How to create adapters?

When domain objects create the adapter, their responsibilities are beyond pure application logic and related to connectivity with other software components.

So, when a domain object creates adapters,

- It does not support goal of separation of concerns.
- It lowers cohesion.

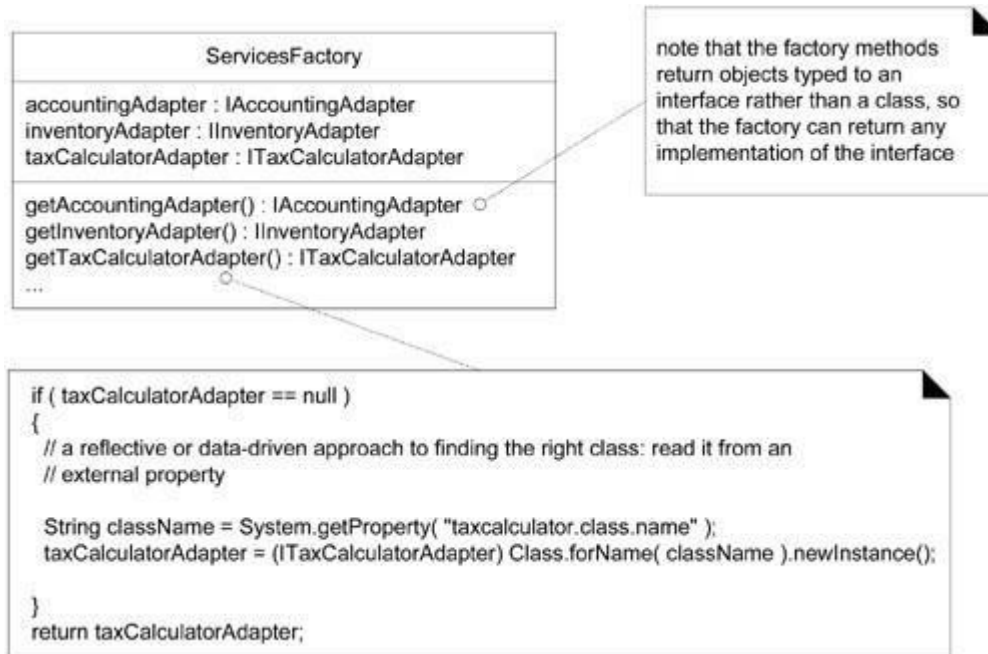
So, we go in for 'factory' pattern, when pure fabrication "factory" object is defined to create objects.

Advantages of factory

A common alternative in this case is to apply the Factory pattern, in which a Pure Fabrication "factory" object is defined to create objects.

Factory objects have several advantages:

- Separate the responsibility of complex creation into cohesive helper objects.
- Hide potentially complex creation logic.
- Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling.



The Factory pattern

In the `ServicesFactory`, the logic to decide which class to create is resolved by reading in the class name from an external source and then dynamically loading the class. This is an example of a partial data-driven design.

4.4.2 Structural Patterns

- Help identify and describe relationship between entities
- Address how classes and objects are composed to form large structures
- Class oriented pattern use inheritance to compose interfaces and implementation
- Object Oriented Patterns describe ways to compose objects to realize new functionality , possibly by changing the composition at runtime.

These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

1. **Adapter:** Match interfaces of different classes. Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

2. **Bridge:** Separates an object's interface from its implementation. Decouple an abstraction from its implementation so that the two can vary independently.
3. **Composite:** A tree structure of simple and composite objects. Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
4. **Decorator:** Add responsibilities to objects dynamically. Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
5. **Facade:** A single class that represents an entire subsystem. Provide a unified interface to a set of interfaces in a system. Facade defines a higher-level interface that makes the subsystem easier to use.
6. **Flyweight:** A fine-grained instance used for efficient sharing. Use sharing to support large numbers of fine-grained objects efficiently. A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context — it's indistinguishable from an instance of the object that's not shared.
7. **Proxy:** An object representing another object. Provide a surrogate or placeholder for another object to control access to it.

ADAPTER

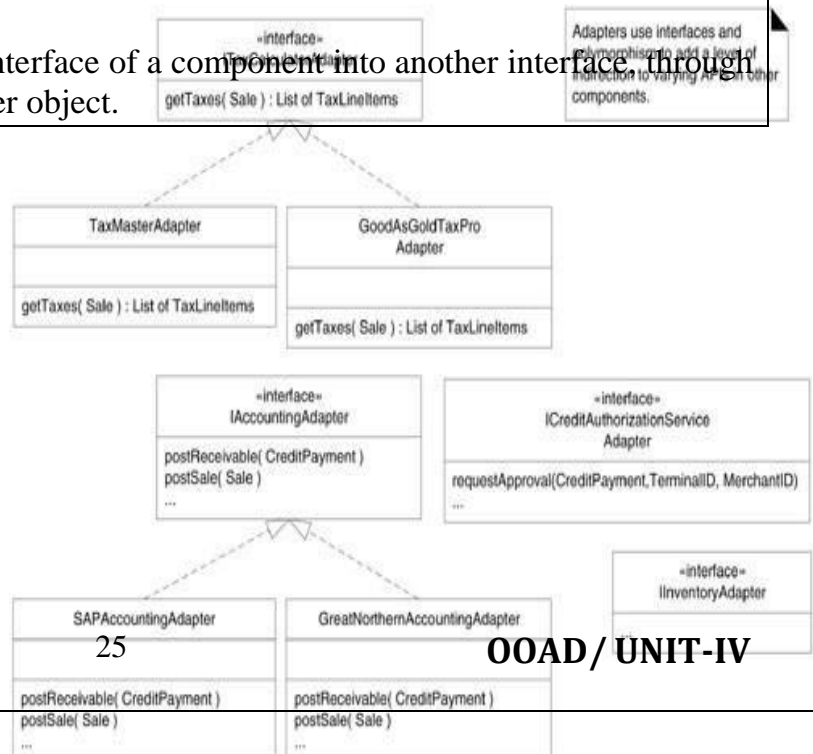
Name	Adapter
Problem:	How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?
Solution: (advice)	Convert the original interface of a component into another interface through an intermediate adapter object.

Adapters use interfaces and composition to add a level of indirection to varying API on other components.

Example:

The NextGen POS system needs to supports many third party services like,

- Tax calculators
- Credit authorization
- Inventory systems
- Accounting systems etc.,

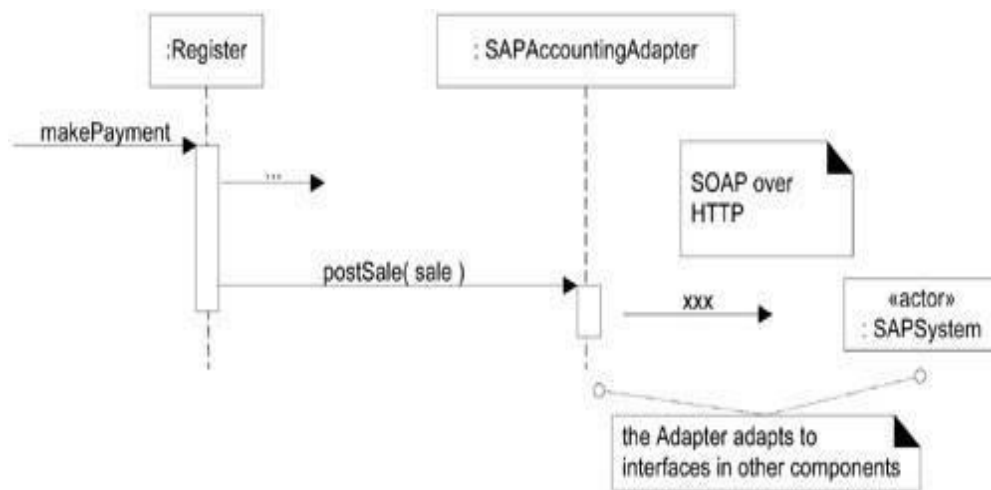


Add a level of indirection with objects to adapt to the solution.

The Adapter pattern

Here a particular adapter instance will be instantiated , such as

- SAP for accounting, and will adapt the postSale request to the external interface.
- SOAP XML interface over HTTPS for an intranet Web services.



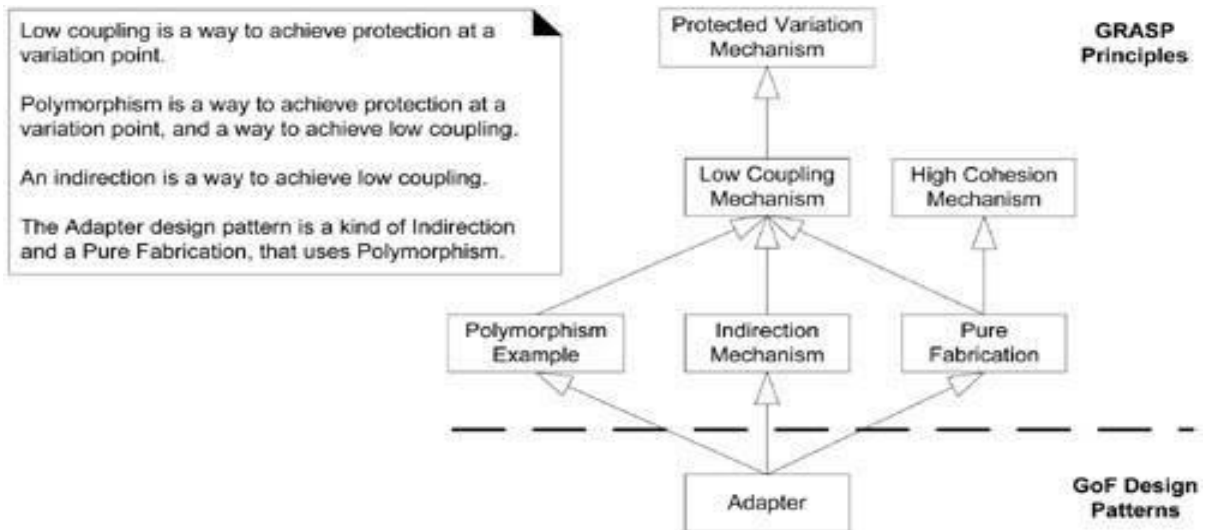
Using an Adapter

The type names include the pattern name "Adapter." This is a relatively common style and has the advantage of easily communicating to others reading the code or diagrams what design patterns are being used.

GRASP Principles as a Generalization of Other Patterns

The Adapter pattern can be viewed as a specialization of some GRASP building blocks. Adapter supports Protected Variations with respect to changing external interfaces or third-party packages through the use of an Indirection object that applies interfaces and Polymorphism.

Example: Adapter and GRASP



Relating Adapter to some core GRASP principles

BRIDGE

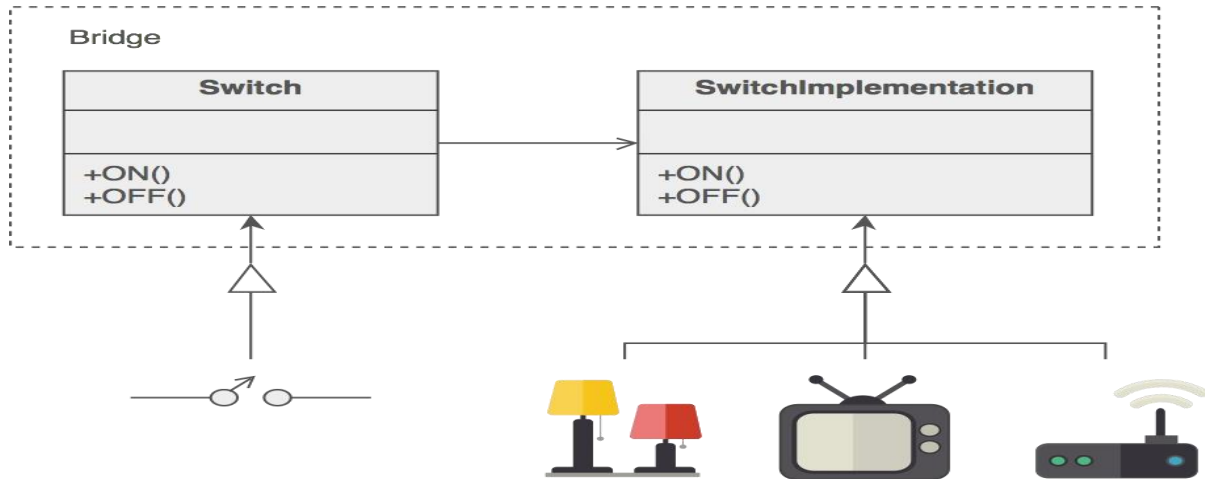
Name:	Bridge
Problem:	To decouple the implementation from its abstraction
Solution: (advice)	Decouple an abstraction from its implementation so that the two can vary independently

The bridge uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes. Bridge design pattern is a modified version of the notion of “prefer composition over inheritance”.

- **Creates two different hierarchies. One for abstraction and another for implementation.**
- Avoids permanent binding by removing the dependency between abstraction and implementation.
- **We create a bridge that coordinates between abstraction and implementation.**
- Abstraction and implementation can be extended separately.
- **Should be used when we have need to switch implementation at runtime.**
- Client should not be impacted if there is modification in implementation of abstraction.
- **Best used when you have multiple implementations.**

Example :

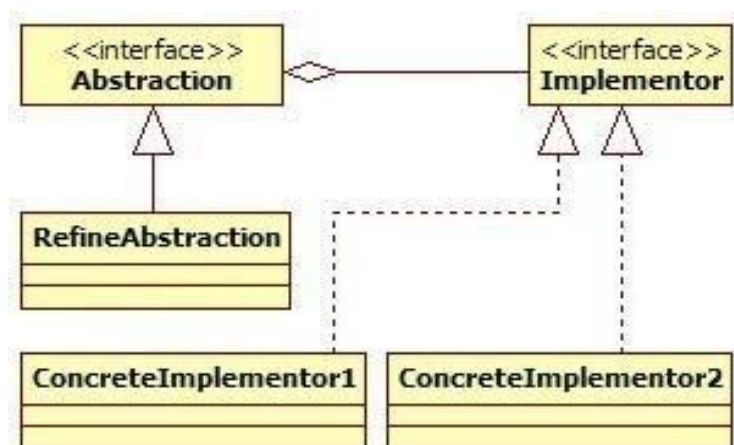
A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches.



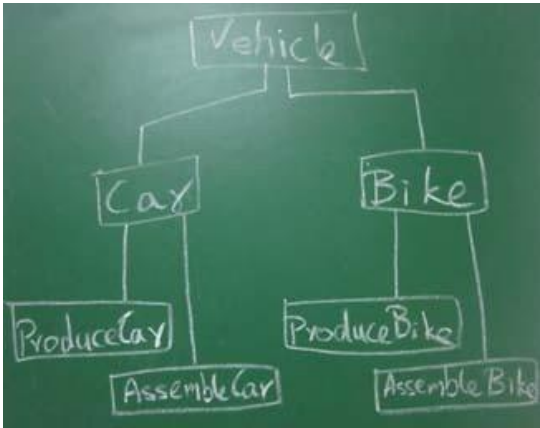
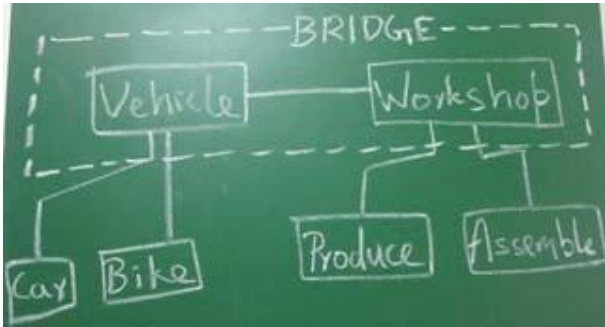
Elements of Bridge Design Pattern

- **Abstraction** – core of the bridge design pattern and defines the crux (Create , Retrieve, Update , Delete) Contains a reference to the implementer.
- **Refined Abstraction** – Extends the abstraction takes the finer detail one level below. Hides the finer elements from implementers.
- **Implementer** – This interface is the higher level than abstraction. Just defines the basic operations.
- **Concrete Implementation** – Implements the above implementer by providing concrete implementation.

Generic UML Diagram for Bridge Design Pattern



Need for Bridge Design Pattern

Without Bridge Pattern	With Bridge Pattern
	
When there are inheritance hierarchies creating concrete implementation, you lose flexibility because of interdependence.	Decouple implementation from interface and hiding implementation details from client is the essence of bridge design pattern.

Example : for core elements of Bridge Design Pattern

Vehicle -> Abstraction
manufacture()

Car -> Refined Abstraction 1
manufacture()

Bike -> Refined Abstraction 2
manufacture()

Workshop -> Implementor
work()

Produce -> Concrete Implementation 1
work()

Assemble -> Concrete Implementation 2
work()

Example Coding :

```
// abstraction in bridge pattern
abstract class Vehicle {
    protected Workshop workShop1;
    protected Workshop workShop2;
    protected Vehicle(Workshop workShop1, Workshop workShop2) {
```

```

        this.workShop1 = workShop1;
        this.workShop2 = workShop2;
    }
    abstract public void manufacture();
}
// Refine abstraction 1 in bridge pattern
public class Car extends Vehicle {
    public Car(Workshop workShop1, Workshop workShop2) {
        super(workShop1, workShop2);
    }

    public void manufacture() {
        System.out.print("Car ");
        workShop1.work();
        workShop2.work();
    } }

public class Bike extends Vehicle {
    public Bike(Workshop workShop1, Workshop workShop2) {
        super(workShop1, workShop2);
    }
    public void manufacture() {
        System.out.print("Bike ");
        workShop1.work();
        workShop2.work();
    } }
// Implementor for bridge pattern
public interface Workshop {
    abstract public void work();
}
//Concrete implementation 1 for bridge pattern
public class Produce implements Workshop {
    public void work() {
        System.out.print("Produced"); }}

public class Assemble implements Workshop {
    public void work() {
        System.out.println(" Assembled.");
    } }
//Demonstration of bridge design pattern
public class BridgePattern {
    public static void main(String[] args) {
        Vehicle vehicle1 = new Car(new Produce(), new Assemble());
        vehicle1.manufacture();
        Vehicle vehicle2 = new Bike(new Produce(), new Assemble());
        vehicle2.manufacture();
    }
}

```


Output:

Car Produced Assembled.

Bike Produced Assembled.

4.4.3 Behavioral Patterns

- Describes algorithms assignments of responsibilities and interaction between objects.
 - Behavioral class patterns use inheritance to distribute behavior
 - Behavioral object patterns use composition
1. **Chain of Resp.** : A way of passing a request between a chain of objects. Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
 2. **Command:** Encapsulate a command request as an object. Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
 3. **Interpreter:** A way to include language elements in a program. Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
 4. **Iterator:** Sequentially access the elements of a collection. Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
 5. **Mediator:** Defines simplified communication between classes. Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
 6. **Memento:** Capture and restore an object's internal state. Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
 7. **Observer:** A way of notifying change to a number of classes. Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
 8. **State:** Alter an object's behavior when its state changes. Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

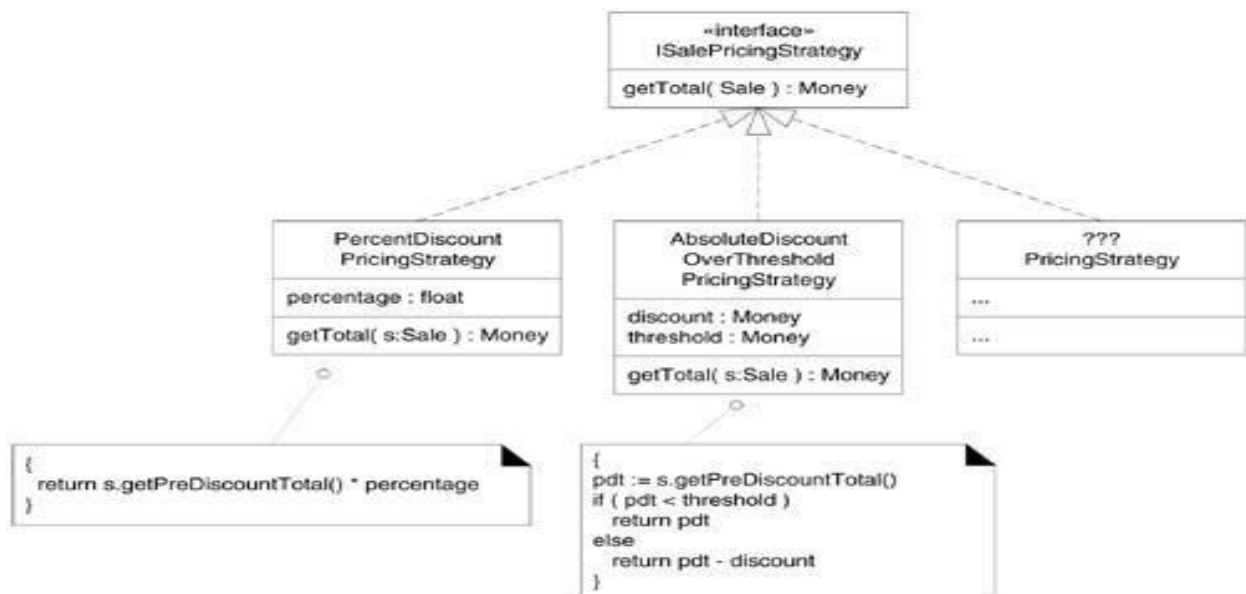
9. **Strategy:** Encapsulates an algorithm inside a class. Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
10. **Template:** Defer the exact steps of an algorithm to a subclass. Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
11. **Visitor:** Defines a new operation to a class without change. Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates

STRATEGY

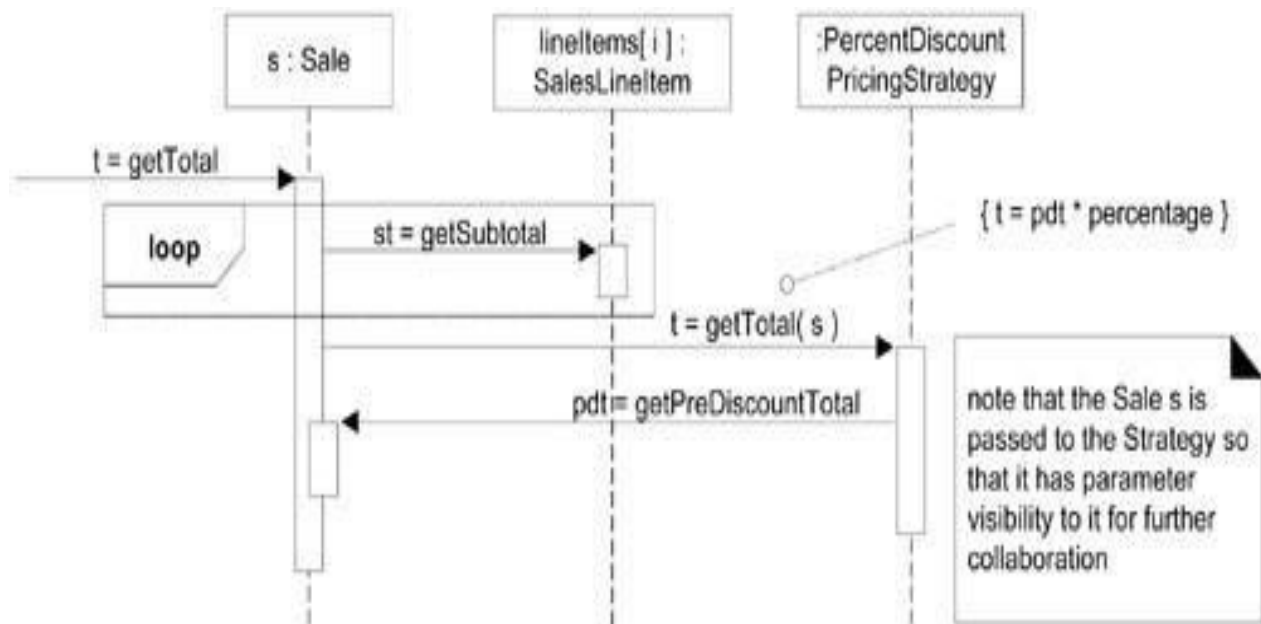
Name	Strategy
Problem:	How to design for varying, but related, algorithms or policies? How to design for the ability to change these algorithms or policies?
Solution: (advice)	Define each algorithm/policy/strategy in a separate class, with a common interface.

Example

Since the behavior of pricing varies by the strategy (or algorithm), we create multiple SalePricingStrategy classes, each with a polymorphic getTotal method . Each getTotal method takes the Sale object as a parameter, so that the pricing strategy object can find the pre-discount price from the Sale, and then apply the discounting rule. The implementation of each getTotal method will be different: PercentDiscountPricingStrategy will discount by a percentage, and so on.

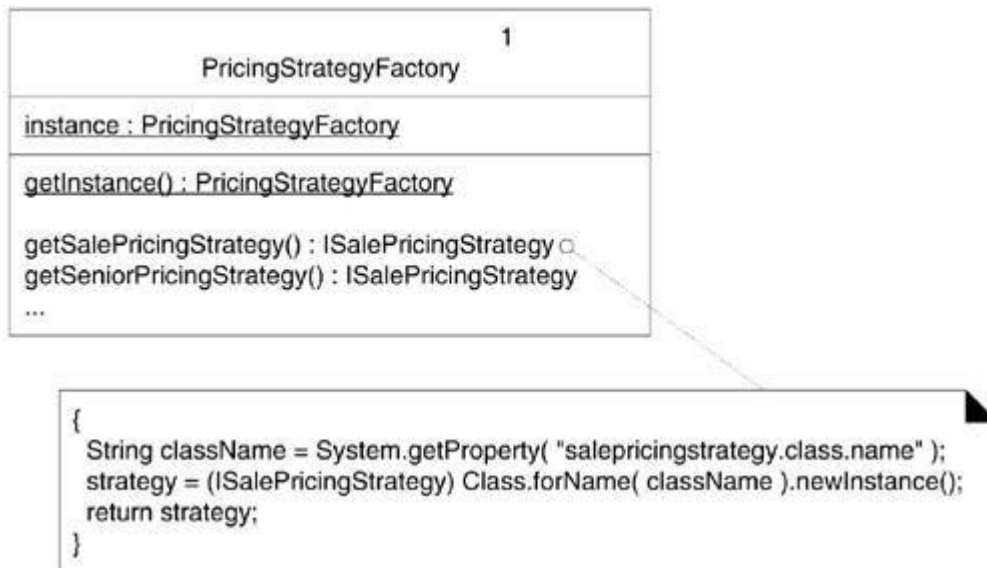


A strategy object is attached to a context object the object to which it applies the algorithm. In this example, the context object is a Sale. When a getTotal message is sent to a Sale, it delegates some of the work to its strategy object



Creating a Strategy with a Factory

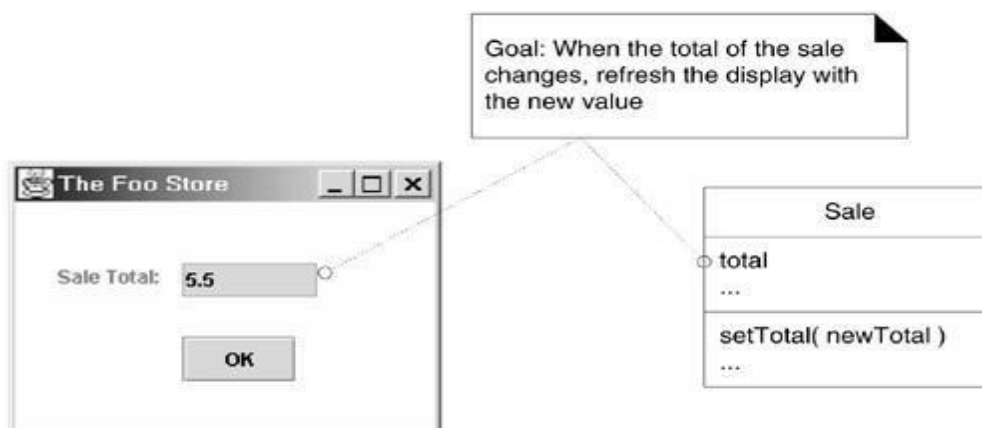
There are different pricing algorithms or strategies, and they change over time. Who should create the strategy? A straightforward approach is to apply the Factory pattern again: A PricingStrategyFactory can be responsible for creating all strategies (all the pluggable or changing algorithms or policies) needed by the application.



OBSERVER

Name:	Observer (Publish-Subscribe)
Problem:	Different kinds of subscriber objects are interested in the state changes or events of a publisher object, and want to react in their own unique way when the publisher generates an event. Moreover, the publisher wants to maintain low coupling to the subscribers. What to do?
Solution: (advice)	Define a "subscriber" or "listener" interface. Subscribers implement this interface. The publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.

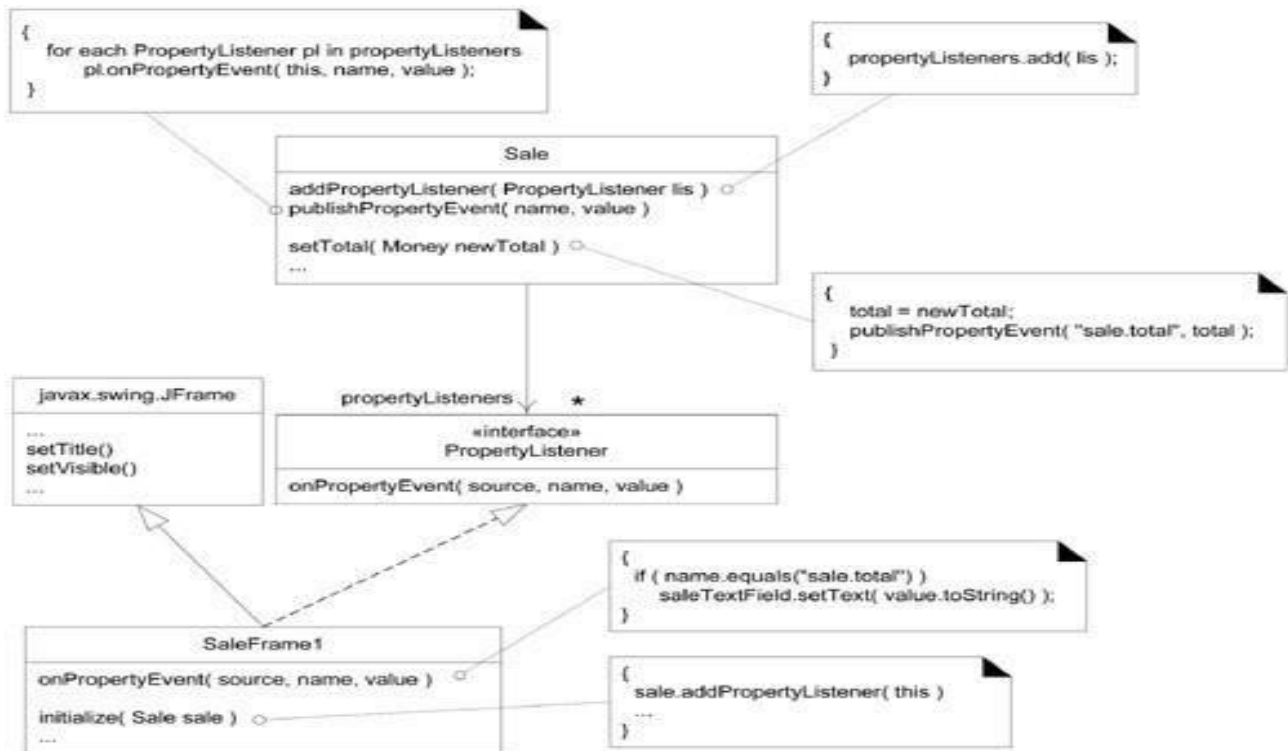
Example:



When the Sale changes its total, the Sale object sends a message to a window, asking it to refresh its display. To extend the solution found for changing data, add the ability for a GUI window to refresh its sale.

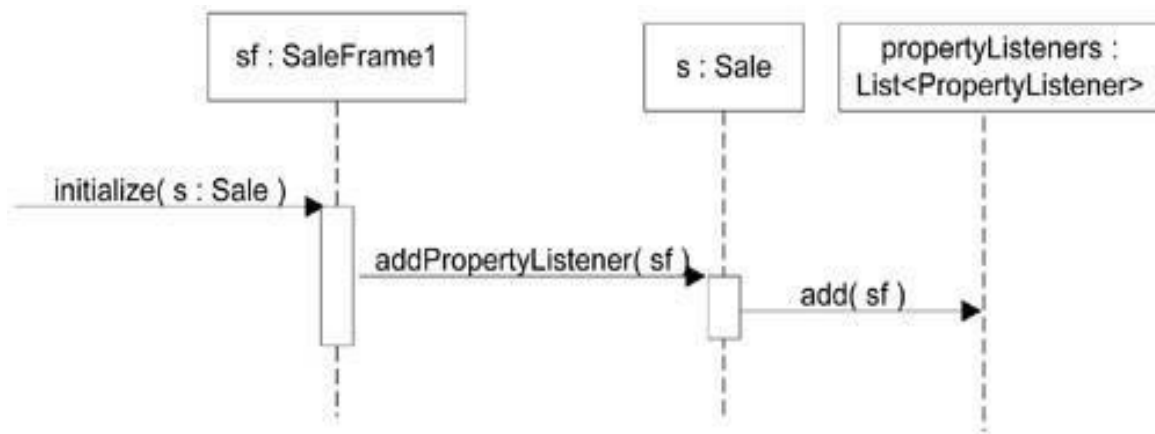
The Model-View Separation principle discourages such solutions. It states that "model" objects (non-UI objects such as a Sale) should not know about view or presentation objects such as a window. It promotes Low Coupling from other layers to the presentation (UI) layer of objects.

Its promotes Low coupling.



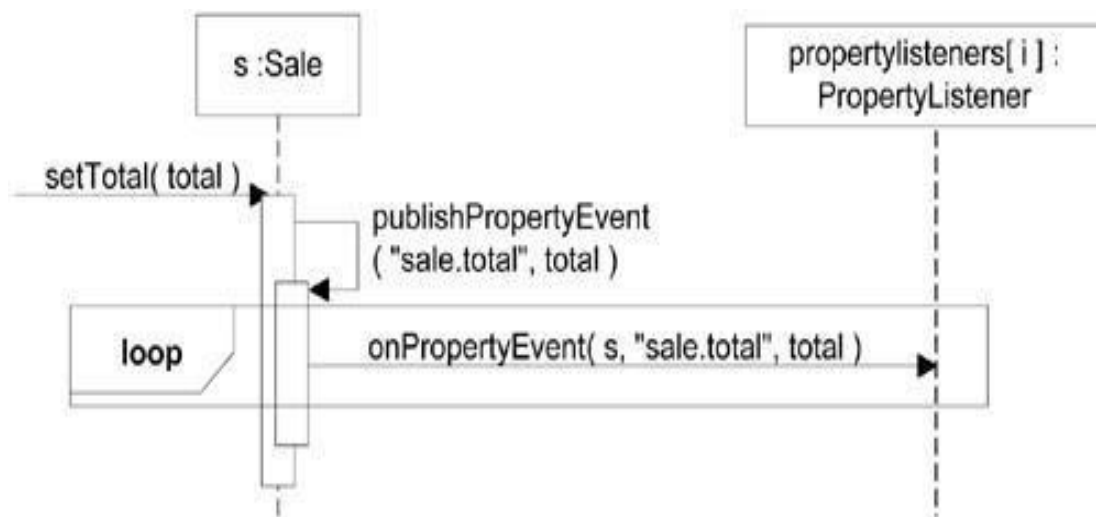
The major ideas and steps in this example:

1. An interface is defined; in this case, `PropertyListener` with the operation `onPropertyEvent`.
2. Define the window to implement the interface. `SaleFrame1` will implement the method `onPropertyEvent`.
3. When the `SaleFrame1` window is initialized, pass it the `Sale` instance from which it is displaying the total.
4. The `SaleFrame1` window registers or subscribes to the `Sale` instance for notification of "property events," via the `addPropertyListener` message.
5. The `Sale` does not know about `SaleFrame1` objects; rather, it only knows about objects that implement the `PropertyListener` interface.
6. The `Sale` instance is thus a publisher of "property events." When the total changes, it iterates across all subscribing `PropertyListeners`, notifying each.



The observer SaleFrame1 subscribes to the publisher Sale

When the Sale total changes, it iterates across all its registered subscribers, and "publishes an event" by sending the onPropertyEvent message to each.

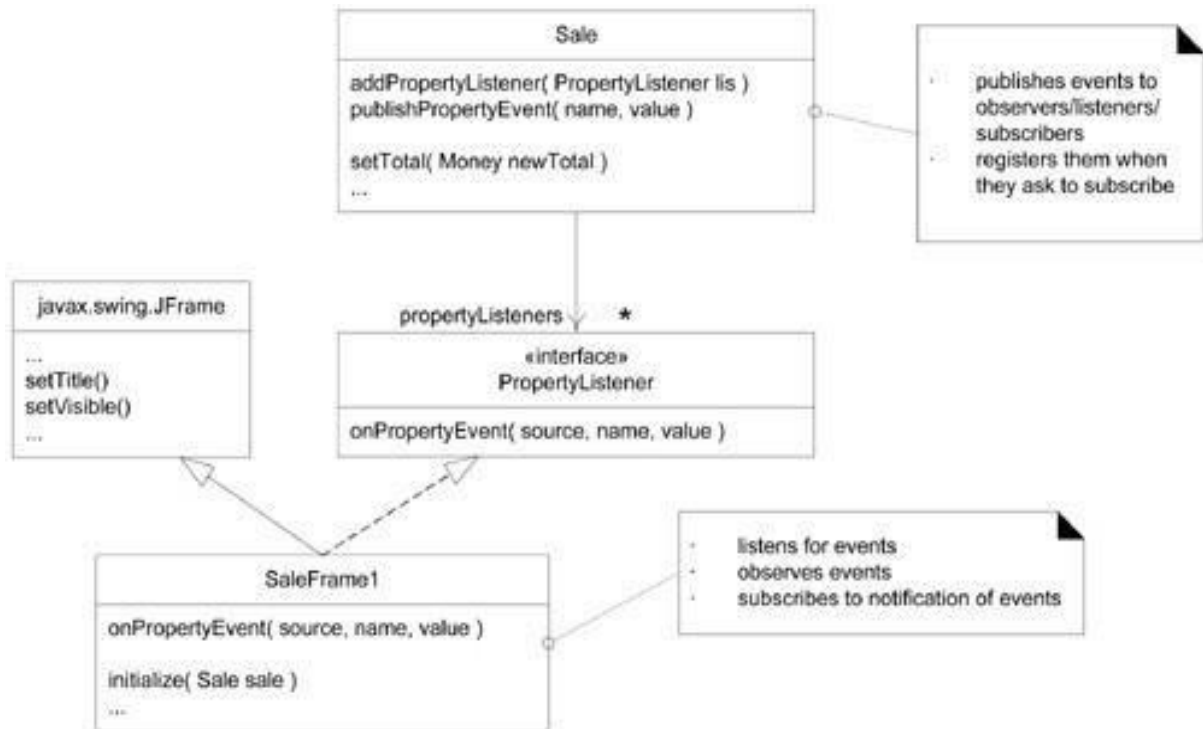


The Sale publishes a property event to all its subscribers

SaleFrame1, which implements the PropertyListener interface, thus implements an onPropertyEvent method.

When the SaleFrame1 receives the message, it sends a message to its JTextField GUI widget object to refresh with the new sale total.

- There is still some coupling from the model object (the Sale) to the view object. But it is a loose coupling to an interface independent of the presentation layer the PropertyListener interface. coupling to a generic interface of objects that do not need to be present, and which can be dynamically added (or removed), supports low coupling.
- Therefore, Protected Variations with respect to a changing user interface has been achieved through the use of an interface and polymorphism.



Who is the observer, listener, subscriber, and publisher

Why Is It Called Observer, Publish-Subscribe, or Delegation Event Model?

This idiom was called “**Publish-subscribe**”, and it is still widely known by that name. One object "publishes events," such as the Sale publishing the "property event" when the total changes.

It has been called “**Observer**” because the listener or subscriber is observing the event.

It has also been called the “**Delegation Event Model** “(in Java) because the publisher delegates handling of events to "listeners".

4.5 MAPPING DESIGN TO CODE

Implementation in an object-oriented language requires writing source code for

- class and interface definitions
- method definitions

Implementation is discussed in Java

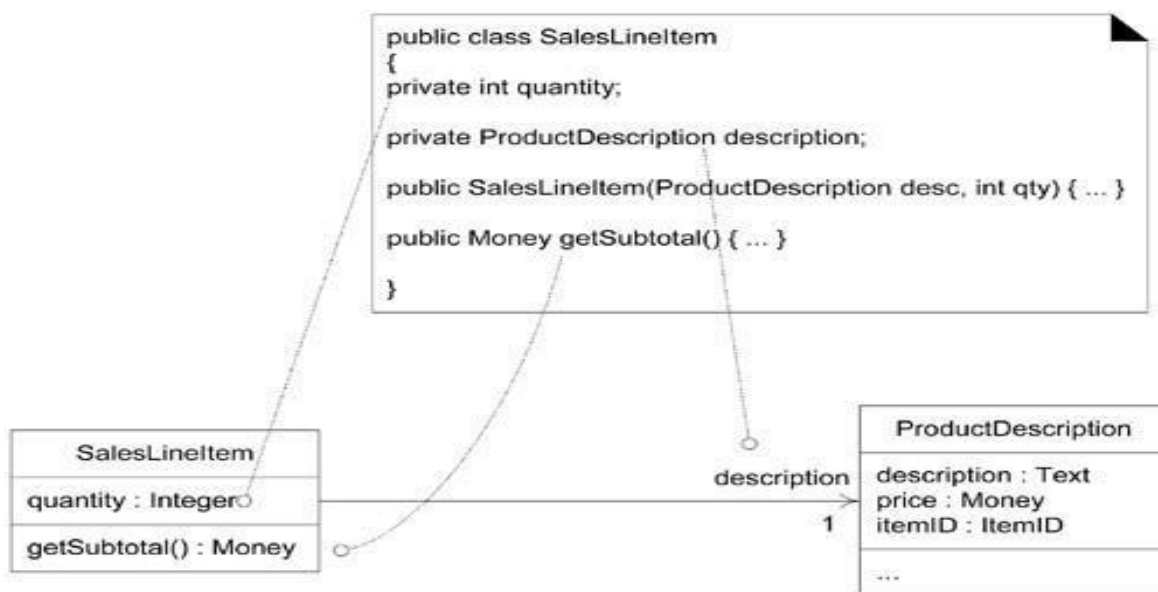
1. Creating Class Definitions from Design Class Diagrams(DCD)
2. Creating Methods from Interaction Diagrams
3. Collection Classes in Code
4. Exceptions and Error Handling
5. Order of Implementation
6. Test-Driven or Test-First Development

1. Creating Class Definitions from DCDs

DCDs depict the class or interface name, superclasses, operation signatures, and attributes of a class. If the DCD was drawn in a UML tool, it can generate the basic class definition from the diagrams.

Defining a Class with Method Signatures and Attributes

From the DCD, a mapping to the attribute definitions (Java fields) and method signatures for the Java definition of `SalesLineItem` is straightforward.



SalesLineItem in Java.

Note :

The addition in the source code of the Java constructor `SalesLineItem(...)`. It is derived from the `create(desc, qty)` message sent to a `SalesLineItem` in the `enterItem` interaction diagram. This indicates, in Java, that a constructor supporting these parameters is required.

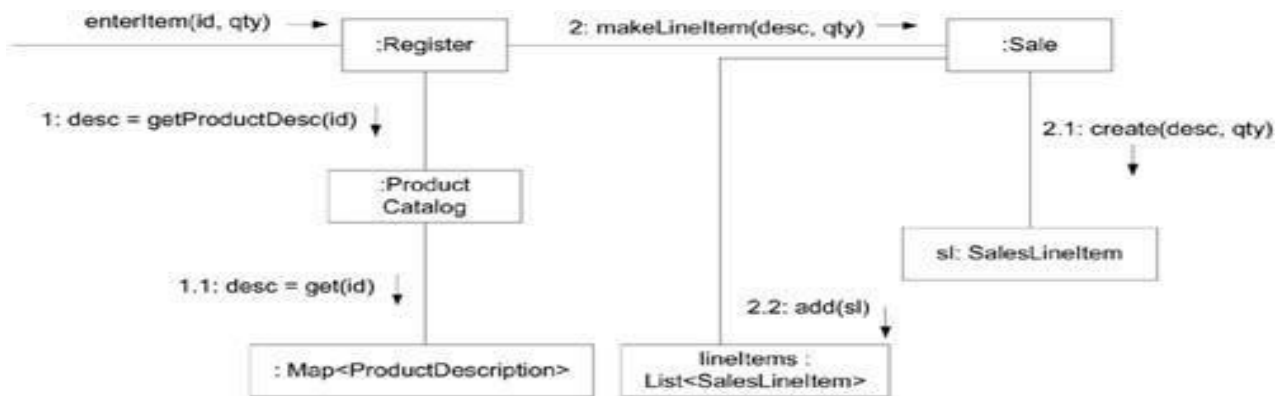
2. Creating Methods from Interaction Diagrams

The sequence of the messages in an interaction diagram translates to a series of statements in the method definitions.

The `enterItem` interaction diagram illustrates the Java definition of the `enterItem` method.

For this example, we will explore the implementation of the `Register` and its `enterItem` method.

A Java definition of the Register class is given below :



The enterItem interaction diagram.

The enterItem message is sent to a Register instance; therefore, the enterItem method is defined in class Register.

```
public void enterItem(ItemID itemID, int qty)
```

Message 1: A getProductDescription message is sent to the ProductCatalog to retrieve a ProductDescription.

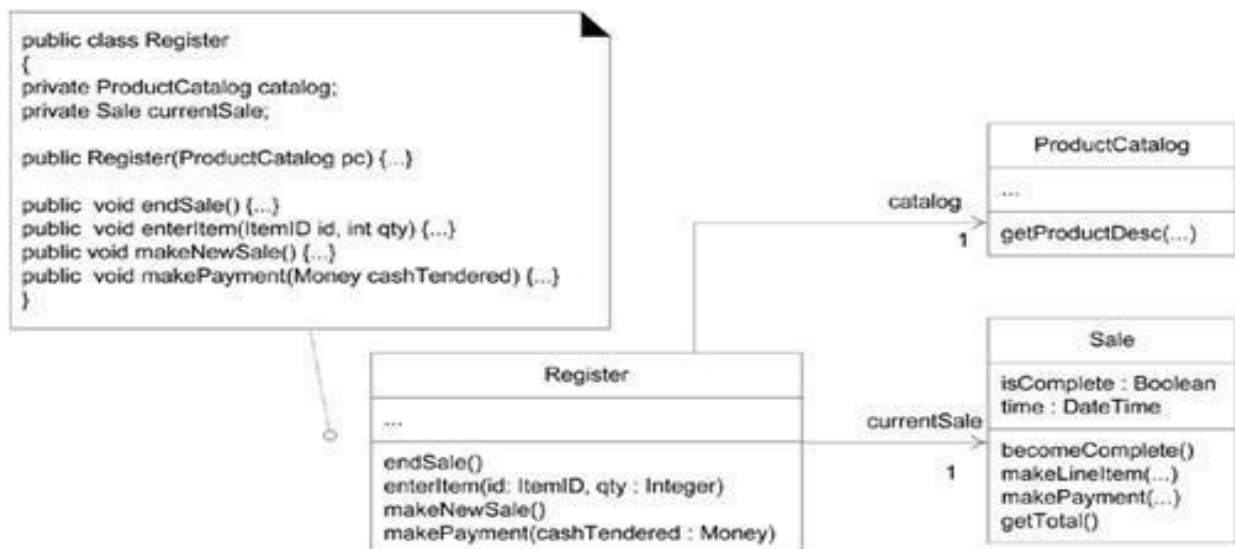
```
ProductDescription desc = catalog.getProductDescription(itemID);
```

Message 2: The makeLineItem message is sent to the Sale.

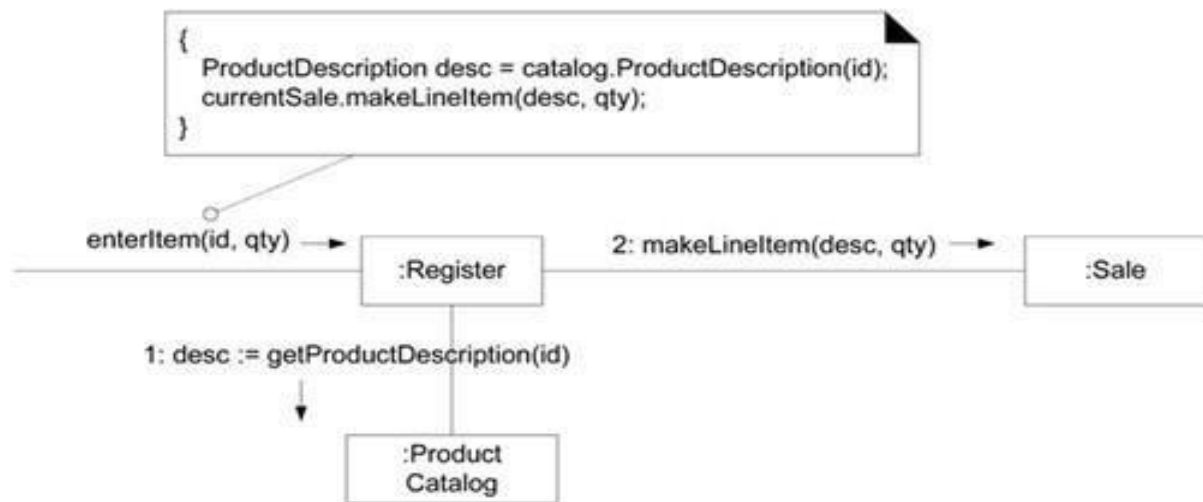
```
currentSale.makeLineItem(desc, qty);
```

The Register class

The Register.enterItem Method



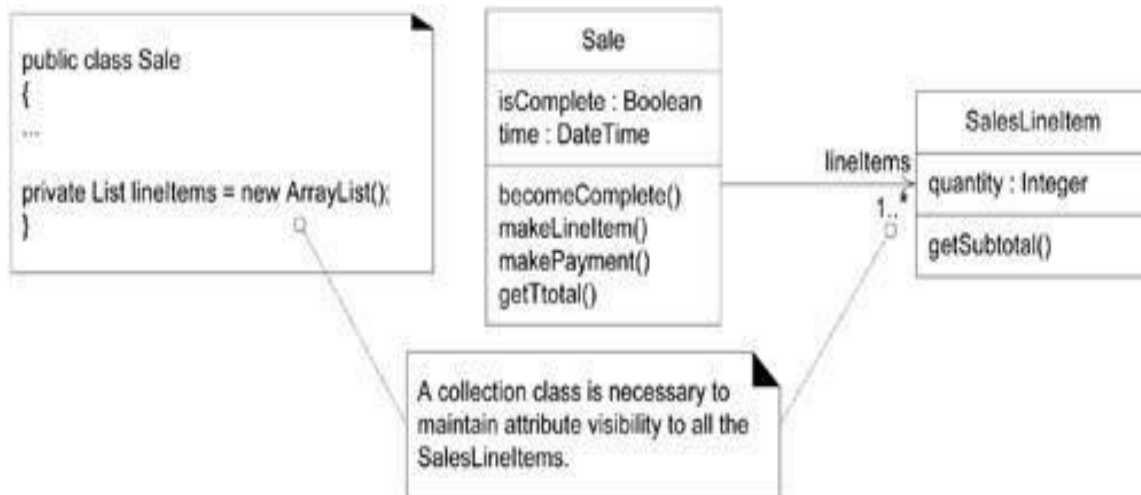
The enterItem method.



3. Collection Classes in Code

One-to-many relationships are common. For example, a `Sale` must maintain visibility to a group of many `SalesLineItem` instances. In OO programming languages, these relationships are usually implemented with the introduction of a collection object, such as a `List` or `Map`, or even a simple array.

Adding a collection.



For example, the Java libraries contain collection classes such as `ArrayList` and `HashMap`, which implement the `List` and `Map` interfaces, respectively. Using `ArrayList`, the `Sale` class can define an attribute that maintains an ordered list of `SalesLineItem` instances.

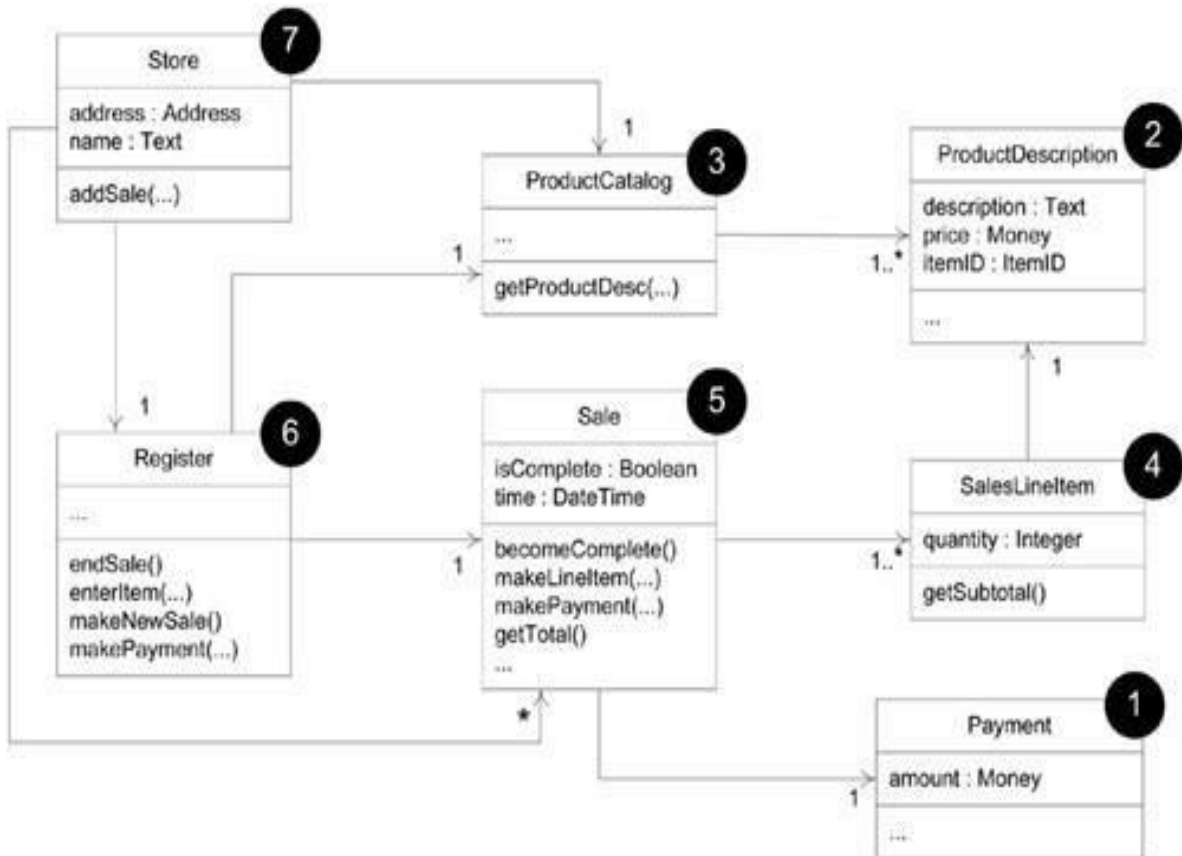
4. Exceptions and Error Handling

In terms of the UML, exceptions can be indicated in the property strings of messages and operation declarations .

5 Order of Implementation

Classes need to be implemented from least-coupled to most-coupled, For example, possible first classes to implement are either Payment or ProductDescription; next are classes only dependent on the prior implementations ProductCatalog or SalesLineItem.

Possible order of class implementation and testing.



6. Test-Driven or Test-First Development

An excellent practice promoted by the iterative and agile XP method and applicable to the UP (as most XP practices are), is test-driven development (TDD). In OO unit testing TDD-style, test code is written before the class to be tested, and the developer writes unit testing code for nearly all production code.

The basic rhythm is

- to write a little test code,
- write a little production code
- make it pass the test
- write some more test code, and so forth.

Example

Suppose if we create TDD for the Sale class. Before programming the Sale class, we write a unit testing method in a SaleTest class that does the following:

Each testing method follows this pattern:

1. Create the fixture.
2. Do something to it (some operation that you want to test).
3. Evaluate that the results are as expected.

Example:

```
public class SaleTest extends TestCase
{
    // ...
    // test the Sale.makeLineItem method

    public void testMakeLineItem()
    {
        // STEP 1: CREATE THE FIXTURE

        // -this is the object to test ,it is an idiom to name it
        'fixture'

        Sale fixture = new Sale();

        // set up supporting objects for the test
        Money total = new Money( 7.5 );
        Money price = new Money( 2.5 );
        ItemID id = new ItemID( 1 );
        ProductDescription desc =
            new ProductDescription( id, price, "product 1" );

        // STEP 2: EXECUTE THE METHOD TO TEST

        // NOTE: We write this code **imagining** there
        // is a makeLineItem method. This act of imagination
        // test makeLineItem

        sale.makeLineItem( desc, 1 );
        sale.makeLineItem( desc, 2 );

        // STEP 3: EVALUATE THE RESULTS

        // there could be many assertTrue statements
        // for a complex evaluation
        // verify the total is 7.5
        assertTrue( sale.getTotal().equals( total ) );
    }
}
```

NextGen POS Program Solution

Class Store

```
public class Store
{
    private ProductCatalog catalog = new ProductCatalog();
    private Register register = new Register( catalog );
    public Register getRegister() { return register; }
}
```

Class ProductDescription

```
public class ProductDescription
{
    private ItemID id;
    private Money price;
    private String description;

    public ProductDescription ( ItemID id, Money price, String
description )
    { this.id = id;
      this.price = price;
      this.description = description;
    }

    public ItemID getItemID() { return id;    }
    public Money getPrice() { return price; }
    public String getDescription() { return description; }
}
```

Class ProductCatalog

```
public class ProductCatalog
{

    public ProductCatalog()
    {
        // sample data
        ItemID id1 = new ItemID( 100 );
        ItemID id2 = new ItemID( 200 );
        Money price = new Money( 3 );

        ProductDescription desc;
        desc = new ProductDescription( id1, price, "product 1" );
        descriptions.put( id1, desc );
        desc = new ProductDescription( id2, price, "product 2" );
        descriptions.put( id2, desc );
    }
    public ProductDescription getProductDescription( ItemID id )
    {
        return descriptions.get( id );
    }
}
```

Class SalesLineItem

```
public class SalesLineItem
{
    private int    quantity;
    private    ProductDescription    description;

    public SalesLineItem (ProductDescription desc, int quantity )
    {
        this.description = desc;
        this.quantity = quantity;
    }
    public Money getSubtotal()
    {
        return description.getPrice().times( quantity );    }
}
```

Class Payment

```
// all classes are probably in a package named
// something like:
package com.foo.nextgen.domain;

public class Payment
{
    private Money amount;

    public Payment( Money cashTendered ){ amount = cashTendered; }
    public Money getAmount() { return amount; }
}
```

Class Register

```
public class Register
{
    private ProductCatalog catalog;
    private Sale currentSale;

    public Register( ProductCatalog catalog )
    {
        this.catalog = catalog;    }

    public void endSale()
    {
        currentSale.becomeComplete();    }

    public void enterItem( ItemID id, int quantity )
    {
        ProductDescription desc = catalog.getProductDescription( id );
        currentSale.makeLineItem( desc, quantity );
    }
    public void makeNewSale()
    {
        currentSale = new Sale();    }

    public void makePayment( Money cashTendered )
    {
        currentSale.makePayment( cashTendered );    }
}
```

Class Sale

```
public class Sale
{
    private List<SalesLineItem> lineItems = new
ArrayList()<SalesLineItem>;
    private Date date = new
Date(); private boolean
isComplete = false;private
Payment payment;
    public Money getBalance()
    {
        return payment.getAmount().minus(

getTotal() );
    }public void bec

    public void makeLineItem ( ProductDescription desc,
intquantity )
    {
        lineItems.add( new SalesLineItem( desc, quantity ) );    }

    public Money getTotal()
    { Money total = new
Money();Money
    subtotal = null;

    for ( SalesLineItem lineItem : lineItems )
    {
        subtotal =
        lineItem.getSubtotal();
        total.add( subtotal );
    }
    return total;
}

    public void makePayment( Money cashTendered )
    {
        payment = new Payment( cashTendered );    }
}
```

UNIT V TESTING

Object Oriented Methodologies – Software Quality Assurance – Impact of object orientation on Testing – Develop Test Cases and Test Plans.

Introduction.

Object oriented systems development is a way to develop software by building self – contained modules or objects that can be easily replaced, modified and reused. In an object–oriented environment, software is a collection of discrete objects that encapsulate their data as well as the functionality of model real–world events “objects” and emphasizes its cooperative philosophy by allocating tasks among the objects of the applications. A class is an object oriented system carefully delineates between its interface (specifications of what the class can do) and the implementation of that interface (how the class does what it does).

A method is an implementation of an object's behavior. A model is an abstract of a system constructed to understand the system prior to building or modifying it. Methodology is going to be a set of methods, models and rules for developing systems based on any set of standards. The process is defined as any operation being performed.

5.1 OBJECT ORIENTED METHODOLOGIES

Object oriented methodologies are set of methods, models, and rules for developing systems. Modeling can be done during any phase of the software life cycle .A model is a an abstraction of a phenomenon for the purpose of understanding the methodologies .Modeling provides means of communicating ideas which is easy to understand the system complexity .

Object-Oriented Methodologies are widely classified into three

- 1.The Rumbaugh et al. OMT (Object modeling technique)
- 2.The Booch methodology
- 3.Jacobson's methodologies

A methodology is explained as the science of methods. A method is a set of procedures in which a specific goal is approached step by step. Too many methodologies have been reviewed earlier stages.

- In 1986, Booch came up with the object-oriented design concept, the Booch method.
- In 1987, Sally Shlaer and Steve Mellor came up with the concept of the recursive design approach.
- In 1989, Beck and Cunningham came up with class-responsibility collaboration (CRC) cards.
- In 1990, Wirfs-Brock, Wilkerson, and Wiener came up with responsibility driven design.
- In 1991, Peter Coad and Ed Yourdon developed the Coad lightweight and prototype-oriented approach. In the same year Jim Rumbaugh led a team at the research labs of General Electric to develop the object modeling technique (OMT).
- In 1994, Ivar Jacobson introduced the concept of the use case.

These methodologies and many other forms of notational language provided system designers and architects many choices but created a much split, competitive and confusing environment. Also same basic concepts appeared in very different notations, which caused confusion among users. Hence, a new evolution of the object oriented technologies which is called as second generation object-oriented methods.

Advantages / Characteristics

- The Rumbaugh et al. method is well-suited for describing the object model or static structure of the system.
- The Jacobson et al. method is good for producing user-driven analysis models
- The Booch method detailed object-oriented design models

Rumbaugh et. al.'s Object Modeling Technique (OMT)

- OMT describes a method for the analysis, design, and implementation of a system using an object-oriented technique.
- Class, attributes, methods, inheritance, and association also can be expressed easily
- The dynamic behavior of objects within a system can be described using OMT
Dynamic model
- Process description and consumer-producer relationships can be expressed using OMT's Functional model
- OMT consists of four phases, which can be performed iteratively:

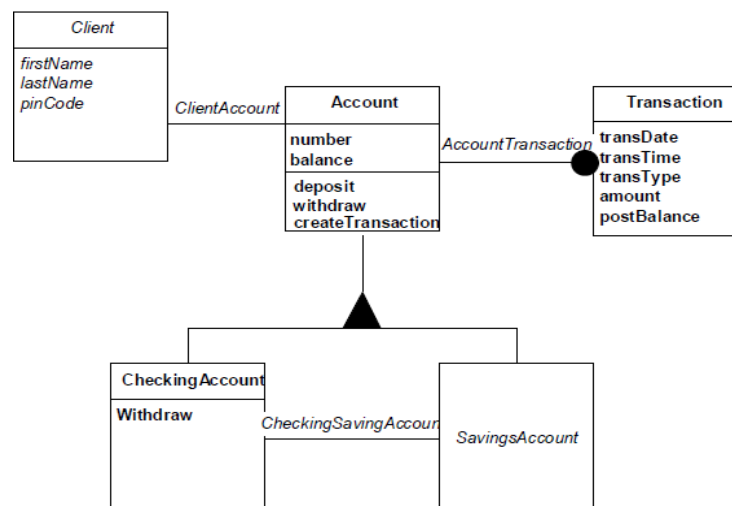
1. Analysis. The results are objects and dynamic and functional models.
2. System design. The result is a structure of the basic architecture of the system.
3. Object design. This phase produces a design document, consisting of detailed objects and dynamic and functional models.
4. Implementation. This activity produces reusable, extendible, and robust code.

- OMT separates modeling into three different parts:

1. An object model, presented by the object model and the data dictionary.
2. A dynamic model, presented by the state diagrams and event flow diagrams.
3. A functional model, presented by data flow and constraints.

OMT Object Model

- The object model describes the structure of objects in a system:
- Their identity, relationships to other objects, attributes, and operations
- The object model is represented graphically with an object diagram
- The object diagram contains classes interconnected by association lines



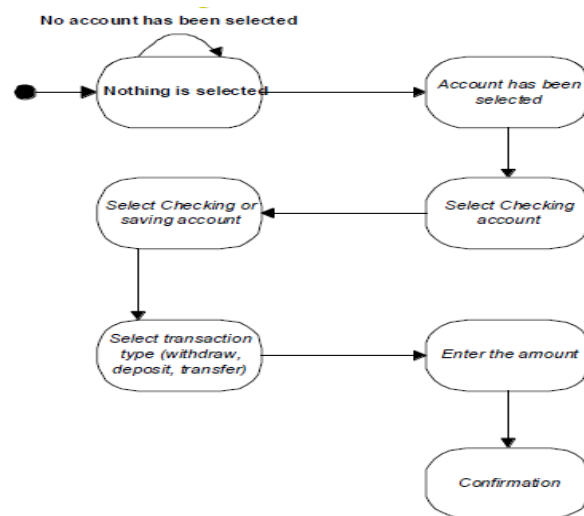
Example of an object model

- The above example provides OMT object model of a bank system. The boxes represent classes and the filled triangle represents specialization.
- Association between **Account** and **Transaction** is one-to-many. Since one account can have many transactions, the filled circle represents many (zero or more).

- The relationship between Client and Account classes is one-to-one. A client can have only one account and account can belong to only one person (in this model joint accounts are not considered)

OMT Dynamic Model

- OMT dynamic model depict states, transitions, events, and actions
- OMT state transition diagram is a network of states and events
- Each state receives one or more events, at which time it makes the transition to the next state.



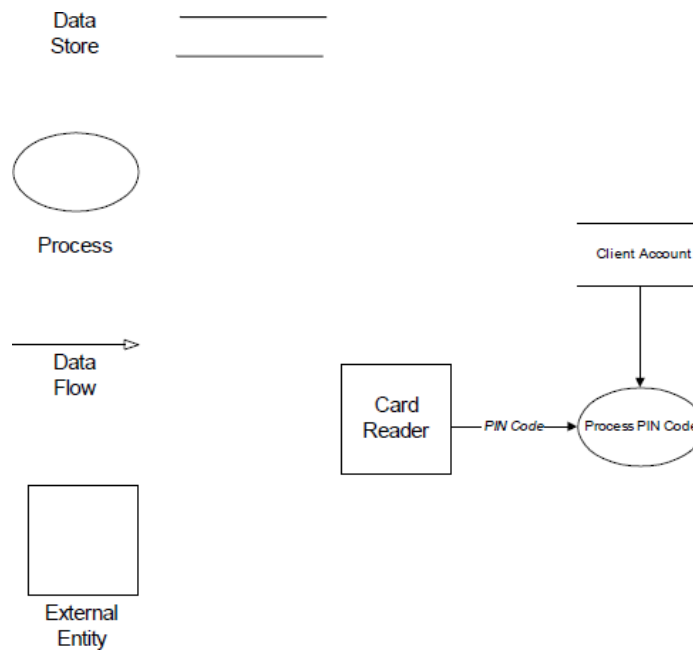
Example of a state transition for ATM Transaction

Here the round boxes represent states and the arrows represent transitions

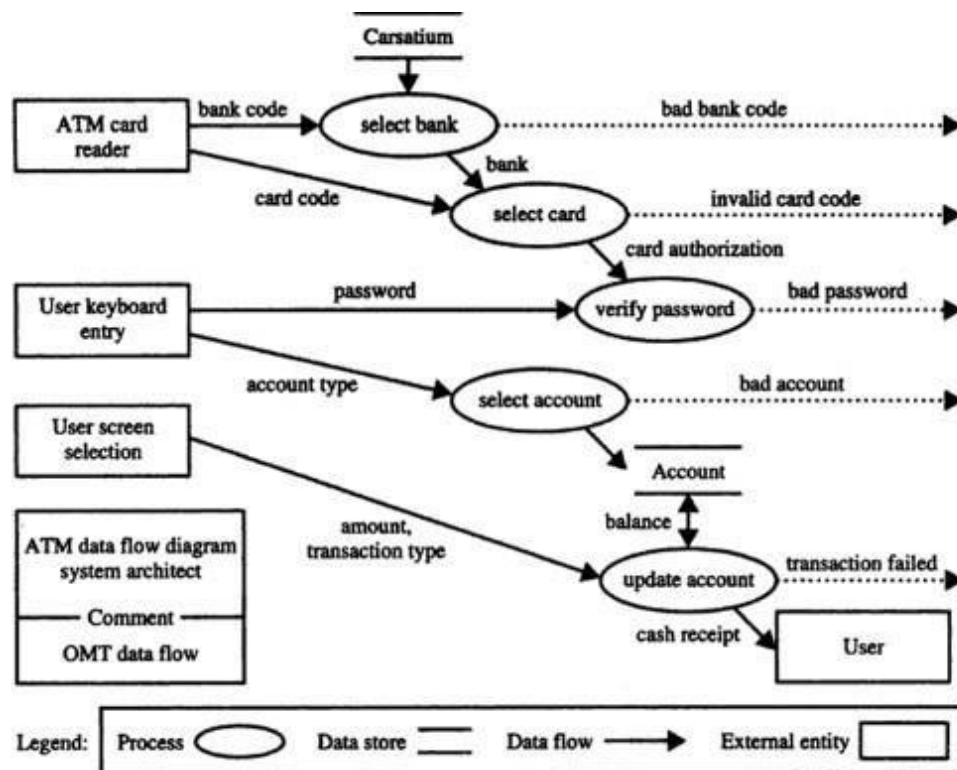
OMT Functional Model

- The OMT DFD shows the flow of data between different process in a business
- DFD use four primary symbols:
 - **Process** is any function being performed ; For Ex, verify password or PIN in the ATM system
 - **Data flow** shows the direction of data element movement: foe Ex. PIN code
 - **Data store** is a location where data are stored: for ex. Account is a data store in the ATM example
 - **External entity** is a source or destination of a data element; fro ex. The ATM card Reader

On the whole , the Rumbaugh et al .OMT methodology provides one of the strongest tool sets for the analysis and design of object-oriented systems .



Example of OMT DFD of an ATM system



The above example is OMT DFD of the ATM system .The data flow lines include arrows to show the direction of data element movement .The circle represents processes. The boxes represents external entities .A data store reveals the storage of data.

The Booch Methodology

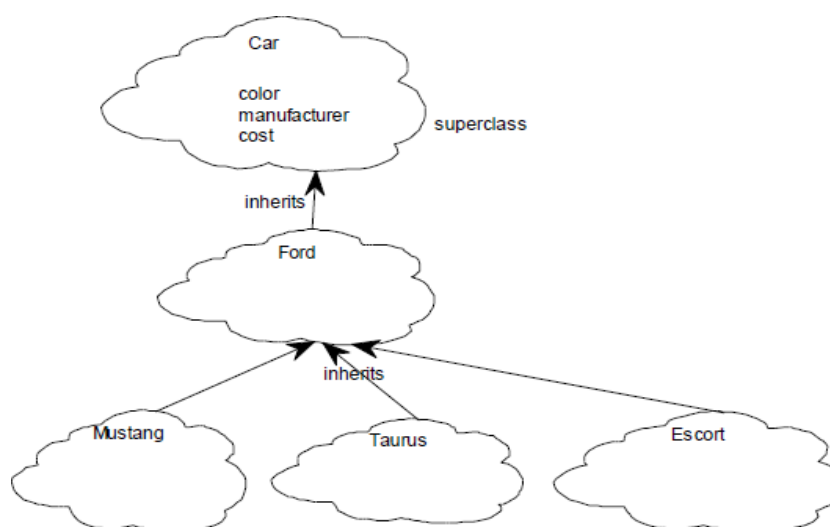
•It is a widely used object oriented method that helps us to design the system using object paradigm.

- The Booch methodology covers the analysis and design phases of systems development.
- Booch sometimes is criticized for his large set of symbols.
- You start with class and object diagram in the analysis phase and refine these diagrams in various steps.

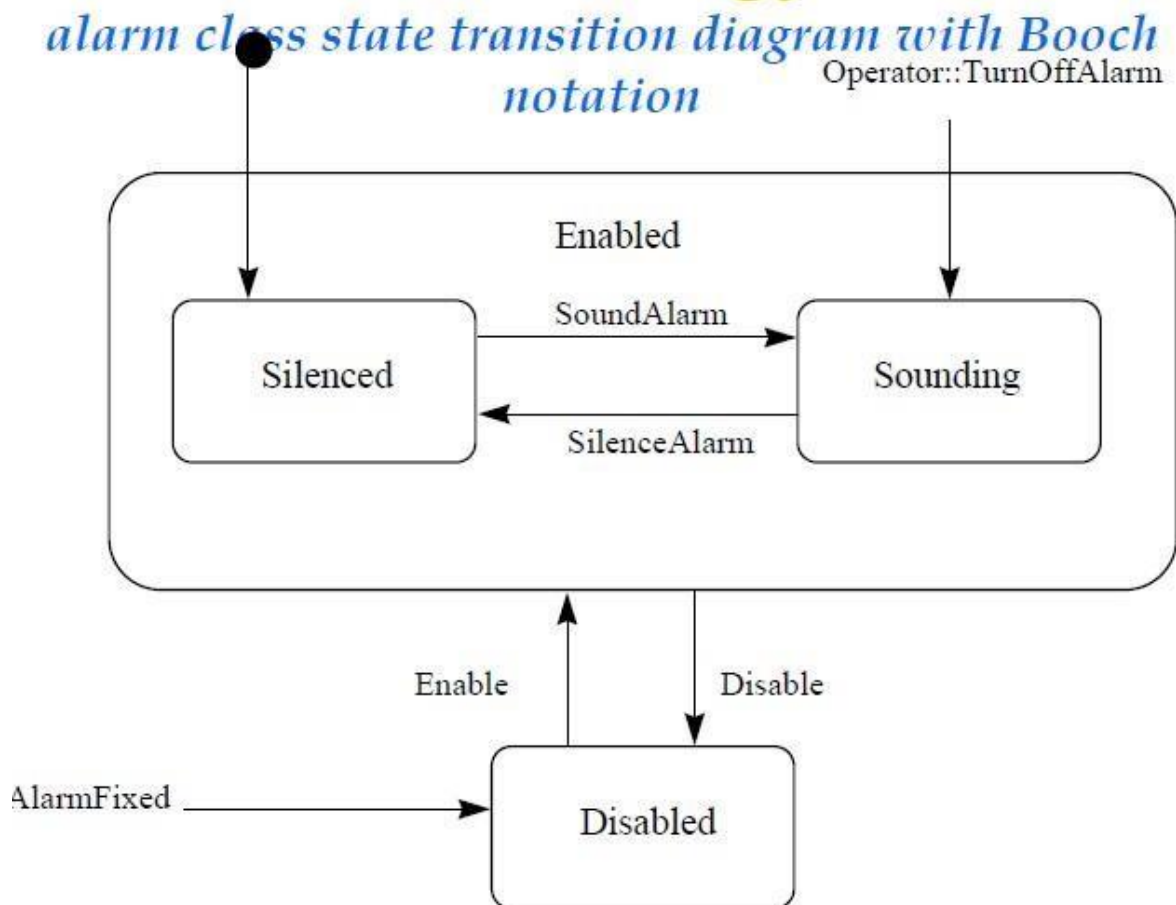
The Booch method consists of the following diagrams:

- Class diagrams
- Object diagrams
- State transition diagrams
- Module diagrams
- Process diagrams
- Interaction diagrams

Object Modeling using Booch Notation



Example : Alarm class state transition diagram with Booch notation. The arrows represents specialization



The Booch methodology prescribes

- A macro development process serve as a controlling framework for the micro process and can take weeks or even months. The primary concern of the macro process is technical management of the system
- A micro development process.

The macro development process consists of the following steps:

1. Conceptualization :

- you establish the core requirements of the system
- You establish a set of goals and develop a prototype to prove the concept

2. Analysis and development of the model.

Use the class diagram to describe the roles and responsibilities objects are to carry out in performing the desired behavior of the system .Also use the Object diagram to

describe the desired behavior of the system in terms of scenarios or use the interaction diagram.

3. Design or create the system architecture.

In this phase, You use the class diagram to decide what class exist and how they relate to each other .Object diagram to used to regulate how objects collaborate. Then use module diagram to map out where each class and object should be declared. Process diagram – determine to which processor to allocate a process.

4. Evolution or implementation. – refine the system through many iterations

5. Maintenance. - make localized changes the the system to add new requirements and eliminate bugs.

Micro Development Process

Each macro development process has its own micro development process

- The micro process is a description of the day to- day activities by a single or small group of s/w developers
- The micro development process consists of the following steps:
 1. Identify classes and objects.
 2. Identify class and object semantics.
 3. Identify class and object relationships.
 4. Identify class and object interfaces and implementation.

The Jacobson et al. Methodologies

- The Jacobson et al. methodologies (e.g., OOBE, OOSE, and Objectory) cover the entire life cycle and stress traceability between the different phases both forward and backward. This traceability enables reuse of analysis and design work, possibly much bigger factors in the reduction of development time than reuse of code.

Use Cases

- Use cases are scenarios for understanding system requirements.
- A use case is an interaction between users and a system.
- The use-case model captures the goal of the user and the responsibility of the system to its users.

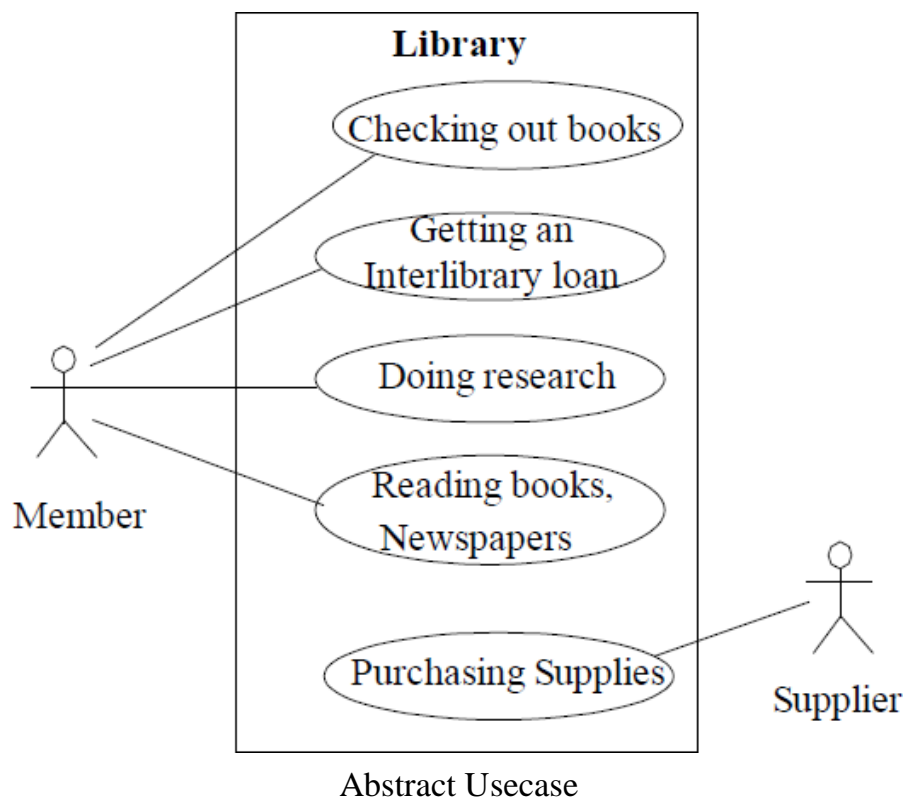
The use case description must contain:

- How and when the use case begins and ends.

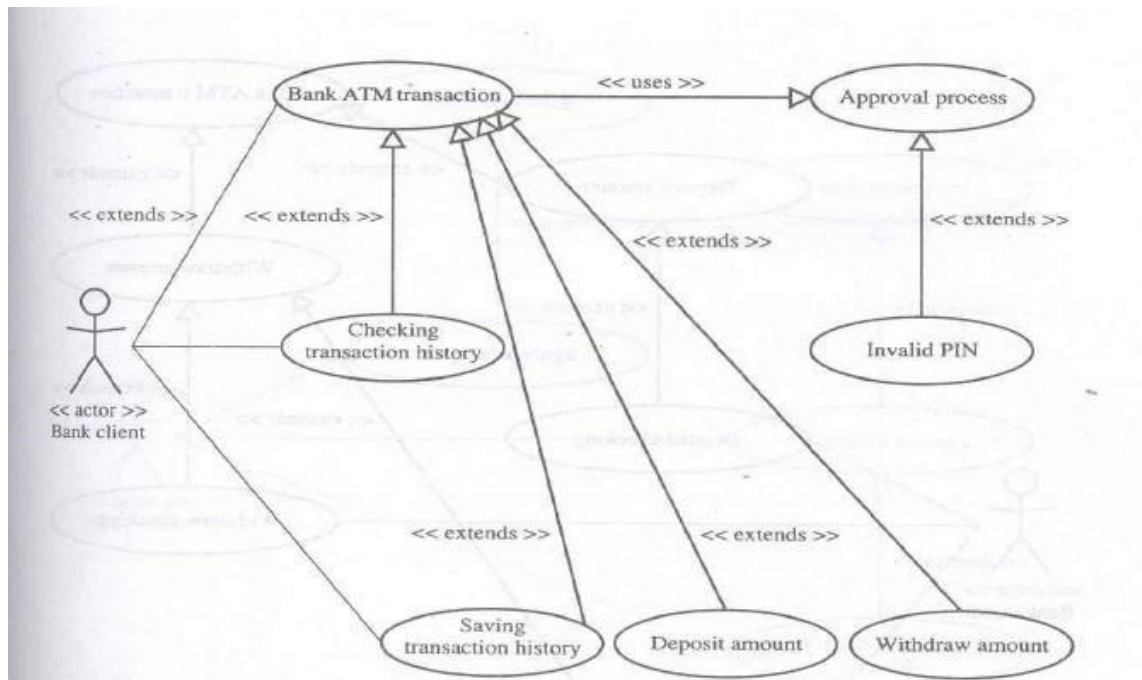
- The interaction between the use case and its actors, including when the interaction occurs and what is exchanged. How and when the use case will store data in the system.
- Exceptions to the flow of events.
 - Every single use case should describe one main flow events
 - An exceptional or additional flow of events could be added
 - The exceptional use case extends another use case to include the additional one
 - The use-case model employs extends and uses relationships
 - The extends relationship is used when you have one use case that is similar to another use case

The uses relationships reuse common behavior in different use cases

- Use cases could be viewed as a concrete or abstract
- Abstract use case is not complete and has no actors that initiate it but is used by another use case.



ATM Transaction use cases.



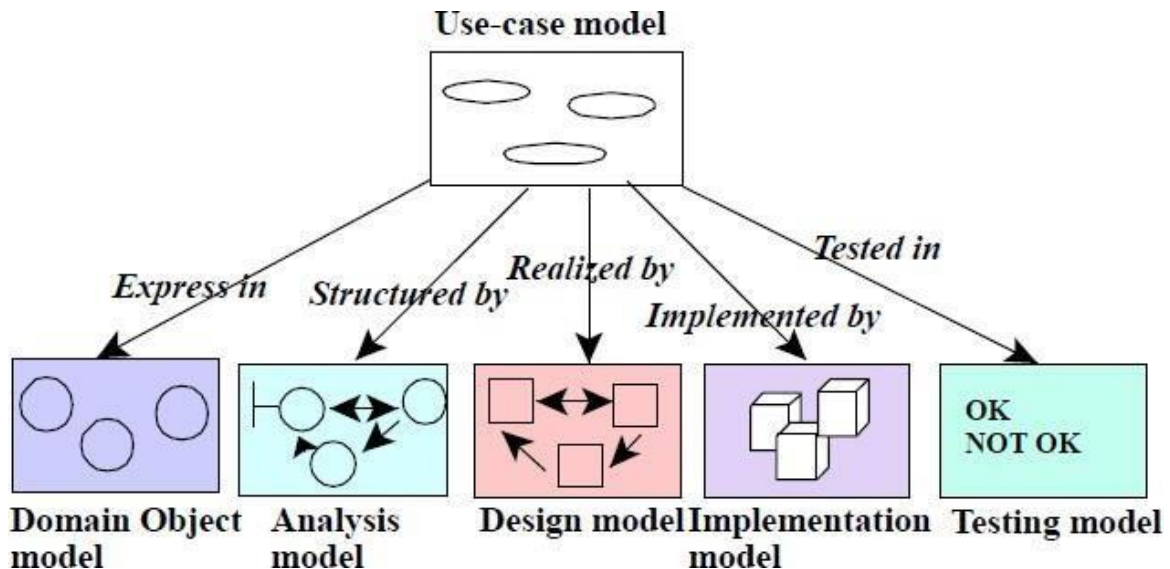
Object-Oriented Software Engineering: Objectory

- Object-oriented software engineering (OOSE), also called Objectory, is a method of object oriented development with the specific aim to fit the development of large, real-time systems. The development process, called use-case driven development, stresses that use cases are involved in several phases of the development.
- The system development method based on OOSE is a disciplined process for the industrialized development of software, based on a use-case driven design. It is an approach to object-oriented analysis and design that centers on understanding the ways in which a system actually is used.
- By organizing the analysis and design models around sequences of user interaction and actual usage scenarios, the method produces systems that are both more usable and more robust, adapting more easily to changing usage
- The maintenance of each model is specified in its associated process. A process is created when the first development project starts and is terminated when the developed system is taken out of service

Objectory is built around several different models:

- Use case model.
- defines the outside (actors) and inside(use case) of the system behavior

- Domain object model. The object of the “real” world are mapped into the domain object model
- Analysis object model.
- how the source code (implementation) should be carried out and written
- Implementation model.
- represents the implementation of the system
- Test model.
 - constitute the test plan, specifications, and reports



Object-Oriented Business Engineering (OOBE)

Object-oriented business engineering (OOBE) is object modeling at the enterprise level. Use cases again are the central vehicle for modeling, providing traceability throughout the software engineering processes.

OOBE consists of : object modeling at enterprises level

– Analysis phase

- The analysis phase defines the system to be built in terms of the problem-domain object model, the requirements model and the analysis model .This reduces complexity and promotes maintainability over the life of the system ,since the description of the system will be independent of hardware and software requirements .
- The analysis process is iterative but the requirements and the analysis models should be stable before moving on to subsequent models. Jacobson et al.

suggest that prototyping with a tool might be useful during this phase to help specify user interfaces.

– Design & Implementation phases

- The implementation environment must be identified for the design model . This include factors such as DBMS, distribution of process ,constraints due to the programming language, available component libraries and incorporation user interface tools
- It may be possible to identify implementation environment concurrently with analysis. The analysis objects that fit the current implementation environment.

– Testing phase.

Finally Jacobson describes several testing levels and techniques such as unit testing, integration testing and system testing.

Patterns

A design pattern is defined as that it identifies the key aspects of a common design sturture that make it useful for creating a reusable object-orinted design . It also identifies the participating classes and instances their roles and collaborations and the distribution of responsibilities.[Gamma,Helson,Johnson definition]

A pattern involves a general description of solution to a recurring problem bundle with various goals and constraints. But a pattern does more than just identify a solution; it also explains why the solution is needed.

- A pattern is useful information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces.
- Its help software developers resolve commonly encountered, difficult problems and a vocabulary for communicating insight and experience about these problems and their solutions.

The main idea behind using patterns is to provide documentation to help categorize and communicate about solutions to recurring problems.

- The pattern has a name to facilitate discussion and the information it represents.

A good pattern will do the following:

- It solves a problem.

Patterns capture solutions, not just abstract principles or strategies.

- It is a proven concept.

Patterns capture solutions with a track record, not theories or speculation.

- The solution is not obvious.

The best patterns generate a solution to a problem indirectly—a necessary approach for the most difficult problems of design.

- It describes a relationship.

Patterns do not just describe modules, but describe deeper system structures and mechanisms.

Generative and Non-Generative Patterns

- Generative patterns are the patterns that not only describe a recurring problem but also tell us how to generate something and can be observed in the resulting system architectures.
- Non-generative patterns are static and passive .They describe recurring phenomena without necessarily saying how to reproduce them.

Patterns Template

Every pattern must be expressed in form of a template which establishes a relationship between a context , a system of forces which raises in that context and a configuration which allows these forces to resolve themselves in that context. The following components should be present in a pattern template

- Name –A meaningful name .This allows us to use a single word or short phrase to refer a pattern and the knowledge and the structure it describes.Sometimes a pattern may have more than one commonly used or recognizable name in the literature .In this case nick names can be used .
- Problem-A statement of a problem that describes its intent: the goals and objectives it wants to reach within the given context and forces .
- Context-The preconditions under which the problem and its solution seem to recur and for which solution is desirable. This tells us about the pattern applicability.
- Forces-A description of the relevant forces and constraints and how they interact or conflict with one another and with goals to that wish to achieve. Forces reveal the intricacies of the problem and define the kinds of trade-offs that must be considered in the presences of the tension or dissonance they create. A good pattern description should fully encapsulate all the forces that have an impact on it.
- Solution

- **Solution.** Static relationships and dynamic rules describing how to realize the desired outcome. This often is equivalent to giving instructions that describe how to construct the necessary products. The description may encompass pictures, diagrams, and prose that identify the pattern's structure, its participants, and their collaborations, to show how the problem is solved. The solution should describe not only the *static structure* but also *dynamic behavior*. The static structure tells us the form and organization of the pattern, but often the behavioral dynamics is what makes the pattern "come alive." The description of the pattern's solution may indicate guidelines to keep in mind (as well as pitfalls to avoid) when attempting a concrete implementation of the solution. Sometimes, possible variants or specializations of the solution are described as well.
- Examples
 - **Examples.** One or more sample applications of the pattern that illustrate a specific initial context; how the pattern is applied to and transforms that context; and the resulting context left in its wake. Examples help the reader understand the pattern's use and applicability. Visual examples and analogies often can be very useful. An example may be supplemented by a *sample implementation* to show one way the solution might be realized. Easy-to-comprehend examples from known systems usually are preferred.
- Resulting context
 - **Resulting context.** The state or configuration of the system after the pattern has been applied, including the consequences (both good and bad) of applying the pattern, and other problems and patterns that may arise from the new context. It describes the *postconditions* and *side effects* of the pattern. This is sometimes called a *resolution of forces* because it describes which forces have been resolved, which ones remain unresolved, and which patterns may now be applicable. Documenting the resulting context produced by one pattern helps you correlate it with the initial context of other patterns (a single pattern often is just one step toward accomplishing some larger task or project).
- Rationale
 - **Rationale.** A justifying explanation of steps or rules in the pattern and also of the pattern as a whole in terms of how and why it resolves its forces in a particular way to be in alignment with desired goals, principles, and philosophies. It explains how the forces and constraints are orchestrated in concert to achieve a resonant harmony. This tells us how the pattern actually works, why it works, and why it is "good." The solution component of a pattern may describe the outwardly visible structure and behavior of the pattern, but the rationale is what provides insight into the *deep structures* and *key mechanisms* going on beneath the surface of the system.
- Related Patterns
 - **Related patterns.** The static and dynamic relationships between this pattern and others within the same pattern language or system. Related patterns often share common forces. They also frequently have an initial or resulting context that is compatible with the resulting or initial context of another pattern. Such patterns might be predecessor patterns whose application leads to this pattern, successor patterns whose application follows from this pattern, alternative patterns that describe a different solution to the same problem but under different forces and constraints, and codependent patterns that may (or must) be applied simultaneously with this pattern.

- Known uses-The known occurrences of the pattern and its application within existing systems .This helps validate a pattern by verifying that it indeed is a proven solution to a recurring problem .

AntiPatterns

- A pattern represents a “best practice” whereas an antipattern represents “worst practice” or a “lesson learned”
- Antipattern come in two varieties:
 - Those describe a bad solution to a problem that resulted in a bad situation
 - Those describing how to get out of a bad situation and how to proceed from there to a good solution
- The pattern has a significant human component.
 - All software serves human comfort or quality of life.
 - The best patterns explicitly appeal to aesthetics and utility.

Capturing Patterns

- Patterns should provide not only facts but also tell us a story that captures the experience they are trying to convey.
- A pattern should help its users comprehend existing systems, customize systems to fit user needs, and construct new systems.
- The process of looking for patterns to document is called pattern mining.
- Guidelines for capturing patterns:
 - Focus on practicability.-Patterns should describe proven solutions to recurring problems rather than the latest scientific results .
 - Aggressive disregard of originality.-Pattern writers do not need to be the original inventor or discoverer of the solutions that they document.
 - Non-anonymous review.-Paper submissions are shepherd rather than reviewed. It contacts the pattern authors and discusses with him or her how the patterns might be clarified or improved on
 - Writers' workshops instead of presentations.-Open forums are used here to improve the patterns which are lacking
 - Careful editing
- .-Incorporating all the review comments and insights given by the writers workshops.

Frameworks

- A framework is a way of presenting a generic solution to a problem that can be applied to all levels in a development.

- A single framework typically encompasses several design patterns and can be viewed as the implementation of a system of design patterns.

A definition of object oriented software framework is given by Gamma et al.

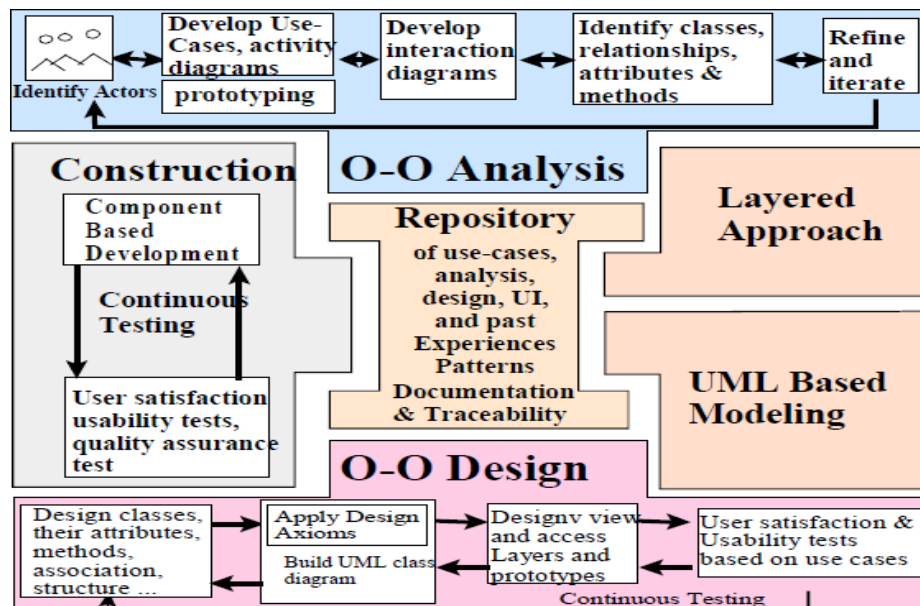
A framework is a set of cooperating classes that make up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes a framework to a particular application by subclassing and composing instances of framework classes. The framework captures the design decisions that are common to its application domain. Frameworks thus emphasize design reuse over code reuse, though a framework will usually include concrete subclasses you can put to work immediately.

Differences between Design Patterns and Frameworks

- Design patterns are more abstract than frameworks.
- Design patterns are smaller architectural elements than frameworks.
- Design patterns are less specialized than frameworks.

The Unified Approach

- The idea behind the UA is not to introduce yet another methodology.
- The main motivation here is to combine the best practices, processes, methodologies, and guidelines along with UML notations and diagrams.



The unified approach to software development revolves around (but is not limited to) the following processes and components.

The processes are:

- Use-case driven development.
- Object-oriented analysis.
- Object-oriented design.
- Incremental development and prototyping.
- Continuous testing.

UA Methods and Technology

- The methods and technology employed includes:
 - Unified modeling language (UML) used for modeling.
 - Layered approach.
 - Repository for object-oriented system development patterns and frameworks.
 - Promoting Component-based development.

UA Object-Oriented Analysis:

Use-Case Driven

- The use-case model captures the user requirements.
- The objects found during analysis lead us to model the classes.
- The interaction between objects provide a map for the design phase to model the relationships and designing classes.

OOA Process consists of the following steps :

1. Identify the Actors
2. Develop the simple business process model using UML activity diagram
3. Develop the Use Case
4. Develop interaction diagrams
5. Identify classes

UA Object-Oriented Design:

- Booch provides the most comprehensive object-oriented design method.
- However, Booch methods can be somewhat imposing to learn and especially tricky to figure out where to start.
- UA realizes this by combining Jacobson et al.'s analysis with Booch's design concept to create a comprehensive design process.

OOD Process consists of:

- Design classes , their attributes, methods, associations, structures and protocols, apply design axioms
- Design the Access Layer
- Design and prototype User Interface
- User satisfaction and usability Test based on the usage / Use cases

Iterative Development and Continuous Testing

- The UA encourages the integration of testing plans from day 1 of the project.
- Usage scenarios or Use Cases can become test scenarios; therefore, use cases will drive the usability testing.
- You must iterate and reiterate until, you are satisfied with the system.

Modeling Based on the Unified Modeling Language

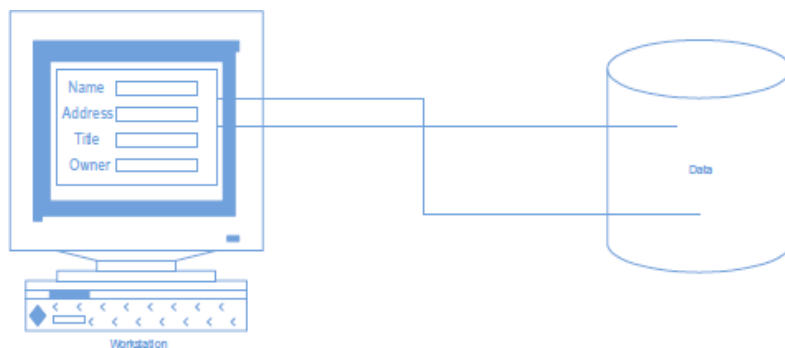
- The UA uses the unified modeling language (UML) to describe and model the analysis and design phases of system development.

The UA Proposed Repository

- The requirement, analysis, design, and implementation documents should be stored in the repository, so reports can be run on them for traceability.
- This allows us to produce designs that are traceable across requirements, analysis, design, implementation, and testing.

Two-Layer Architecture

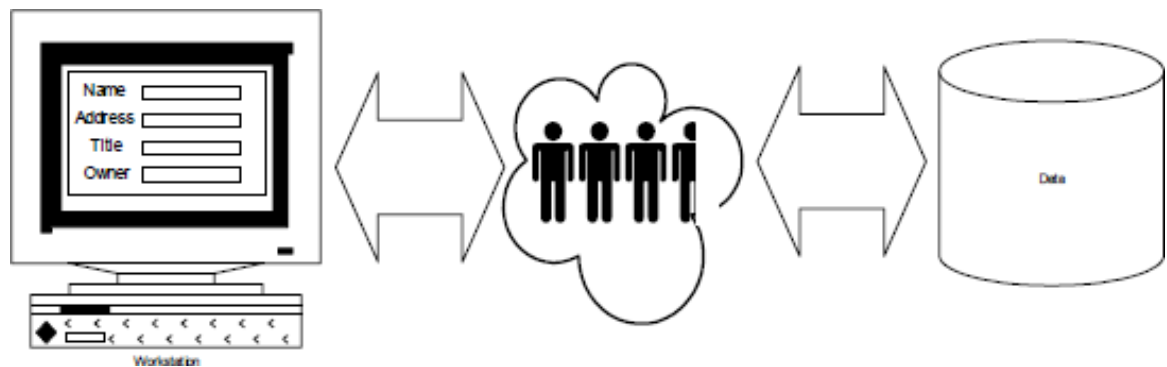
In a two-layer system, user interface screens are tied directly to the data through routines that sit directly behind the screens.



This approach results in objects that are very specialized and cannot be reused easily in other projects.

Three-Layer Architecture

- Your objects are completely independent of how:
 - they are represented to the user (through an interface) or
 - how they are physically stored.



User Interface layer

This layer consists of objects with which the user interacts as well as the objects needed to manage or control the interface. It is also called as a view layer. The UI interface layer objects are identified during OOD phase .

This layer is typically responsible for two major aspects of the applications:

- Responding to user interaction-Here the user interface layer objects must be designed to translate actions by the user , such as clicking on a button or selecting from a menu ,into an appropriate response .

That response may be to open or close another interface or to send a message down into the business layer to start some business process.

- Displaying business objects.-The display of the objects is shown by using list boxes and graphs

Business Layer

- 1.The responsibilities of the business layer are very straightforward:
- 2.model the objects of the business and how they interact to accomplish the business processes.

Business Layer: Real Objects

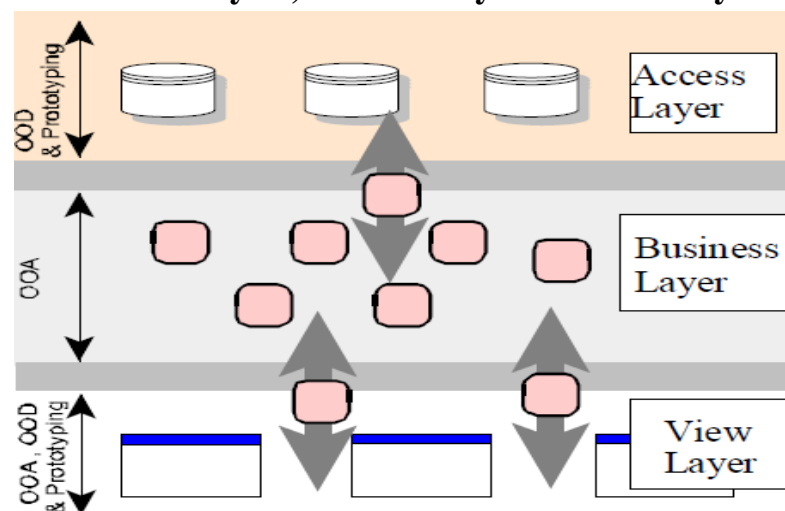
These objects should not be responsible for:

- **Displaying details.** Business objects should have no special knowledge of how they are being displayed and by whom. They are designed to be independent of any particular interface, so the details of how to display an object should exist in the interface (view) layer of the object displaying it.
- **Data access details.** Business objects also should have no special knowledge of “where they come from.” It does not matter to the business model whether the data are stored and retrieved via SQL or file I/O. The business objects need to know only to whom to talk about being stored or retrieved. The business objects are modeled during the object-oriented analysis.

Access Layer

- The access layer contains objects that know how to communicate with the place where the data actually resides,
- Whether it be a relational database, mainframe, Internet, or file.
- The access layer has two major responsibilities:
- Translate request-This layer must be able to translate any data-related requests from the business layer into the appropriate protocol for data access.(For eg . if a customer number 5333 is to be retrieved from the Database , an SQL statement is created by the access layer and execute it)
- Translate result –It translates the data retrieved back into the appropriate business objects and passes those objects back up into the business layer

Architecture for Access layer ,Business layer and view layer



5.2 SOFTWARE QUALITY ASSURANCE

The major key areas of SQA are

- Bugs and Debugging
- Testing strategies.
- The impact of an object orientation on testing.
- How to develop test cases.
- How to develop test plans.

Two issues in software quality are:

- Validation or user satisfaction
- Verification or quality assurance.

Elimination of the syntactical bug is the process of debugging. Detection and elimination of the logical bug is the process of testing.

Error Types:

- Language errors or syntax errors
- Run-time errors
- Logic errors

Identifying Bugs and Debugging

- The first step in debugging is recognizing that a bug exists.
- Sometimes it's obvious; the first time you run the application, it shows itself.
- Other bugs might not surface until a method receives a certain value, or until you take a closer look at the output

However, these steps might help:

- Selecting appropriate testing strategies
- Developing test cases and sound test plan.

Debugging Tools

- Debugging tools are a way of looking inside the program to help us determine what happens and why.
- It basically gives us a snapshot of the current state of the program.

Testing Strategies

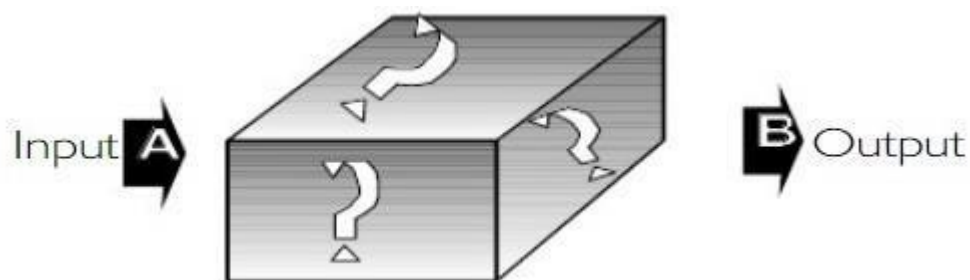
There are four types of testing strategies, These are:

- ✓ Black Box Testing
- ✓ White Box Testing
- ✓ Top-down Testing

✓ Bottom-up Testing

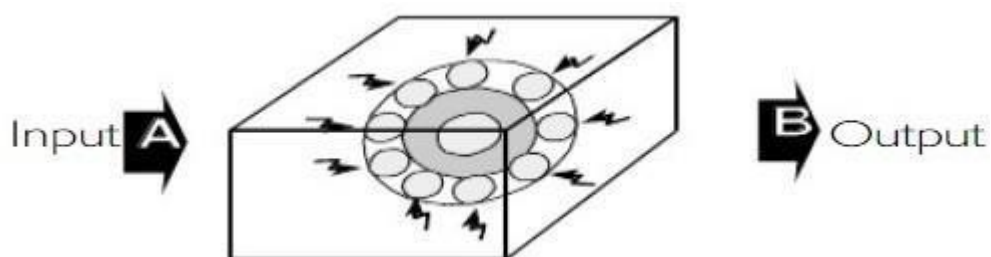
Black Box Testing

- In a black box, the test item is treated as "black" whose logic is unknown.
- All that's known is what goes in and what comes out, the input and output
- Black box test works very nicely in testing objects in an Object-Oriented environment.



White Box Testing

White box testing assumes that specific logic is important, and must be tested to guarantee system's proper functioning. This testing looks for bugs that have a low probability of execution that has been overlooked in previous investigations. The main use of this testing is error-based testing, when all level based objects are tested carefully.



One form of white box testing is called path testing

- It makes certain that each path in a program is executed at least once during testing.

Two types of path testing are:

- Statement testing coverage- The main idea of the statement testing coverage is test every statement in the objects method executing it at least once.

- Branch testing coverage –The main idea here is to perform enough tests to ensure that every branch alternative has been executed at least once under some test. It is feasible to fully test any program of considerable size.

Top-down Testing

It assumes that the main logic of the application needs more testing than supporting logic.

Bottom-up Approach

- It takes an opposite approach.
- It assumes that individual programs and modules are fully developed as standalone processes.
- These modules are tested individually, and then combined for integration testing.

System Usability & Measuring User Satisfaction

- Verification
- "Am I building the product right?"

Validation

- "Am I building the right product?"

Two main issues in software quality are
Validation or user satisfaction and
verification or quality assurance.

- The process of designing view layer classes consists of the following steps:
 1. In the macro-level user interface (UI) design process, identify view layer objects.
 2. In the micro-level UI, apply design rules and GUI guidelines.
 3. Test usability and user satisfaction.
 4. Refine and iterate the design.

Usability and User Satisfaction Testing

Two issues will be discussed:

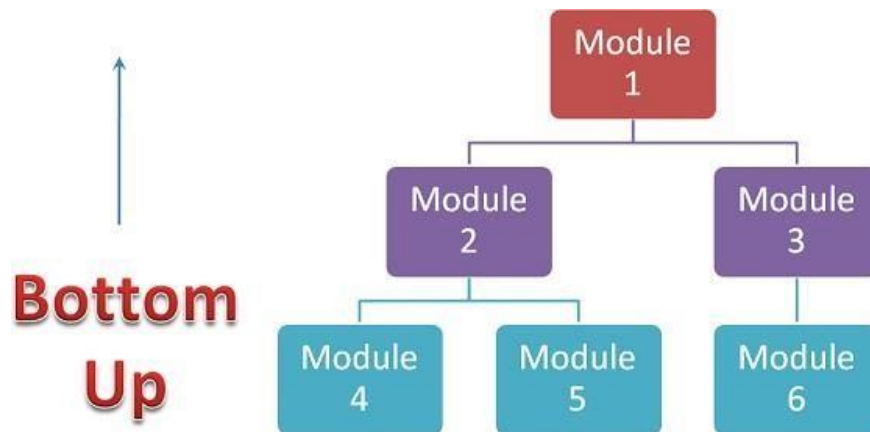
1. Usability Testing and how to develop a plan for usability testing.
 2. User Satisfaction Test and guidelines for developing a plan for user satisfaction testing.
- The International Organization for Standardization (ISO) defines usability as the effectiveness, efficiency, and satisfaction with which a specified set of users can achieve a specified set of tasks in particular environments.
 - Defining tasks. What are the tasks?

- Defining users. Who are the users?
- A means for measuring effectiveness, efficiency, and satisfaction

The phrase two sides of the same coin is helpful for describing the relationship between the Usability and functionality of a system.

Bottom – Up Testing

It supports testing user interface and system integration. In the bottom-up strategy, each module at lower levels is tested with higher modules until all modules are tested. It takes help of Drivers for testing



Advantages:

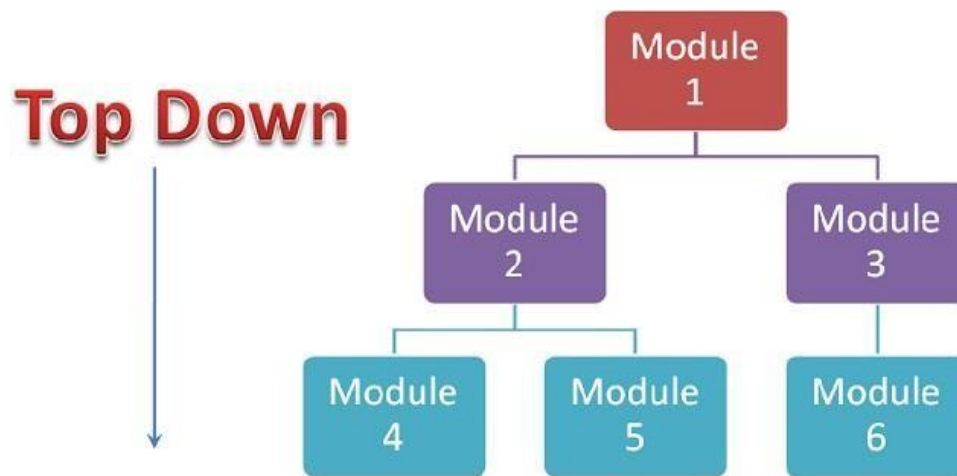
- Fault localization is easier.
- No time is wasted waiting for all modules to be developed unlike Big-bang approach

Disadvantages:

- Critical modules (at the top level of software architecture) which control the flow of application are tested last and may be prone to defects.
- An early prototype is not possible

Top-down Testing:

In Top to down approach, testing takes place from top to down following the control flow of the software system. Takes help of stubs for testing. It starts with the details of the system and proceeds to higher levels by a progressive aggregation of details until they fit requirements of system.



Advantages:

- Fault Localization is easier.
- Possibility to obtain an early prototype.
- Critical Modules are tested on priority; major design flaws could be found and fixed first.

Disadvantages:

- Needs many Stubs.
- Modules at a lower level are tested inadequately.

5.3 IMPACT OF OBJECT ORIENTATION ON

TESTING Errors.

- Less Plausible (not worth testing for)
- More Plausible (worth testing for now)
- New types of errors may appear

Impact of Inheritance on Testing.

- Does not reduce the volume of test cases
- Rather, number of interactions to be verified goes up at each level of the hierarchy
- Testing approach is essentially the same for OO oriented and Non-Object oriented environment.
- However, can reuse superclass/base class test cases
- Since OO methods are generally smaller, these are easier to test . But there are more opportunities for integration faults.

Reusability of tests.

Reusable Test Cases and Test Steps is a tool to improve re-usability and maintainability of Test Management by reducing redundancy between Test Cases in projects. Often the Test scenarios require that some Test Cases and Test Steps contain repeated or similar actions performed during a Testing cycle.

The models used for analysis and design should be used for testing at the same time. The class diagram describes relationship between objects .which is a useful information form testing .Also it shows the inheritance structure which is important information for error-based testing.

Error based testing

Error based testing techniques search a given class's method for particular clues of interests, and then describe how these clues should be tested.

Usability testing

Measures the ease of use as well as the degree of comfort and satisfaction users have with the software.

- Usability testing must begin with defining the target audience and test goals.
- Run a pilot test to work out the bugs of the tasks to be tested.
- Make certain the task scenarios, prototype, and test equipment work smoothly.

Guidelines for Developing Usability Testing

—Focus groups" are helpful for generating initial ideas or trying out new ideas. It requires a moderator who directs the discussion about aspects of a task or design

- Apply usability testing early and often.
- Include all of software's components in the test.
- The testing doesn't need to be very expensive, a tape recorder, stopwatch, notepad and an office can produce excellent results.
- Tests need not involve many subjects.
- More typically, quick, iterative tests with a small, well-targeted sample of 6 to 10 participants can identify 80– 90 percent of most design problems.
- Focus on tasks, not features.
- Remember that your customers will use features within the context of particular tasks.
- Make participants feel comfortable by explaining the testing process.
- Emphasize that you are testing the software, not the participants.

- If they become confused or frustrated, it is not a reflection on them.
- Do not interrupt participants during a test.
- If they need help, begin with general hints before moving to specific advice.
- Keep in mind that less intervention usually yields better results.
- Record the test results using a portable tape recorder, or better, a video camera.
- You may also want to follow up the session with the user satisfaction test.
- The test is inexpensive, easy to use and it is educational to those who administer it and those who fill it out. Even if the results may never be summarized, or filled out, the process of creating the test itself will provide us with useful information.

5.4 TEST CASES

A test case is a set of What – if questions. To test a system you must construct some best input cases, that describe how the output will look. Next, perform the tests and compare the outcome with the expected output.

Myer's (objective of testing)

Testing is a process of executing a program with the intent of finding errors.
 Good test case. That has a high probability of finding an as – yet – undiscovered error.
 Successful test case That detects an as – yet – undiscovered error.

Specifying results is crucial in developing test cases. You should test cases that are supposed to fail. During such tests, it is a good idea to alert the person running them that failure is expected. Say, we are testing a File Open feature. We need to specify the result as follows:

- 1. Drop down the File menu and select Open.**
- 2. Try opening the following types of files:**
 - A file that is there (should work).
 - A file that is not there (should get an *Error* message).
 - A file name with international characters (should work).
 - A file type that the program does not open (should get a message or conversion dialog box).

Guidelines for Developing quality assurance test cases.

Freedman and Thomas have developed guidelines that have been adopted for the UA:

- Describe which feature or service your test attempts to cover.
- If the test case is based on a use case, it is good idea to refer to the use-case name.
- Specify what you are testing and which particular feature.
- test the normal use of the object methods.

- test the abnormal but reasonable use of the objects methods.
- test the boundary conditions.
- Test objects interactions and the messages sent among them.
- Attempting to reach agreement on answers generally will raise other what-if questions.
- The internal quality of the software, such as its reusability and extensibility, should be assessed as well.

5.5 TEST PLAN

* A Test plan is developed to detect and identify potential problems before delivering the software to its users.

* A test plan offers a road map.

* A dreaded and frequently overlooked activity in software development.

Steps

- Objectives of the test.- create the objectives and describes how to achieve them
- Development of a test case- develop test case, both input and expected output.
- Test analysis.- This step involves the examination of the test output and the documentations of the test results

Regression Testing.- All passed tests should be repeated with the revised program, called "Regression". This can discover errors introduced during the debugging process. When sufficient testing is believed to have been conducted, this fact should be reported, and testing to this specific product is complete

Beta Testing.

Beta Testing can be defined as the second stage of testing any product before release where a sample of the released product with minimum features and characteristics is being given to the intended audience for trying out or temporarily using the product.

Unlike an alpha test, the beta test is being carried out by real users in the real environment. This allows the targeted customers to dive into the product's design, working, interface, functionality, etc.

Alpha Testing.

Alpha Testing can be defined as a form of acceptance testing which is carried out for identifying various types of issues or bugs before publishing the build or executable of software public or market. This test type focuses on the real users

through black box and white box testing techniques. The focus remains on the task which a general user might want or experience.

Alpha testing any product is done when product development is on the verge of completion. Slight changes in design can be made after conducting the alpha test. This testing methodology is performed in lab surroundings by the developers.

Here developers see things in the software from users point and try to detect the problems. These testers are internal company or organization's employees or may be a part of the testing team. Alpha testing is done early at the end of software development before beta testing.

Guidelines (for preparing test plan)

- Specify Requirements generated by user.
 - Specify Schedule and resources.
 - Determine the testing strategy.
 - Configuration Control System.
 - Keep the plan up to date.
 - At the end of each milestone, fill routine updates.
-
- You may have requirements that dictate a specific appearance or format for your test plan. These requirements may be generated by the users. Whatever the appearance of your test plan, try to include as much detail as possible about the tests.
 - The test plan should contain a schedule and a list of required resources. List how many people will be needed, when the testing will be done, and what equipment will be required.
 - After you have determined what types of testing are necessary (such as black box, white box, top-down, or bottom-up testing), you need to document specifically what you are going to do. Document every type of test you plan to complete. The level of detail in your plan may be driven by several factors, such as the following: How much test time do you have? Will you use the test plan as a training tool for newer team members?
 - A **configuration control system** provides a way of tracking the changes to the code. At a minimum, every time the code changes, a record should be kept that tracks which module has been changed, who changed it, and when it was altered, with a comment about why the change was made. Without configuration control, you may have difficulty keeping the testing in line with the changes, since frequent changes may occur without being communicated to the testers.
 - A well-thought-out design tends to produce better code and result in more complete testing, so it is a good idea to try to keep the plan up to date. Generally, the older a plan gets, the less useful it becomes. If a test plan is so old that it has become part of the fossil record, it is not terribly useful. As you approach the end of a project, you will have less time to create plans. If you do not take the time to document the work that needs to be done, you risk forgetting something in the mad dash to the finish line. Try to develop a habit of routinely bringing the test plan in sync with the product or product specification.
 - At the end of each month or as you reach each milestone, take time to complete routine updates. This will help you avoid being overwhelmed by being so out-of-date that you need to rewrite the whole plan. Keep configuration information on your plan, too. Notes about who made which updates and when can be very helpful down the road.

MYER'S DEBUGGING PRINCIPLES

Bug locating principles.

- ✓ Think
- ✓ If you reach an impasse, sleep on it.
- ✓ If the impasse remains, describe the problem to someone else. Use debugging tools.
- ✓ Experimentation should be done as a last resort.

Debugging principles.

- ✓ Where there is one bug , there is likely to be another.
- ✓ Fix the error, not just the symptom.
- ✓ The probability of solution being correct drops down as the size increases.
- ✓ Beware of error correction, it may create new errors

Case study :

Developing Test cases for vianet ATM bank system

Test cases are derived from the following use case scenarios

1. Bank Transaction
2. Checking transaction history
3. Savings/current account
4. Deposit/Withdarw
5. valid/invalid PIN

How do you rate the ViaNet bank ATM kiosk interface?											
	10	9	8	7	6	5	4	3	2	1	
Is easy to operate:	Very Easy	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Very Hard
Buttons are right size and easily can be located:	Very Appropriate	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Not Appropriate
Is efficient to use:	Very Efficient	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Very Inefficient
Is fun to use:	Fun	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	No Fun
Is visually pleasing:	Very Pleasing	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Not Pleasing
Provides easy recovery from errors:	Very Easy Recovery	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Not at All

Example of vianet ATM system for user satisfaction test