UNIT-I
PART-A

1. What is Analysis?

Analysis emphasizes an investigation of the problem and requirements, rather than a solution. Analysis is a broad term best qualified as in requirements analysis or object analysis. Object–Oriented Analysis (OOA) is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system's object model, which comprises of interacting objects. The main difference between object-oriented analysis and other forms of analysis is that in object-oriented approach, requirements are organized around objects, which integrate both data and functions.

2. What is Design?

Design emphasizes a conceptual solution that fulfills the requirements, rather than its implementation. For example a description of a database schema and software objects. Ultimately design can be implemented. Object–Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis. In OOD, concepts in the analysis model, which are technology−independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies.

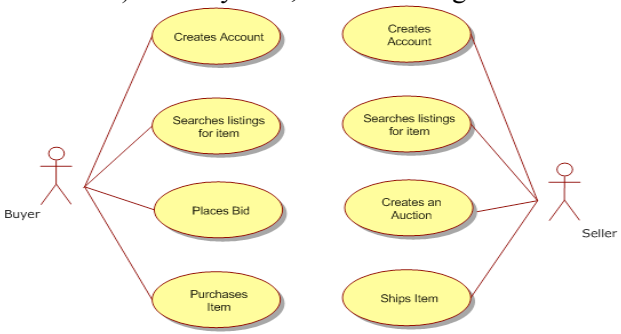
3. Define Object-Oriented Analysis and Design (Nov/Dec 2013) (May/June 2014)

OOA-there is an emphasis on finding and describing the objects or concepts in the problem domain.

OOD-there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. and Object-oriented analysis design (OOAD) is a popular technical approach to analyzing, designing an application, system, or business by applying the object-oriented paradigm and visual modeling throughout the development life cycles to foster better stakeholder communication and product quality.

4. Define Use Case. (Nov/Dec 2013)

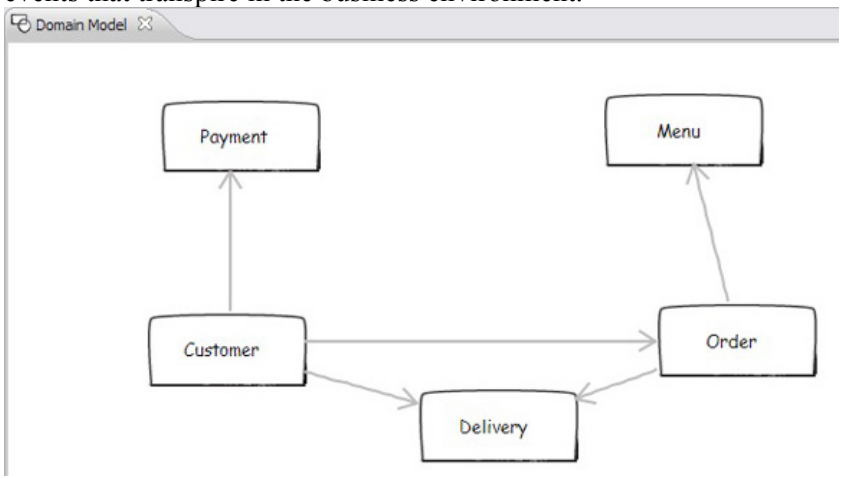
Requirements analysis may include a description of related domain processes these can be written as use cases. Use cases are not an object oriented artifact-they are simply written stories. However, they are a popular tool in requirements analysis and are an important part of the unified process. a use case is a list of steps, typically defining interactions between a role (known in Unified Modeling Language (UML) as an "actor") and a system, to achieve a goal. The actor can be a human, an external system, or time.



5. Define a Domain Model

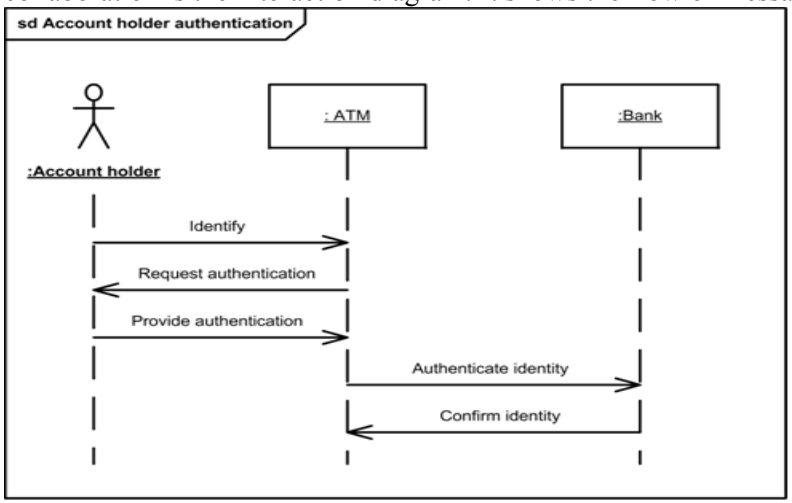
Domain model is a model in a set of diagrams that show domain concepts or objects. A domain

model captures the most important types of objects in the context of the business. The domain model represents the 'things' that exist or events that transpire in the business environment.



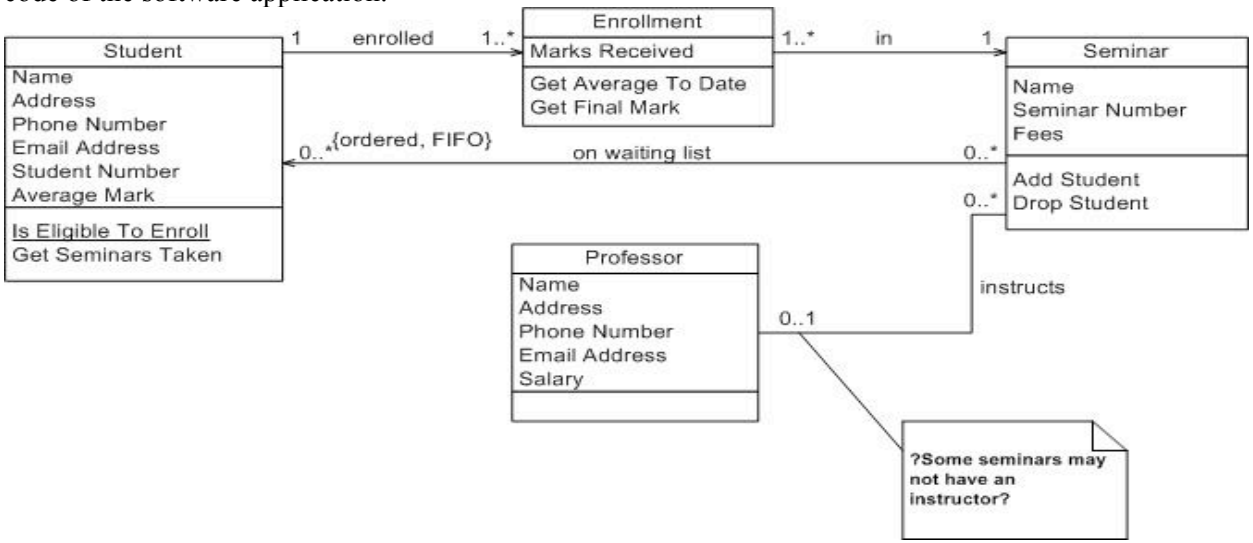
6.Define Interaction Diagrams

Object oriented design is concerned with defining software objects and their collaborations. A common notation to illustrate these collaboration is the interaction diagram. It shows the flow of messages between software objects and thus the invocation of methods.



7.Define class Diagrams

It will show the attributes and methods of the classes. The class diagram is a static diagram. It represents the static view of an application. Class diagram is not only used for visualizing, describing and documenting different aspects of a system but also for constructing executable code of the software application.



8.Define UML

The Unified Modeling Language is a language for specifying visualizing constructing and documenting the artifacts of software systems. As well as for business modeling and other non-software systems. The Unified Modeling Language (UML) is a general-purpose modeling language in the field of software engineering, which is designed to provide a standard way to visualize the design of a system.

9.Define Software development process

A software development process or life cycle is a structure imposed on the development of a software product. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process. It describes an approach to building, deploying and possibly maintaining software.

10.What is the use of Unified Process ?

The UP has emerged as a popular software development process for building object-oriented systems.The Unified Process is a design framework which guides the tasks, people and products of the design process. It is a framework because it provides the inputs and outputs of each activity. The Unified Software Development Process or Unified Process is a popular iterative and incremental software development process framework. The best-known and extensively documented refinement of the Unified Process is the Rational Unified Process(RUP). Other examples are OpenUP and Agile Unified Process.

11.Define Iterative and Incremental Development.

The system grows incrementally over time iteration and thus this approach is also know as iterative and incremental development. Iterative and Incremental development is any combination of both iterative design or iterative method and incremental build model for software development.

The combination is of long standing and has been widely suggested for large development efforts.

12.Benefits of Iterative Development.

Early visible progress

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.

13.Four major phases of UP

Inception

Elaboration

Construction

Transition

14.Define Development Case

The choice of UP artifacts for a project may be written up in a short document called the Development Case. It can show the sets of possible interactions between the system and the people who use it.It can also show interactions between computer systems

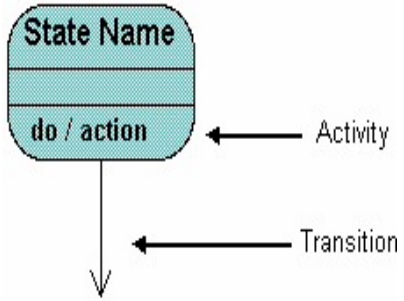
15.Define Elaboration (May/June 2014)

Elaboration is the intial series of iterations during which the team does serious investigation, implements the core architecture clarifies most requriments, and tackles the high –risk issues. Achieve a stable version of the majority of the requirements. Domain model, Design model and data model are the artifacts at the beginning of the elaboration.

16.Define State Diagram.

State diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur. Each diagram usually represents objects of a single class and track the different states of its objects through the system. The basic elements are rounded boxes representing the state of the object and arrows indicting the transition to the next state.

The activity section of the state symbol depicts what activities the object will be doing while it is in that state.

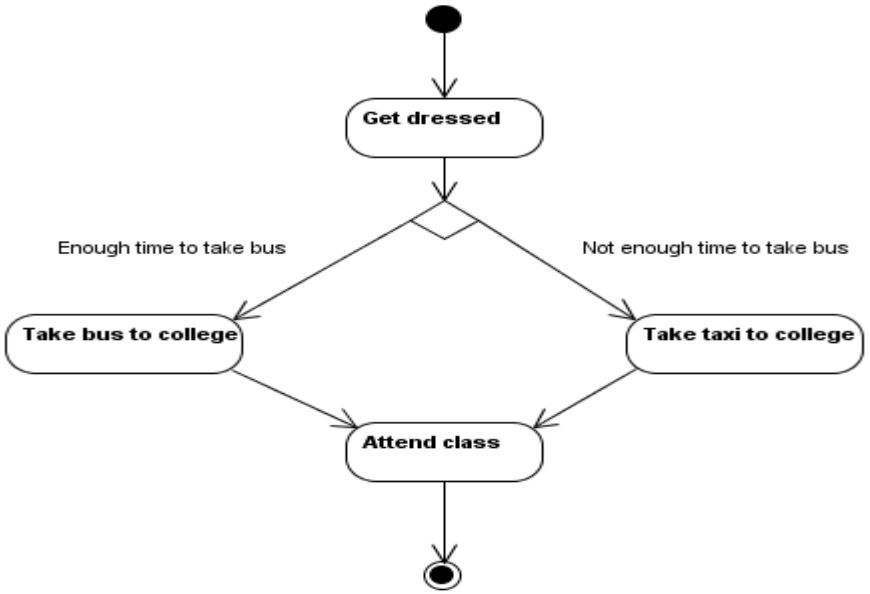


17.What is object oriented system development methodology?

Object oriented system development methodology is a way to develop software by building self-contained modules or objects that can be easily replaced,modified and reused.

18. Define activity diagram.

A diagram which is useful to visualize workflows and business processes. These can be a useful alternative or adjunct to writing the use case text, especially for business use cases that describe complex workflows involving many parties and concurrent actions.

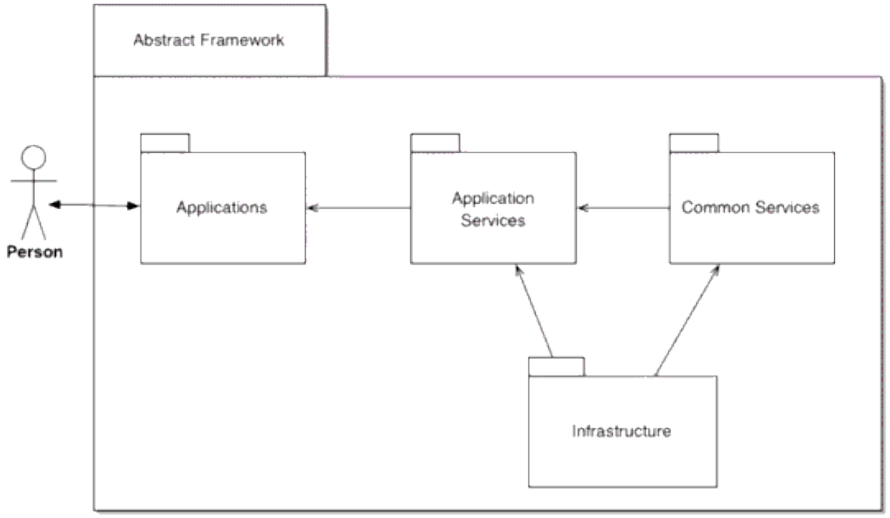


19. Define package diagram.

A package diagram in the Unified Modeling Language depicts the dependencies between the packages that make up a model. there are two special types of dependencies defined between packages:

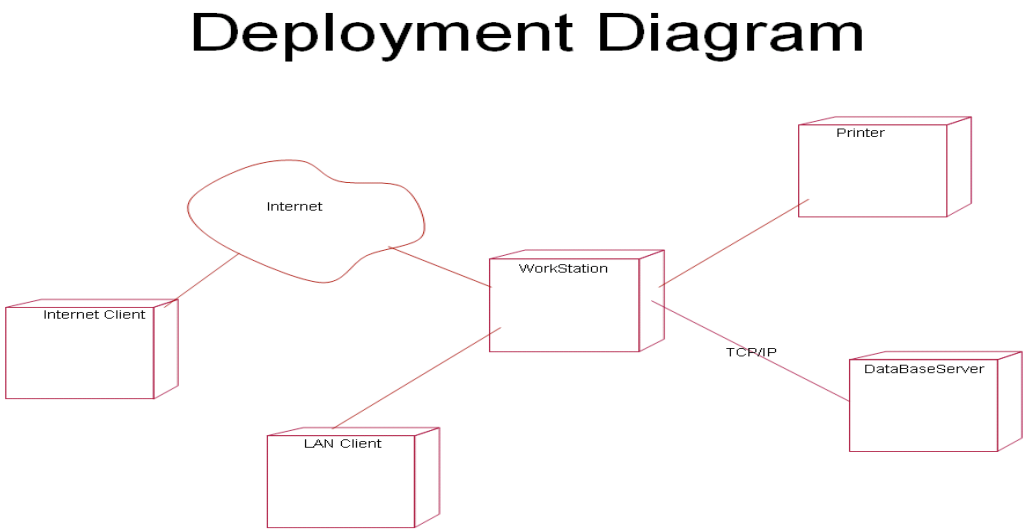
package import

package merge



20. Define Deployment diagram

A deployment diagram the Unified Modeling Language models the physical deployment of artifacts on nodes. The nodes appear as boxes, and the artifacts allocated to each node appear as rectangles within the boxes. Nodes may have sub nodes, which appear as nested boxes. A single node in a deployment diagram may conceptually represent multiple physical nodes, such as a cluster of database servers.



PART-B

1. **Explain in detail about object oriented emphasizes representation of objects.**

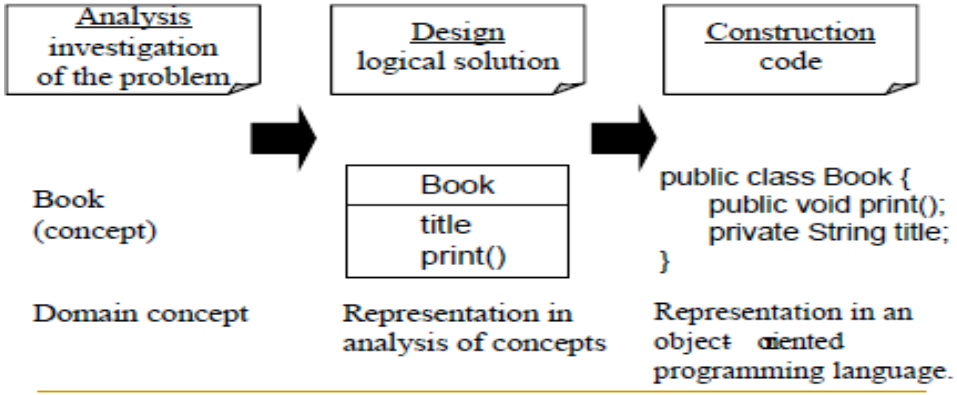
Object-Oriented Analysis:

- An investigation of the problem (rather than how a solution is defined)
- During OO analysis, there is an emphasis on finding and describing the objects (orconcepts) in the problem domain

Object-Oriented Design:

- Emphasizes a conceptual solution that fulfils the requirements.
- Need to define software objects and how they collaborate to fulfill the requirements
- For example, in the Library Information System, a *Book software object may have a title Attribute* and a *get Chapter method*

From Design to Implementation:



Constructing Use Case Diagrams from the External View

- Collect information sources—How am I supposed to know that?
- Identify potential actors—Which partners and customers use the goods and services of the business system?
- Identify potential business use cases—Which goods and services can actors draw upon?
- Connect business use cases—Who can make use of what goods and services of the business system?
- Describe actors—Who or what do the actors represent?
- Search for more business use cases—What else needs to be done?
- Edit business use cases—What actually has to be included in a business use case?
- Document business use cases—What happens in a business use case?
- Model relationships between business use cases—What activities are conducted repeatedly?

- Verify the view—Is everything correct?

2. **List various UML diagrams and explain the purpose of diagram (May/June 2014)**

UML Diagrams:

Diagrams are the heart of UML. These diagrams are broadly categorized as structural and behavioral diagrams.

- Structural diagrams are consists of static diagrams like class diagram, object diagram etc.
- Behavioral diagrams are consists of dynamic diagrams like sequence diagram, collaboration diagram etc.

The static and dynamic nature of a system is visualized by using these diagrams.

Class diagrams:

Class diagrams are the most popular UML diagrams used by the object oriented community. It describes the objects in a system and their relationships. Class diagram consists of attributes and functions.

A single class diagram describes a specific aspect of the system and the collection of class diagrams represents the whole system. Basically the class diagram represents the static view of a system.

Class diagrams are the only UML diagrams which can be mapped directly with object oriented languages. So it is widely used by the developer community.

Object Diagram:

An object diagram is an instance of a class diagram. So the basic elements are similar to a class diagram. Object diagrams are consists of objects and links. It captures the instance of the system at a particular moment.

Object diagrams are used for prototyping, reverse engineering and modeling practical scenarios.

Component Diagram:

Component diagrams are special kind of UML diagram to describe static implementation view of a system. Component diagrams consist of physical components like libraries, files, folders etc.

This diagram is used from implementation perspective. More than one component diagrams are used to represent the entire system. Forward and reverse engineering techniques are used to make executables from component diagrams.

Deployment Diagram:

Component diagrams are used to describe the static deployment view of a system. These diagrams are mainly used by system engineers.

Deployment diagrams are consists of nodes and their relationships. An efficient deployment diagram is an integral part of software application development.

Use Case Diagram;

Use case diagram is used to capture the dynamic nature of a system. It consists of use cases, actors and their relationships. Use case diagram is used at a high level design to capture the requirements of a system.

So it represents the system functionalities and their flow. Although the use case diagrams are not a good candidate for forward and reverse engineering but still they are used in a slightly differently way to model it.

Interaction Diagram:

Interaction diagrams are used for capturing dynamic nature of a system. Sequence and collaboration diagrams are the interaction diagrams used for this purpose.

Sequence diagrams are used to capture time ordering of message flow and collaboration diagrams are used to understand the structural organization of the system. Generally a set of sequence and collaboration diagrams are used to model an entire system.

Statechart Diagram:

Statechart diagrams are one of the five diagrams used for modeling dynamic nature of a system. These diagrams are used to model the entire life cycle of an object. Activity diagram is a special kind of Statechart diagram.

State of an object is defined as the condition where an object resides for a particular time and the object again moves to other states when some events occur. Statechart diagrams are also used for forward and reverse engineering.

Activity Diagram:

Activity diagram is another important diagram to describe dynamic behaviour. Activity diagram consists of activities, links, relationships etc. It models all types of flows like parallel, single, concurrent etc.

Activity diagram describes the flow control from one activity to another without any messages. These diagrams are used to model high level view of business requirements.

So before drawing an activity diagram we should identify the following elements:

- Activities
- Association
- Conditions
- Constraints

Once the above mentioned parameters are identified we need to make a mental layout of the entire flow. This mental layout is then transformed into an activity diagram.

The following is an example of an activity diagram for order management system. In the diagram four activities are identified which are associated with conditions. One important point should be clearly understood that an activity diagram cannot be exactly matched with the code. The activity diagram is made to understand the flow of activities and mainly used by the business users.

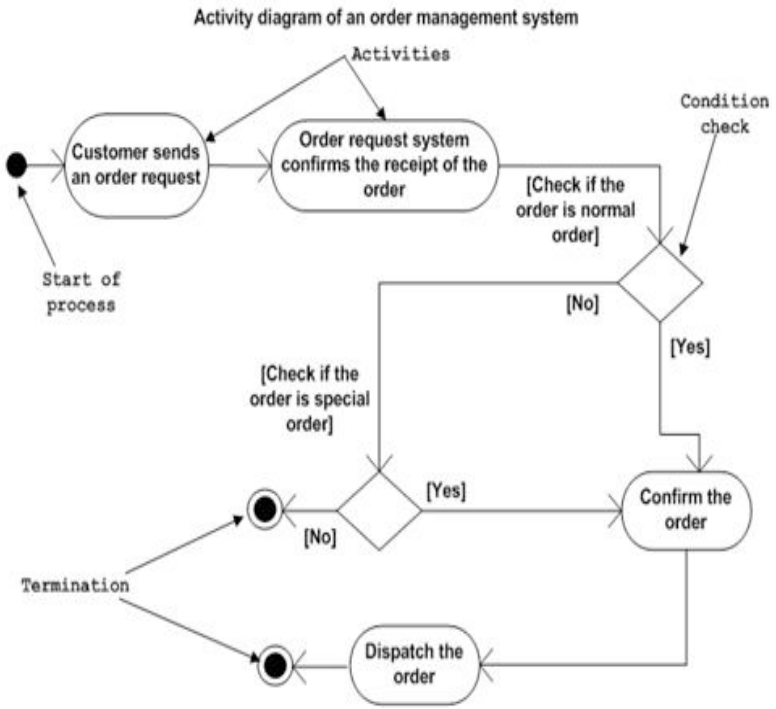
Once the above mentioned parameters are identified we need to make a mental layout of the entire flow. This mental layout is then transformed into an activity diagram.

The following is an example of an activity diagram for order management system. In the diagram four activities are identified which are associated with conditions. One important point should be clearly understood that an activity diagram cannot be exactly matched with the code. The activity diagram is made to understand the flow of activities and mainly used by the business users.

The following diagram is drawn with the four main activities:

- Send order by the customer
- Receipt of the order
- Confirm order
- Dispatch order

After receiving the order request condition checks are performed to check if it is normal or special order. After the type of order is identified dispatch activity is performed and that is marked as the termination of the process.



Where to use Activity Diagrams?

The basic usage of activity diagram is similar to other four UML diagrams. The specific usage is to model the control flow from one activity to another. This control flow does not include messages.

The 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 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.

Now we will look into the practical applications of the activity diagram. From the above discussion it is clear that an activity diagram is drawn from a very high level. So it 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 which are nothing but business requirements. So the diagram has more impact on business understanding rather implementation details.

Following are the **main usages of activity diagram**:

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

Before drawing a **Statechart diagram** we must have clarified the following points:

- Identify important objects to be analyzed.
- Identify the states.
- Identify the events.

the **main usages** can be described as:

- To model object states of a system.
- To model reactive system. Reactive system consists of reactive objects.
- To identify events responsible for state changes.

3. Briefly explain the different phases of unified process. (Nov/Dec 2013)

The Unified Process:

- The Unified Process has emerged as a popular and effective software development process.
- In particular, the Rational Unified Process, as modified at Rational Software, is widely practiced and adopted by industry.
- The critical idea in the Rational Unified Process is *Iterative Development*.
- Iterative Development is successively enlarging and refining a system through multiple iterations, using feedback and adaptation.
- Each iteration will include requirements, analysis, design, and implementation.
- Iterations are *timeboxed*.

Why a new methodology?

- The philosophy of process-oriented methods is that the requirements of a project are completely frozen before the design and development process commences. As this approach is not always feasible, there is also a need for flexible, adaptable and agile methods, which allow the developers to make late changes in specifications.

What is Rational Unified Process (RUP)?

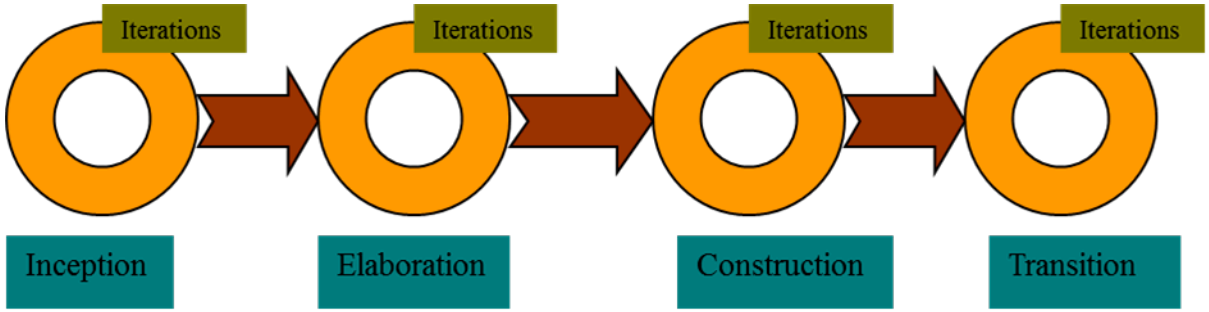
- RUP is a complete software-development process framework , developed by Rational Corporation.
- It's an iterative development methodology based upon six industry-proven best practices.
- Processes derived from RUP vary from lightweight—addressing the needs of small projects —to more comprehensive processes addressing the needs of large, possibly distributed project teams.

Phases in RUP

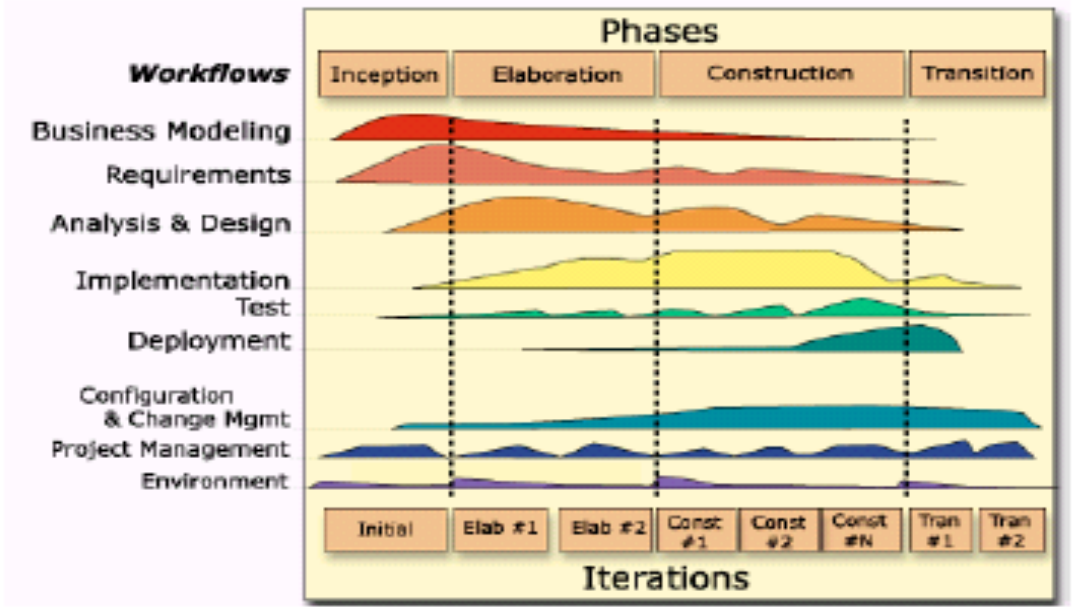
- RUP is divided into four phases, named:
- Inception
- Elaboration
- Construction
- Transition

Iterations

EEach phase has iterations, each having the purpose of producing a demonstrable piece of software. The duration of iteration may vary from two weeks or less up to six months.



Resource Histogram



RUP Overview Diagram

Un unified Process best practices

- Get high risk and high value first
- Constant user feedback and engagement
- Early cohesive core architecture
- Test early, often, and realistically
- Apply use cases where needed

- Do some visual modeling with UML
- Manage requirements
- Manage change requests and configuration

Inception

- The life-cycle objectives of the project are stated, so that the needs of every stakeholder are considered. Scope and boundary conditions, acceptance criteria and some requirements are established.

Inception – Activities

- **Formulate the scope of the project.**
- Needs of every stakeholder, scope, boundary conditions and acceptance criteria established.
- **Plan and prepare the business case.**
- Define risk mitigation strategy, develop an initial project plan and identify known cost, schedule, and profitability trade-offs.
- **Synthesize candidate architecture.**
- Candidate architecture is picked from various potential architectures

Elaboration

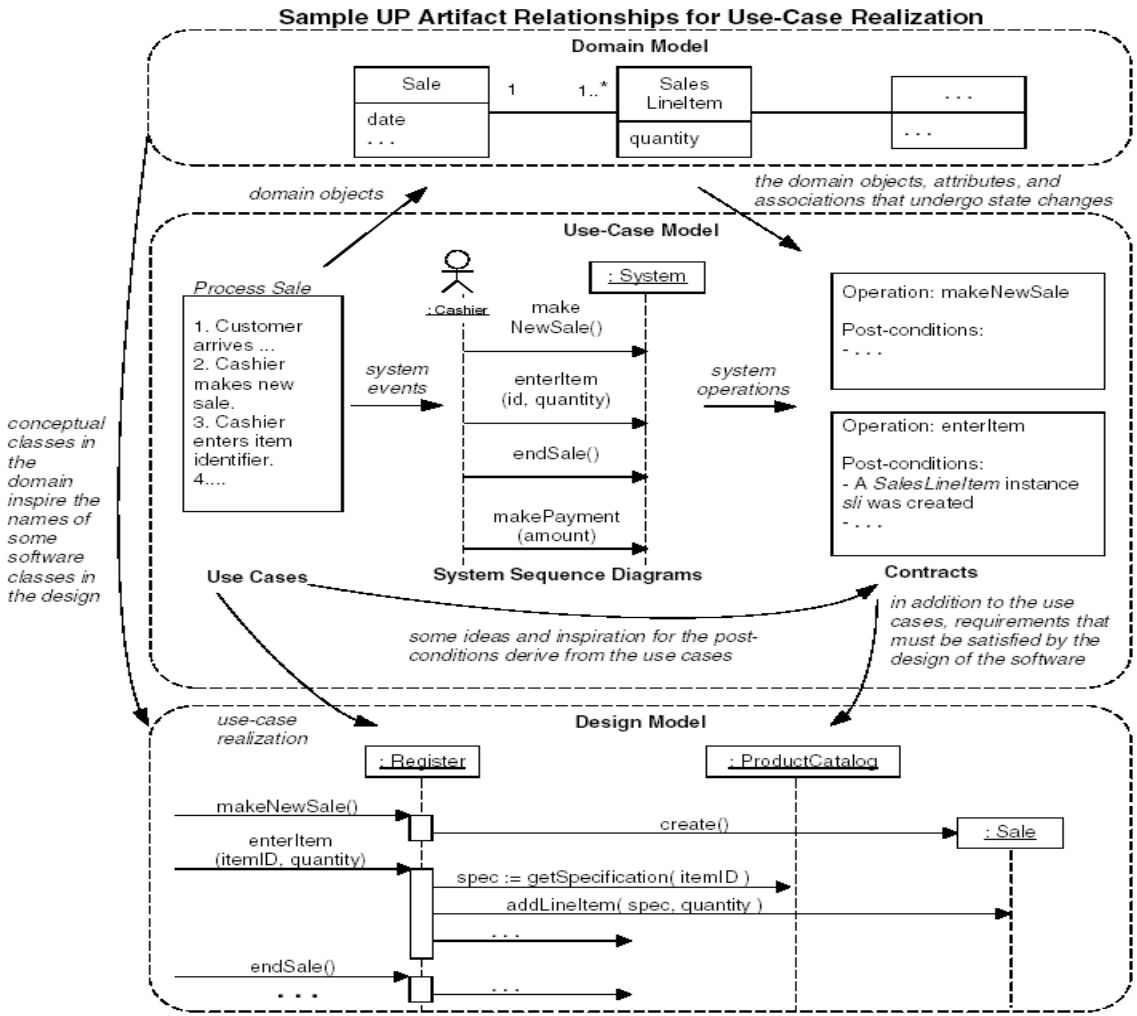
- **An analysis is done to determine the risks, stability of vision of what the product is to become, stability of architecture and expenditure of resources.**

Construction

- The Construction phase is a manufacturing process. It emphasizes managing resources and controlling operations to optimize costs, schedules and quality. This phase is broken into several iterations.

Transition

- The transition phase is the phase where the product is put in the hands of its end users. It involves issues of marketing, packaging, installing, configuring, supporting the user-community, making corrections, etc.



4. Explain with an example interaction diagram. (April/May 2011)

Interaction Diagram:

Interaction diagrams are used for capturing dynamic nature of a system. Sequence and collaboration diagrams are the interaction diagrams used for this purpose.

Sequence diagrams are used to capture time ordering of message flow and collaboration diagrams are used to understand the structural organization of the system. Generally a set of sequence and collaboration diagrams are used to model an entire system.

5. Explain about activity diagram with an example. (April/May 2011)

Activity Diagram:

Activity diagram is another important diagram to describe dynamic behaviour. Activity diagram consists of activities, links, relationships etc. It models all types of flows like parallel, single, concurrent etc.

Activity diagram describes the flow control from one activity to another without any messages. These diagrams are used to model high level view of business requirements.

So before drawing an activity diagram we should identify the following elements:

- Activities
- Association
- Conditions
- Constraints

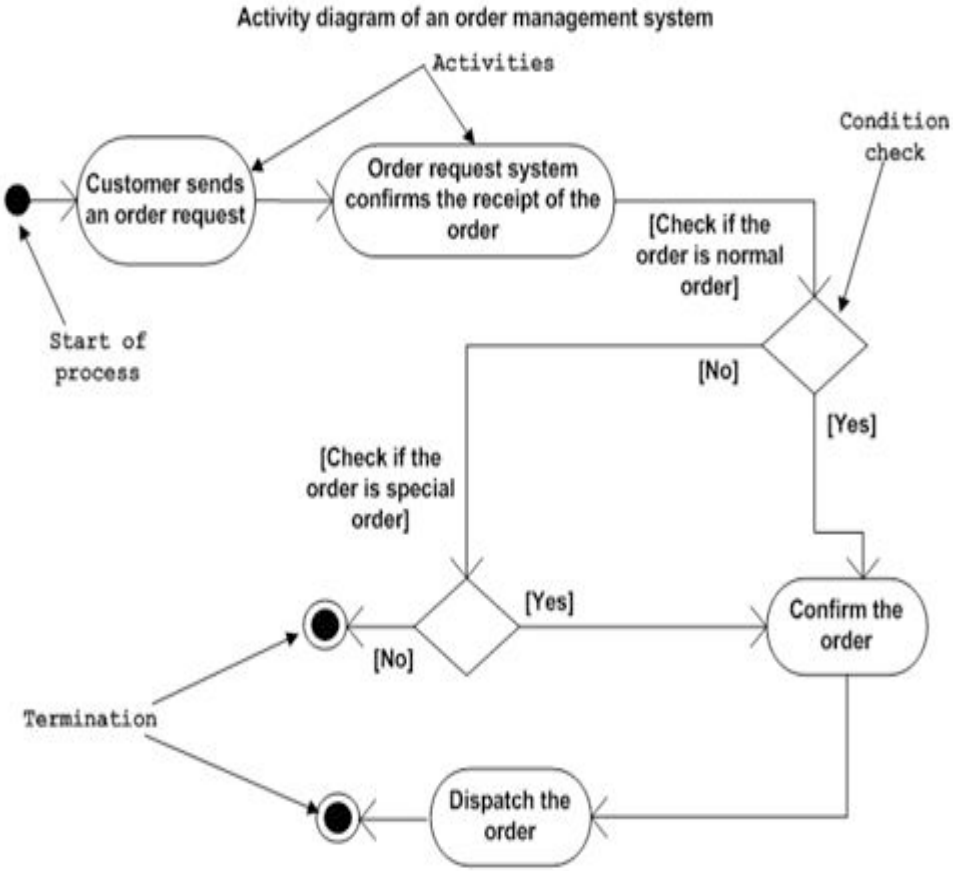
Once the above mentioned parameters are identified we need to make a mental layout of the entire flow. This mental layout is then transformed into an activity diagram.

The following is an example of an activity diagram for order management system. In the diagram four activities are identified which are associated with conditions. One important point should be clearly understood that an activity diagram cannot be exactly matched with the code. The activity diagram is made to understand the flow of activities and mainly used by the business users.

The following diagram is drawn with the four main activities:

- Send order by the customer
- Receipt of the order
- Confirm order
- Dispatch order

After receiving the order request condition checks are performed to check if it is normal or special order. After the type of order is identified dispatch activity is performed and that is marked as the termination of the process.



Where to use Activity Diagrams?

The basic usage of activity diagram is similar to other four UML diagrams. The specific usage is to model the control flow from one activity to another. This control flow does not include messages.

The 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 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.

Now we will look into the practical applications of the activity diagram. From the above discussion it is clear that an activity diagram is drawn from a very high level. So it 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 which are nothing but business requirements. So the diagram has more impact on business understanding rather implementation details.

Following are the **main usages of activity diagram**:

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

6. Explain in detail about the UP artifact and process context. (April/May 2011)

- The systems engineering discipline focuses on an elegant universe we call reality wherein the two dimensions of time and spaceestablish the landscape for the intertwining dance between the two naturalforces of change and complexity. It is within this arena that the keyingredients of teams and people, methodologies and processes, and tools andenabling technologies converge to bridge the chasm between vision and reality.At the core of every mature discipline, from the arts to the sciences andengineering, is a common language and common approaches enabling practitionersto collaborate and the discipline to evolve; and at the heart of this evolutionis capturing or acquiring, communicating or sharing, and leveraging or utilizingknowledge. Language establishes the boundaries of thought and behavior, itdefines concepts; methodology and process establish behavior within thatboundary, they apply concepts; and tools establish the automation of behaviorwithin that boundary, they automate the application of concepts. Quite simply,if we can't think it, we can't do it nor communicate it, and if we can't do it,we can't

automate it! Within the information systems and technology industry,the Unified Process (UP), Rational Unified Process (RUP), Unified ModelingLanguage (UML), and Software Process Engineering Metamodel (SPEM) are at the heart of this evolution.

- The Unified Process (UP) and Rational Unified Process (RUP)
- The Unified Process (UP) is a use-case-driven, architecture-centric, iterative and incremental development process frameworkthat leverages the Object Management Group's (OMG) UML and is compliant with theOMG's SPEM. The UP is broadly applicable to different types of software systems,including small-scale and large-scale projects having various degrees ofmanagerial and technical complexity, across different application domains and organizational cultures.
- The Unified Modeling Language (UML) is an evolutionary general-purpose, broadly applicable, tool-supported, and industry-standardized modeling language or collection of modeling techniques for specifying, visualizing, constructing, and documenting the artifacts of a system-intensive process. The UML is broadly applicable to different types of systems (software and non-software), domains (business versus software), and methods and processes. The UML enables and promotes (but does not require nor mandate) a use-case-driven, architecture-centric, iterative and incremental process.
- The UML emerged from the unification that occurred in the 1990s within the information systems and technology industry. Unification was led by Rational Software Corporation and the Three Amigos. The UML gained significant industry support from various organizations via the UML Partners Consortium and was submitted to and adopted by the OMG as a standard (November 1997).
- As the UML is an industry-standardized modeling language for communicating about systems, the Software Process Engineering Metamodel (SPEM) is an industry-standardized modeling language for communicating about processes and process frameworks (families of related processes) but it does not describe process enactment (the planning and execution of a process on a project). The SPEM began to emerge after the UML standardization effort, gained significant industry support from various organizations, and was adopted by the OMG as a standard (November 2001).
- System Development, Systems, Models, and Views
- The system development lifecycle process involves a problem-solving process at a macro-level and the scientific method at a micro-level. Requirements may be characterized as problems. Systems that address requirements may be characterized as solutions. Problem solving involves understanding or conceptualizing the problem or requirements by representing and interpreting the problem, solving the problem by manipulating the representation of the problem to derive or specify a representation of the solution, and implementing or realizing and constructing the solution or system that addresses the requirements by mapping the representation of the solution onto the solution world. Within each problem-solving step, the scientific method involves planning or predicting a hypothesis, executing or empirically testing the hypothesis, evaluating the hypothesis against the results, and deriving a conclusion that is used to update the hypothesis. These macro-level and micro-level processes are very natural and often occur subtly and sometimes unconsciously in system development!
- The UML facilitates and enables the problem-solving process. It facilitates specifying, visualizing, understanding, and documenting the problem or requirements; capturing, communicating, and leveraging strategic, tactical, and operational

knowledge in solving the problem; and specifying, visualizing, constructing, and documenting the solution or system that satisfies the requirements. It enables capturing, communicating, and leveraging knowledge concerning systems using models, architectural views, and diagrams.

- A system is a purposefully organized collection of elements or units. The architecture of a system entails two dimensions, the structural dimension and behavioral dimension, within its context. The structural or static dimension involves what elements constitute the system and their relationships. The behavioral or dynamic dimension involves how these elements collaborate and interact to satisfy the purpose of the system and provide its functionality or behavior.
- A model is a complete abstraction of a system that captures knowledge (semantics) about a problem and solution. An architectural view is an abstraction of a model that organizes knowledge in accordance with guidelines expressing idioms of usage. A diagram is a graphical projection of sets of model elements that depicts knowledge (syntax) about problems and solutions for communication. Within the fundamental UML notation, concepts are depicted as symbols and relationships among concepts are depicted as paths (lines) connecting symbols.
- Methodologies and Process Frameworks
- A program is a collection or portfolio of projects. A project is a specific problem-solving effort that formalizes the "work hard and hope for the best" approach. A method specifies or suggests how to conduct a project. A method's descriptive aspect specifies or suggests what knowledge is captured and communicated regarding a problem and solution. A method's prescriptive aspect specifies or suggests how knowledge is leveraged to solve the problem. A process is the execution of a method on a project.
- A methodology is a discipline or taxonomy, or well-organized collection, of related methods that addresses who does what activities on what work products, including when, how, why, and where such activities should be done. Workers (who), activities (how), work products (what), and the heuristics concerning them are commonly known as process elements. Methodologies group methods as a family, methods describe processes, and processes execute methods on projects.
- To provide more flexibility and scalability to address increasingly more diverse problems, where applying a single method may be insufficient and applying a whole methodology may be impractical, a subset of a whole methodology may be applied where the methodology is called a process framework and the actual subset of all of its methods that are applied on a specific project is called a process instance.
- A process framework specifies or suggests who does what activities on what work products, including when, how, why, and where such activities should be done for various types of projects. A process instance specifies or suggests who does what activities on what work products, including when, how, why, and where such activities should be done for a specific project. Process frameworks describe process instances as a more flexible and scaleable family of related processes, and process instances execute a subset of a process framework on projects. The UP is a process framework and Development Cases are process instances.
- The Unified Process (UP)

- To effectively and successfully apply the UP, we must understand collaborations, contexts, and interactions. As an effort or project leverages the UP, collaborations focus on the elements of the project, context focuses on the process framework for the project, and interactions focus on the execution of the project. Figure 1 shows the various elements of the UP.

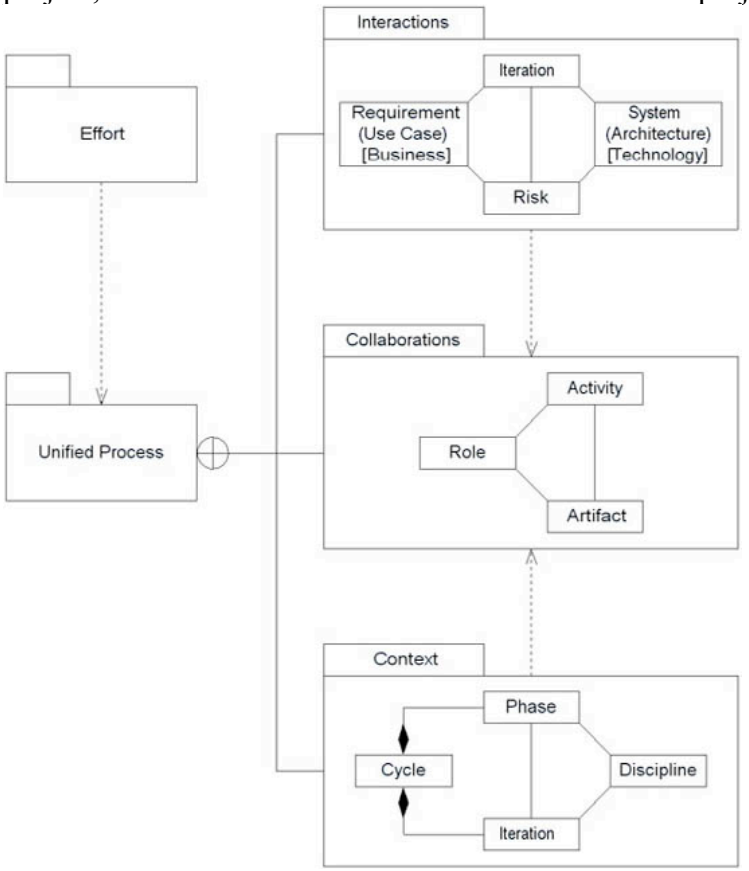


Figure 1: Elements of the Unified Process (UP)

- Collaborations
- A collaboration involves an interaction within a context. A collaboration captures who does what activities (how) on what work products. Thus, it establishes the elements of a project.
- A role is an individual or team who has responsibility for activities and artifacts. An activity is a unit of work, composed of steps, that is performed by a role. An artifact is an element of information that is the responsibility of a role and that is produced or consumed by activities. The UP defines numerous roles, artifacts, and activities.

- A context emphasizes the structural or static aspect of a collaboration, the elements that collaborate and their conglomeration or spatial relationships. A context captures when and where such activities should be done and work products produced and consumed. Thus, it establishes the context for a project. Figure 2 shows the context established by the UP.

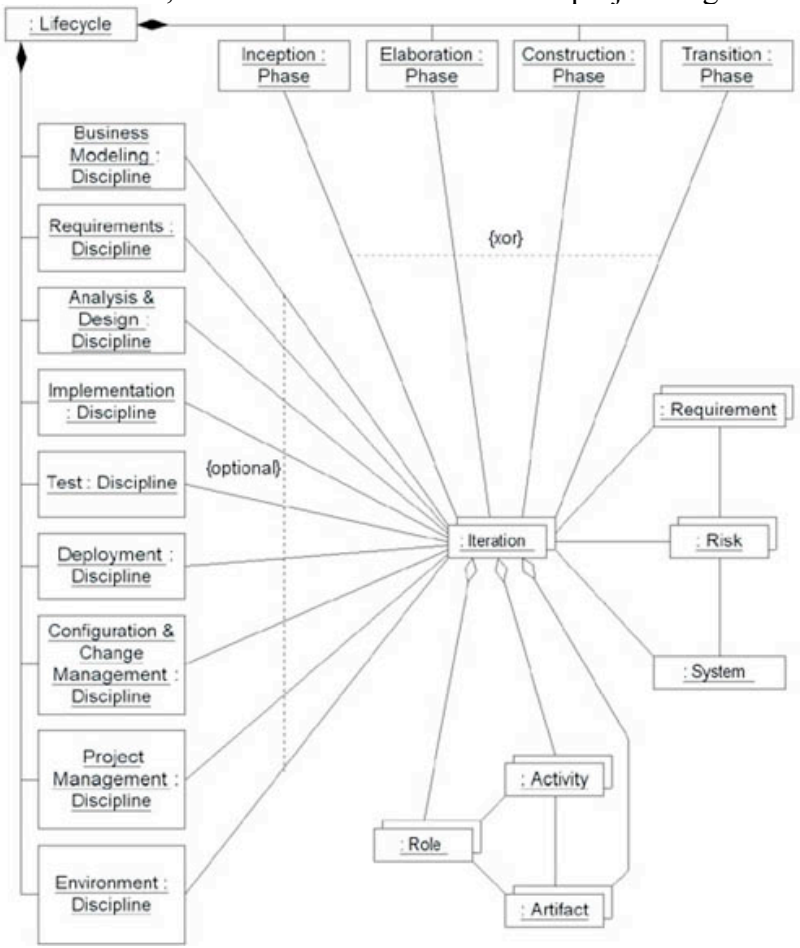


Figure 2: Context established by the Unified Process (UP).

The UP defines the following four phases:

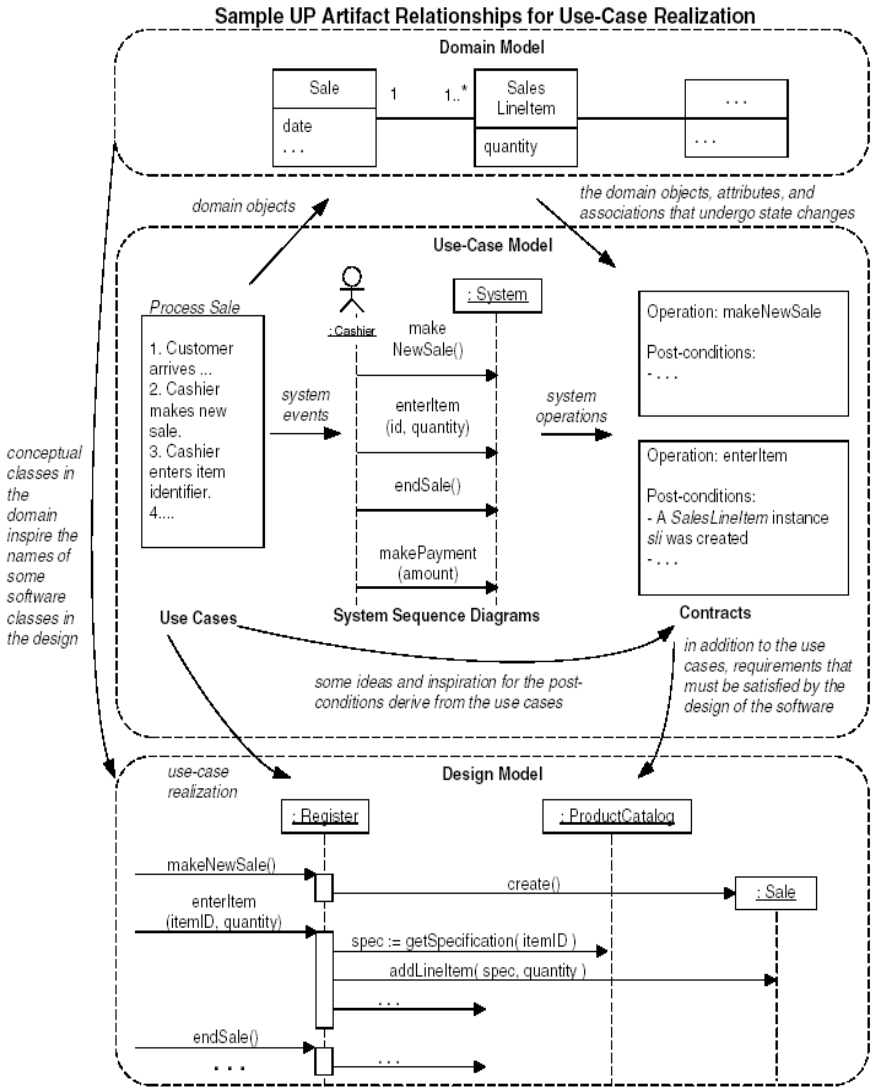
- The Inception phase, concluding with the Objective milestone, focuses on establishing the project's scope and vision; that is, establishing the business feasibility of the effort and stabilizing the objectives of the project.

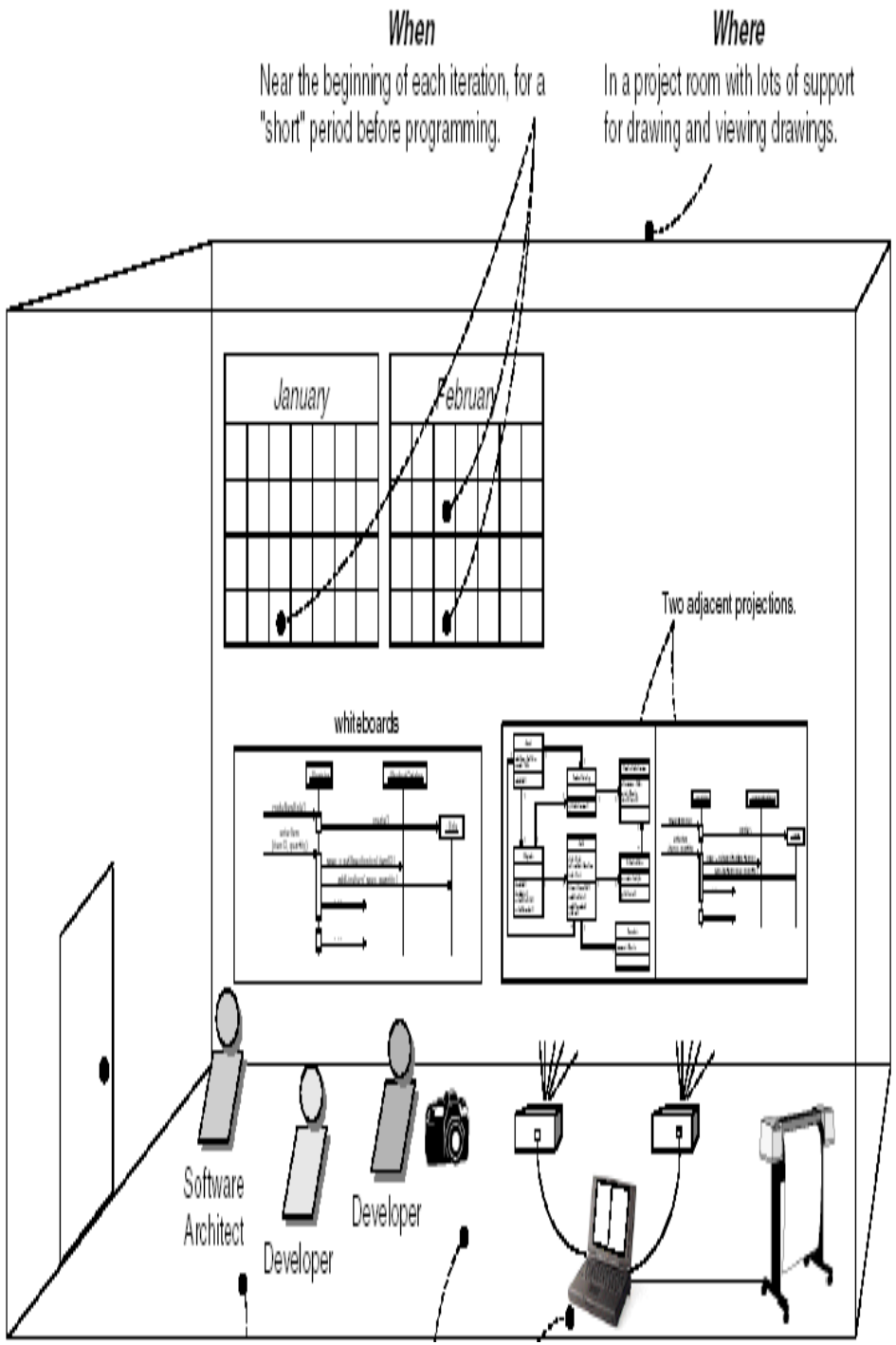
- The Elaboration phase, concluding with the Architecture milestone, focuses on establishing the system's requirements and architecture; that is, establishing the technical feasibility of the effort and stabilizing the architecture of the system.
- The Construction phase, concluding with the Initial Operational Capability milestone, focuses on completing construction or building of the system.
- The Transition phase, concluding with the Product Release milestone, focuses on completing transitioning or deployment of the system to the user community.
- The UP defines the following three supporting disciplines:
- The Configuration & Change Management discipline focuses on managing the configuration of the system and change requests.
- The Project Management discipline focuses on managing the project.
- The Environment discipline focuses on the environment for the project, including the process and tools.
- The UP defines the following six core disciplines:
- The Business Modeling discipline focuses on understanding he business being automated by the system and capturing such knowledge in a Business model.
- The Requirements discipline focuses on understanding the requirements of the system that automates the business and capturing such knowledge in a Use-case model.
- The Analysis & Design discipline focuses on analyzing the requirements and designing the system and capturing such knowledge in an Analysis/Design model.
- The Implementation discipline focuses on implementing the system based on the Implementation model.
- The Test discipline focuses on testing (evaluating) the system against the requirements based on the Test model.
- The Deployment discipline focuses on deploying the system based on the Deployment model.
- The distribution of effort across phases, iterations, and disciplines focuses on addressing business and technical risks. During the Inception phase, most of the effort is distributed across the Business Modeling and Requirements disciplines. During the Elaboration phase, most of the effort is distributed across the Requirements, Analysis & Design, and Implementation disciplines. During the Construction phase, most of the effort is distributed across the Analysis & Design, Implementation, and Test disciplines. During the Transition phase, most of the effort is distributed across the Test and Deployment disciplines. The supporting disciplines are generally distributed throughout the four phases. The overall objective is to produce the resulting system; therefore, all of the core disciplines are engaged as soon as possible without introducing risk to the project; that is, practitioners are responsible for determining which disciplines to engage and when they should be engaged.
- Interactions
- An interaction emphasizes the behavioral or dynamic aspect of a collaboration, the elements that collaborate and their cooperation or temporal communication. An interaction captures when and why such activities should be done and work products produced and consumed. Thus, it establishes the execution of a project as it is governed by various forces.

- As minor milestones occur within major milestones, technical decision points occur within management decision points such as to align technical tactics and operations with business strategy and objectives -- essentially, establishing a bridge between business and technical forces.
- An iteration is a step or leg along a path or route to a destination. An iteration is planned and is not ad hoc, has evaluation criteria, and results in demonstrable progress. An iteration is iterative in that it is repetitive and involves work and rework, incremental in that it is additive and involves more than rework alone, and parallel in that work may be concurrent within the iteration.
- A use-case is a functional requirement. For example, functionality to login or logout of a system, input data, process the data, generate reports, and so forth. As the UP is use-case driven, use cases drive or feed iterations. That is, iterations are planned and evaluated against "chunks" of functionality (or parts thereof) such as to manage agreement with users and trace project activities and artifacts back to requirements. Thus, accounting for business forces by planning and evaluating iterations against functional requirements. Non-functional requirements (usability, reliability, performance, and other such characteristics) are incrementally considered as use cases evolve through the disciplines.
- A system has an architecture. For example, the architecture of a system includes a collection of elements and how they collaborate and interact, including various subsystems for handling security, input and output, data storage, external communications, reporting, and so forth. As the UP is architecture-centric, iterations focus on architecture and evolving the system. That is, iterations demonstrate progress by evolving a puzzle of "chunks" such as to manage the complexity and integrity of the system. Thus, accounting for technical forces by demonstrating progress via the production and evolution of the real system.
- A risk is an obstacle to success, including human, business, and technical concerns or issues. For example, human risks include having insufficient, untrained, or inexperienced human resources, and so forth; business risks include having insufficient funding, time, or commitment from the business community, and so forth; and technical risks include having an insufficient understanding of the requirements or technology, using unproven technology, using technology that will not sufficiently address the requirements, and so forth. As the UP is risk-confronting, iterations confront risk and leverage feedback from previous iterations to confirm progress and discover other unknown risks. That is, iterations confront risk that is derived from use cases and architecture such as to achieve project success,thus reconciling business and technical forces.
- An iteration is a time-box with a fixed beginning and end wherein a collection of collaborations are planned, executed, and assessed in order to progressively demonstrate progress. The beginning and end are negotiated among stakeholders, management and technical members of the project community who impact and are impacted by the effort. Use cases that feed an iteration are selected based on the highest risks they confront. A use case may evolve across any number of iterations and may evolve through any number of core disciplines in an iteration. An iteration results in one or more intermediate builds or operational versions of the system. An iteration results in a single internal or external baselined and evaluated release of the system. The feedback and lessons-learned gained from an iteration feed into future iterations. Within an iterative approach,

metrics and estimates are also iteratively derived, and trends across iterations form the basis for metrics and estimation for the overall effort. The duration of an iteration is inversely proportional to the level of risk associated with the effort. As iterations execute, they only minimally overlap. Development cycles and phases may also be time-boxed; as development cycles, phases, and iterations are planned, the further the plans are in the future, the less accurate the estimates.

- Although iterations are composed of the same disciplines as a "pure waterfall" approach, there are key distinctions. A waterfall approach aims for one hundred percent completeness of activities and artifacts of a discipline before proceeding to the next discipline; however, an iterative approach involves iterative collaboration and aims for incremental refinement and evolving levels of detail of artifacts throughout the lifecycle. A waterfall approach does not offer explicit opportunities for partial deployment of a system or explicit opportunities for introducing change into the lifecycle, and is therefore quite reactive to change; however, an iterative approach does offer explicit opportunities for partial deployment of a system at the end of an iteration and explicit opportunities for introducing change into the lifecycle at the end of an iteration and before the next iteration, and is therefore quite proactive or responsive to change. A waterfall approach progresses serially through disciplines; however, an iterative approach may progress forward or backward across phases to change focus and involves various disciplines in order to address risk.
- Iterations
- To effectively and successfully apply the UP, we must understand iterations and how they are applied in linear, sequential, and iterative approaches.
- An iteration is planned, executed, and evaluated. Use cases and risks are prioritized, and use cases are ranked against the risks they mitigate. When planning an iteration, those use cases that address the highest risks and can be accommodated given the iteration's limiting factors (funding, time, resources, and so forth) are selected for driving the iteration. When executing an iteration, use cases evolve through the core disciplines and the system and its architecture evolve.
- However, use cases need not evolve through every core discipline in a single iteration. When evaluating an iteration, actual results are compared against the planned objectives of the iteration, and plans and risks are updated and adjusted. The overall objective is to produce the resulting system; therefore, all of the core disciplines are engaged as soon as possible without introducing risk to the project; that is, practitioners are responsible for determining which disciplines to engage and when they should be engaged.


Sample UP Artifact Relationships for Use-Case Realization
Domain Model
Sale
date
. . .
1
1..*
Sales LineItem
quantity
. . .
. . .
domain objects
the domain objects, attributes, and associations that undergo state changes
Use-Case Model
Process Sale
1. Customer arrives ...
2. Cashier makes new sale.
3. Cashier enters item identifier.
4....
: Cashier
: System
make NewSale()
enterItem (id, quantity)
endSale()
makePayment (amount)
system events
system operations
Operation: makeNewSale
Post-conditions:
- . . .
Operation: enterItem
Post-conditions:
- A SalesLineItem instance sli was created
- . . .
Use Cases
System Sequence Diagrams
Contracts
conceptual classes in the domain inspire the names of some software classes in the design
some ideas and inspiration for the post-conditions derive from the use cases
in addition to the use cases, requirements that must be satisfied by the design of the software
use-case realization
Design Model
: Register
: ProductCatalog
: Sale
makeNewSale()
create()
enterItem (itemID, quantity)
spec := getSpecification(itemID)
addLineItem(spec, quantity)
. . .
endSale()
. . .
. . .


When
Near the beginning of each iteration, for a "short" period before programming.
Where
In a project room with lots of support for drawing and viewing drawings.
January
February
Two adjacent projections.
whiteboards
Software Architect
Developer
Developer

7. **Discuss about UML deployment and component diagrams. Draw the diagrams for the banking application.(APRIL/MAY 2011)**

Refer Question No. 10

8. **Identify the actors, scenario and use cases for the example. (Nov/Dec 2013) (May/June 2014)**

Identifying the Actors

Often, people find it easiest to start the requirements elicitation process by identifying the actors. The following questions can help you identify the actors of your system

- Who uses the system?
- Who installs the system?
- Who starts up the system?
- Who maintains the system?
- Who shuts down the system?
- What other systems use this system?
- Who gets information from this system?
- Who provides information to the system?
- Does anything happen automatically at a present time?

Identifying the Use Cases

Then, the scenario-based requirements elicitation process continues by asking what outwardly visible, measurable result of value that each actor desires.

- What functions will the actor want from the system?
- Does the system store information? What actors will create, read, update or delete this information?
- Does the system need to notify an actor about chances in the internal state?
- Are there any external events the system must know about? What actor informs the system of those events?

A use case diagram is a visual representation of the relationships between actors and use cases together that documents the system's intended behavior.
There are several different kinds of relationships between actors and use cases. Earlier, we said that the default relationship is the communication relationship. The communication relationship indicates that one of these entities initiated communication or invoked request of the other. Obviously, an actor communicates with use cases because actors want measurable results. It might not be quite as obvious that use cases can communicate with other use cases. This happens when a use case needs information from or to initiate action of another use case. When a line or an arrow is draw on a diagram and there is no label on the arrow, it is, by default, a communication relationship.

The include relationship signifies that one use class is included in another's functionality. You use the include relationship when a chunk of behavior is similar across more than one use case and you don't want to keep copying the description of that behavior. This is similar to breaking out re-used functionality in a program into its own methods that other methods invoke for that functionality. For example, suppose many actions of a system require the user to log into the system before the functionality can be performed. These use cases would include the Login use case. Here's a hint. You should not break out a use case to be included by other use cases unless more than one other use case will include it (i.e. in a case diagram there should be more than one arrow coming into the included use case).
The include relationship is not the default relationship.

The extend Relationship
You use the extend relationship when you are describing a variation on normal behavior or behavior that is only executed under certain, stated conditions. You might wonder how this is different from simply stating alternative flows. The extend relationship is similar to the alternative flows of a use case. However, the extend relationship is used when the alternative flow is fairly complex and/or multi-stepped, possibly with sub-flows and alternative flows. For example, consider an earlier scenario of the chapter.
A player moves on the board because he or she has to go to jail.
A player moves on the board because he or she has to go to Free Parking.
This scenario involves a player moving. However, sometimes a player has to deal with "exceptional" situations – rather than just moving to a new property cell. Therefore, we can extend the Move use case with the Go to Jail and the Go to Free Parking use case.

Include Versus extend
Frequently, software developers are confused as to whether to use the include relationship or the extend relationship. Consider the following distinctions between the two:
• Use Case X includes Use Case Y:

X has a multi-step subtask Y. In the course of doing X or a subtask of X, Y will always be completed.

• Use Case X extends Use Case Y:

Y performs a sub-task and X is a similar but more specialized way of accomplishing that subtask (e.g. going to jail is a sub-task of Y; X provides an alternate means of moving). X only happens in an exception situation. Y can complete without X ever happening. In general, the extend relationship makes use cases difficult to understand. It is suggested that developers use this relationship sparingly.

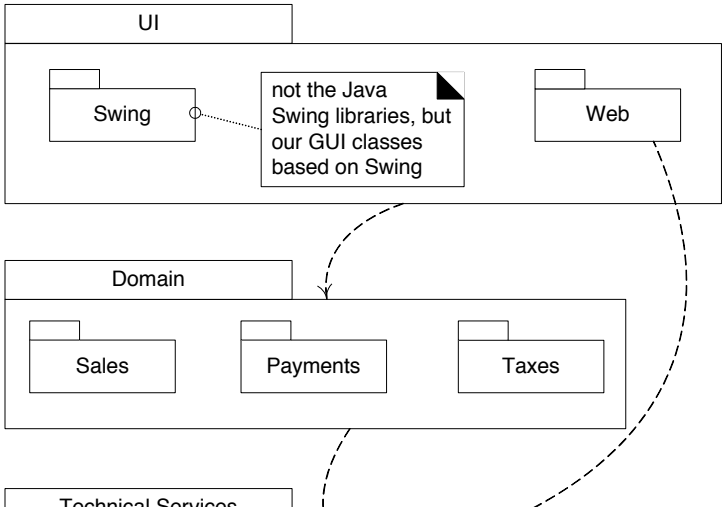
Misuse Cases

Privacy and security requirements are also included as a special kind of use case, the misuse case. A misuse case is a use case from the point of view of an actor hostile to the system; the actor is a hacker deliberately threatening the security of the system and/or the privacy of the users of the system.

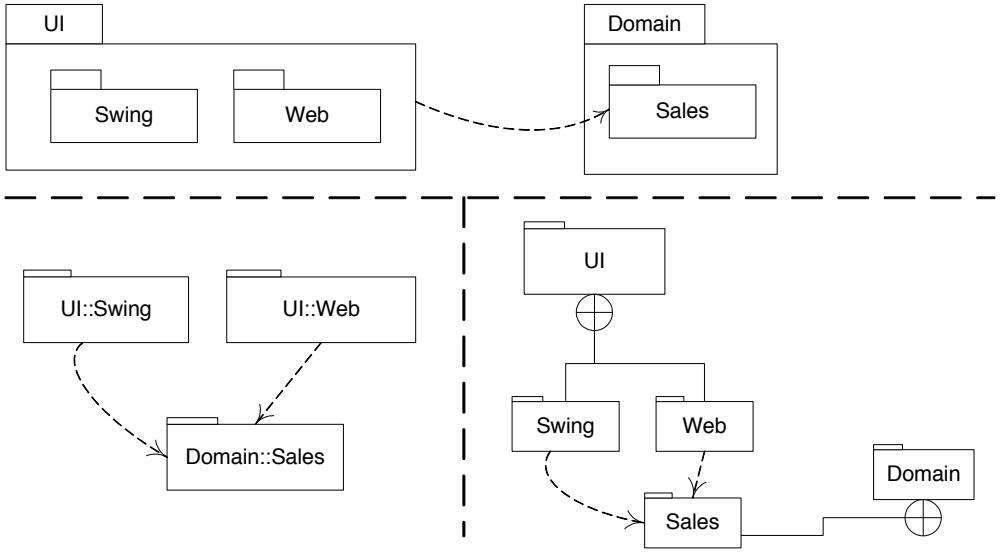
9. Explain in detail about Package diagram.

A UML package diagram provides a way to group elements. A UML package can group anything: classes, other packages, use cases, and so on. UML package diagrams are often used to illustrate the logical architecture of a system the layers, subsystems, packages (in the Java sense), etc. A layer can be modeled as a UML package; for example, the UI layer modeled as a package named UI.Nesting packages is very common.

Layers shown with UML package diagram



Various UML notations for package nesting



10. **Discuss about UML deployment and component diagrams. Draw the diagram for a banking Application (May/June 2014)**

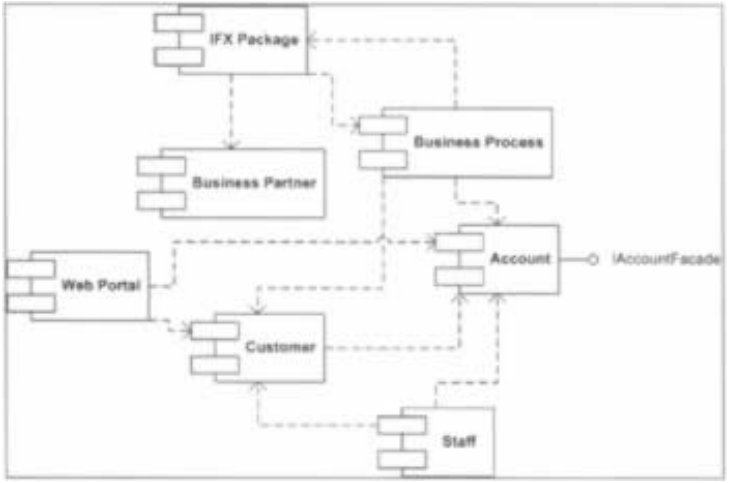
A deployment diagram in the Unified Modeling Language models the *physical* deployment of artifacts on nodes.To describe a web site, for example, a deployment diagram would show what hardware components ("nodes") exist (e.g., a web server, an application server, and a database server), what software components ("artifacts") run on each node (e.g., web application, database), and how the different pieces are connected (e.g. JDBC, REST, RMI).

The nodes appear as boxes, and the artifacts allocated to each node appear as rectangles within the boxes. Nodes may have subnodes, which appear as nested boxes. A single node in a deployment diagram may conceptually represent multiple physical nodes, such as a cluster of database servers.

There are two types of Nodes:

1. **Device Node**
2. **Execution Environment Node**

Device nodes are physical computing resources with processing memory and services to execute software, such as typical computers or mobile phones. An execution environment node (EEN) is a software computing resource that runs within an outer node and which itself provides a service to host and execute other executable software elements.



UNIT II

PART –A

1.GRASP methodical approach to learning basic object design?

General Responsibility Assignment Software Patterns (or Principles), abbreviated GRASP, consists of guidelines for assigning responsibility to classes and objects in object-oriented design. GRASP patterns are a learning aid to help one understand essential object design, and apply design reasoning in a methodical, rational, explainable way.

2.Define GRASP patterns?

GRASP patterns are really more accurately described as best practices. GRASP patterns outline best practices which can be employed in any object-oriented design. These best practices, if used properly, will lead to maintainable, reusable, understandable, and easy to develop software.

9 GRASP patterns: Creator, Information Expert, Low Coupling, Controller, High Cohesion, Indirection, Polymorphism, Protected Variations ,Pure Fabrication.

3. Define GRASP responsibilities?

Responsibilities is "a contract or obligation of a classifier" (UML definition). Responsibilities can include behaviour, data storage, object creation and more. They often fall into two categories:

Doing responsibilities of an object include:

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

Knowing

- Knowing responsibilities of object include :
  - Knowing about private encapsulated data
  - Knowing about related objects
  - Knowing about things it can derive or calculate

4. Define cohesion?

Cohesion – the degree to which the information and responsibilities of a class are related to each other. Cohesion refers to the degree to which the elements of a module belong together. Thus, it is a measure of how strongly related each piece of functionality expressed by the source code of a software module is. Cohesion refers to what the class (or module) will do. Low cohesion would mean that the class does a great variety of actions and is not focused on what it should do. High cohesion would then mean that the class is focused on what it should be doing, i.e. only methods relating to the intention of the class.

5.Define coupling? (Nov/Dec 2013)

Coupling of classes is a measure of how strongly a class is connected to another class. Coupling is the degree to which one class knows about another class. Let us consider two classes class **A** and class **B**. If class **A** knows class**B** through its interface only i.e it interacts with class **B** through its API then class **A** and class **B** are said to be loosely coupled.

6.What is creator?

The Creator GRASP ensures that coupling due to object instantiation only occurs on closely related classes .An aggregate or container of a class is already coupled with that class. Creation of objects is one of the most common activities in an object-oriented system. Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes. In general, a class B should be responsible for creating instances of class A if one, or preferably more, of the following apply:

- Instances of B contain or compositely aggregate instances of A
- Instances of B record instances of A
- Instances of B closely use instances of A
- Instances of B have the initializing information for instances of A and pass it on creation.

7.Define low coupling? (May/June 2014)

Assign a responsibility so that coupling remains low. This is pretty vague, but it means that low number of classes to which a class is coupled. Creator is a more specific case of Low Coupling, related to instantiation. Low Coupling is an evaluative pattern, which dictates how to assign responsibilities to support:

lower dependency between the classes, change in one class having lower impact on other classes, higher reuse potential.

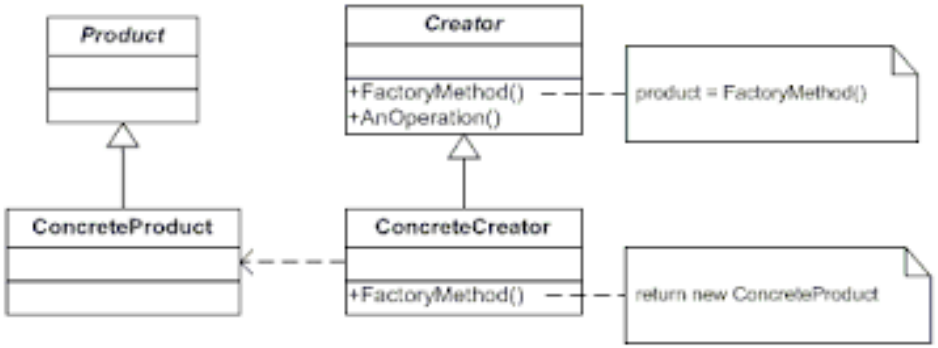
8.Define high cohesion?

High Cohesion is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable. High cohesion is generally used in support of Low Coupling. High cohesion means that the responsibilities of a given element are strongly related and highly focused. Breaking programs into classes and subsystems is an example of activities that increase the cohesive properties of a system. Alternatively, low cohesion is a situation in which a given element has too many unrelated responsibilities. Elements with low cohesion often suffer from being hard to comprehend, hard to reuse, hard to maintain and averse to change.

9.What is creator and its responsibilities?

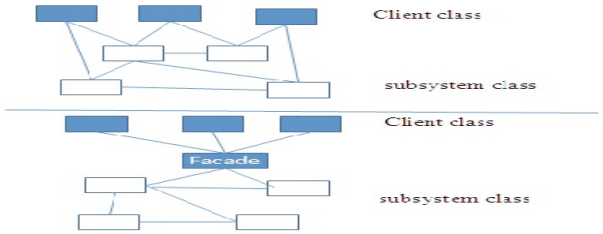
Creation of objects is one of the most common activities in an object-oriented system. Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes.Assign the responsibility for receiving and handling a system event message to a class that is either:

- Representative of the entire subsystem (e.g. a Façade Controller).
- Representative of the entire use case scenario.



10.What is facade controller?

The GRASP mentioned that a Controller that represents an entire subsystem might be called a Façade Controller. A facade is an object that provides a simplified interface to a larger body of code, such as a class library.



11. What is polymorphism?

In object-oriented programming, polymorphism (from the Greek meaning "having multiple forms") is the characteristic of being able to assign a different meaning or usage to something in different contexts - specifically, to allow an entity such as a variable, a function, or an object to have more than one form. There are several different kinds of polymorphism. When related behaviours vary by type (class), assign the responsibility polymorphically to the specialization classes. This is basically the purpose of polymorphism, so it is natural for software developers to understand. This is not much of a pattern, and yet another best practice.

12. What about pure fabrication?

To support high cohesion and low coupling, where no appropriate class is present, invent one even if the class does not represent a problem domain concept .This is a compromise that often has to be made to preserve cohesion and low coupling. This kind of class is called "Service" in Domain-driven design.

13. Define indirection?

To avoid direct coupling between objects, assign an intermediate object as a mediator.Recall that coupling between two classes of different subsystems can introduce maintenance problems. Another possibility is that two classes would be otherwise reusable (in other contexts) except that one has to know of the other.

14. What is protected variations?

Assign responsibility to create a stable interface around an unstable or predictably variable subsystem or component. If a component changes frequently, the users of the component will also have to be modified.This is especially time consuming if the component has many users.

15. What are the five GRASP patterns?

- Creator
- Information Expert
- Low Coupling
- Controller
- High Cohesion

16. What is Responsibility-Driven Design?

Responsibility-driven design is a design technique in object-oriented programming. A popular way of thinking about the design of software objects and also larger scale components are in terms of responsibilities, roles and collaborations. Responsibility-driven design is inspired by the client/server model. It focuses on the contract by asking:
What actions is this object responsible for?
What information does this object share?
This is part of a larger approach called responsibility driven design or RDD.

17. What are the advantages of Factory objects?

• 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.

18. Give example for Information Expert pattern or principle?

Name: Information Expert
Problem: How to assign responsibilities to objects?
Solution: Assign responsibility to the class that has the information needed to fulfill it

- ATM application – to know the balance amt we need
- 1. The acc no of a customer whose balance amt must be known.
- 2. The transactions till date for the same customer.

This information must be available in a class bank.
Therefore by information expert the bank class will return the acc no and balance details

19. State the use of Design patterns? (Nov/Dec 2013) (Nov/Dec 2014)

- **Understandability**: Classes are easier to understand in isolation.
- **Maintainability:** Classes aren't affected by changes in other components.
- **Reusability**: easier to grab hold of classes.

20. Write the 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.

PART B

1. **Explain about GRASP designing objects with responsibilities. (Nov/Dec 2013) (May/June 2014)**

GRASP stands for General Responsibility Assignment Software Patterns.
There are nine GRASP patterns:

Creator	Controller	Pure Fabrication
Information Expert	High Cohesion	Indirection
Low Coupling	Polymorphism	Protected Variations

Creator:

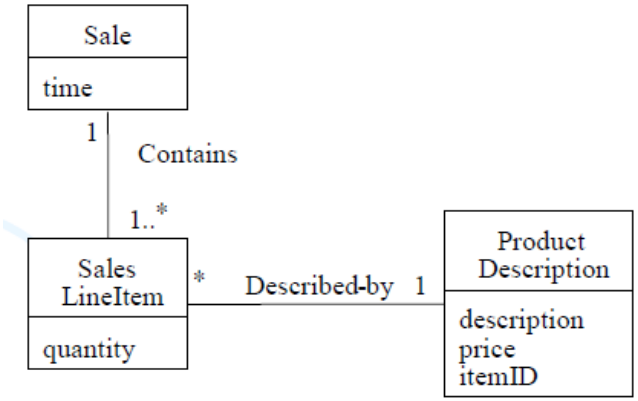
Problem: Who should be responsible for creating a new instance of some class?

The creation of objects is one of the most common activities in an object-oriented system. It is useful to have a general principle for the assignment of creation responsibilities. Assigned well, the design can support low coupling, increased clarity, encapsulation and reusability.

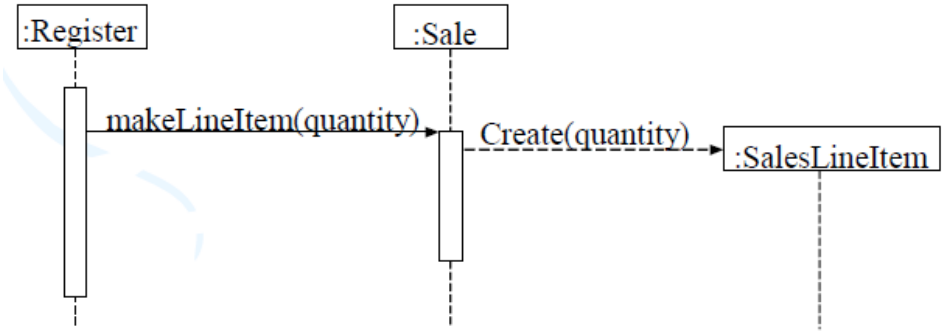
Solution: Assign class B the responsibility to create an instance of class A if one of these is true (the more the better):

- ♣ B "contains" or compositely aggregates A.
- ♣ B records A.
- ♣ B closely uses A.
- ♣ B has the initializing data for A that will be passed to A when it is created. Thus B is an Expert with respect to creating A.

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



PARTIAL DOMAIN MODEL



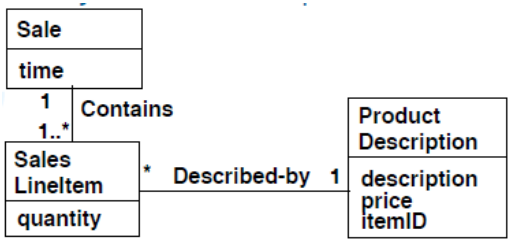
CREATING A SALESLINEITEM

Information Expert:

Problem: What is a general principle of assigning responsibilities to objects?

A design model may define hundreds of thousands of software classes, and an application may require hundreds of thousands of responsibilities to be fulfilled. During object design, when the interactions between objects are defined, choices about the assignment of responsibilities to software classes can be made.

Solution: Assign a responsibility to the information expert-the class that has the information necessary to fulfill the responsibility.



ASSOCIATION OF SALE

Low Coupling

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

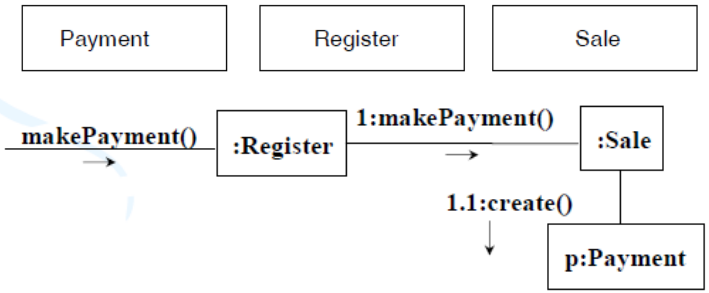
Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on another elements. An element with low or weak coupling is not dependent on too many other elements. These elements include classes, subsystems, and systems.

A class with high or strong coupling relies on many other classes. Such classes may be undesirable; some suffer from the following problems:

- ♣ 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.

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



SALE CREATES PLACEMENT

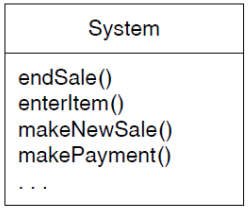
Controller:

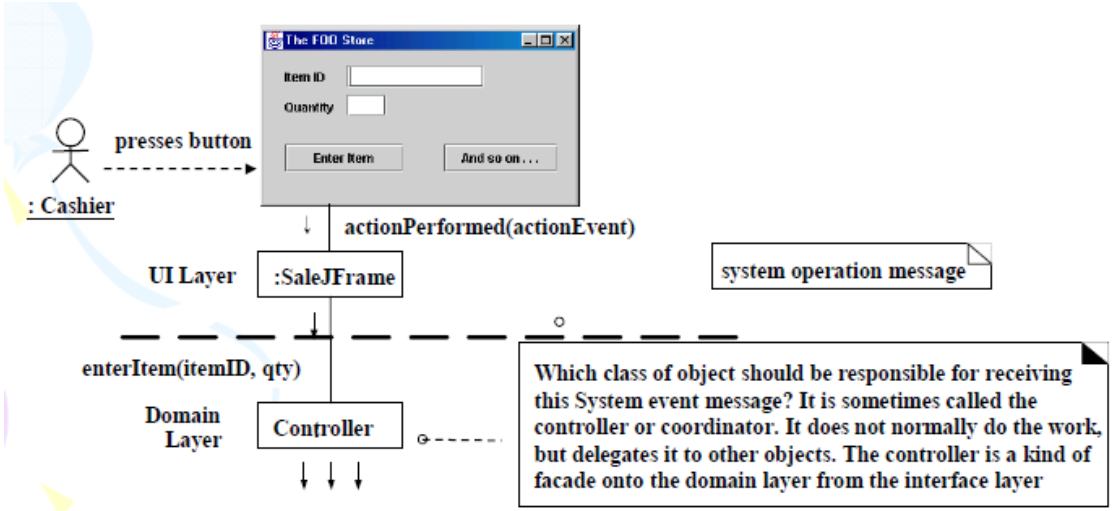
Problem: What first object beyond the UI layer receives and coordinates (controls) a system operation?

A Controller is the first object beyond the UI layer that is responsible for receiving or handling a system operation message.

Solution: assign the responsibility to a class representing one of the following choices:

- ♣ 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.*
- ♣ Represents a use case scenario within which the system event occurs, often named <UseCaseName> Handler, <UseCaseName> Coordinator, or <UseCaseName> Session.
  - → Use the same controller class for all system events in the same use case scenario.
  - → Informally, asession is an instance of a conversation with an actor. Sessions can be of any length but are often organized in terms of use cases.



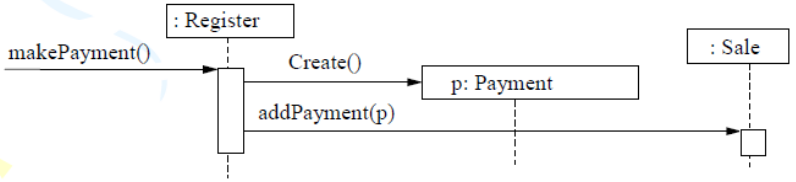


WHAT OBJECT SHOULD BE THE CONTROLLER FOR ENTERITEM?

High Cohesion

Problem: How to keep objects focused, understandable, and manageable and as a side effect, support low coupling?

Cohesion is a measure of how strongly related and focused the responsibilities of an element are. An element with highly 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.



REGISTER CREATES PAYMENT

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 the following problems:

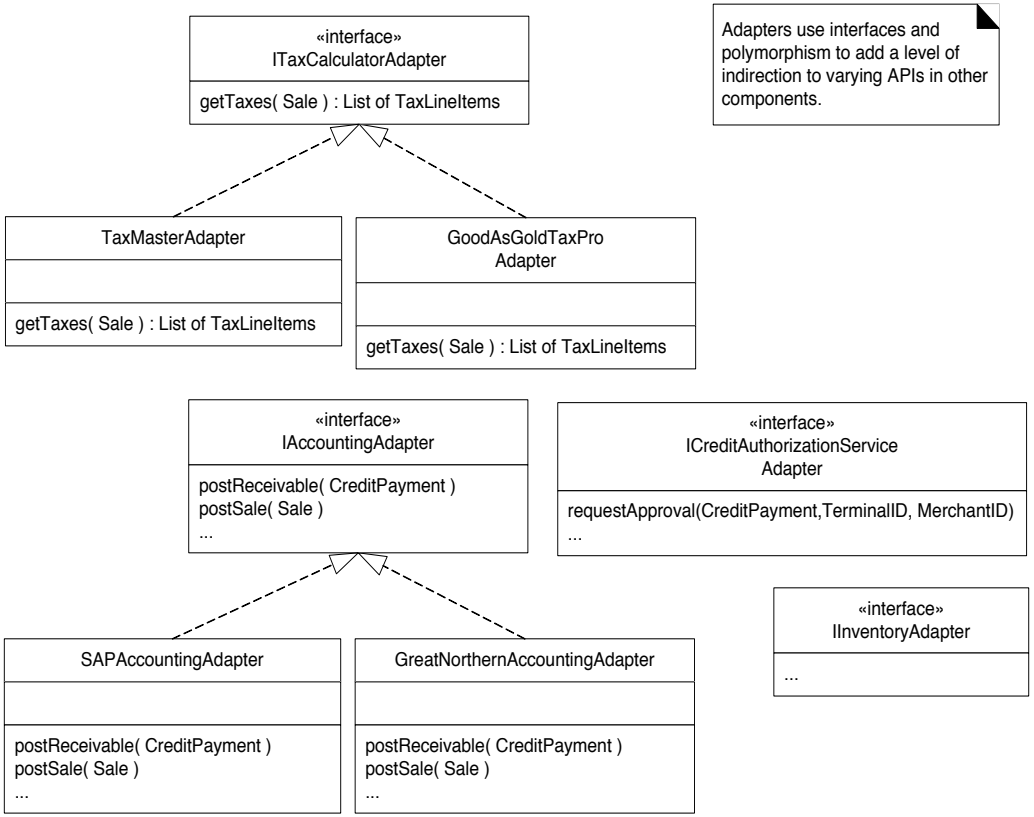
- ♣ Hard to comprehend.
- ♣ Hard to reuse.
- ♣ Hard to maintain.
- ♣ Delicate; constantly affected by change.

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

2. Write short notes on adapter, singleton, factory and observer patterns.(Nov/Dec 2013) (May/June 2014)

Adapter (GoF)

The NextGen problem explored to motivate the polymorphism pattern and its solution is an example of the GoF Adapter Pattern



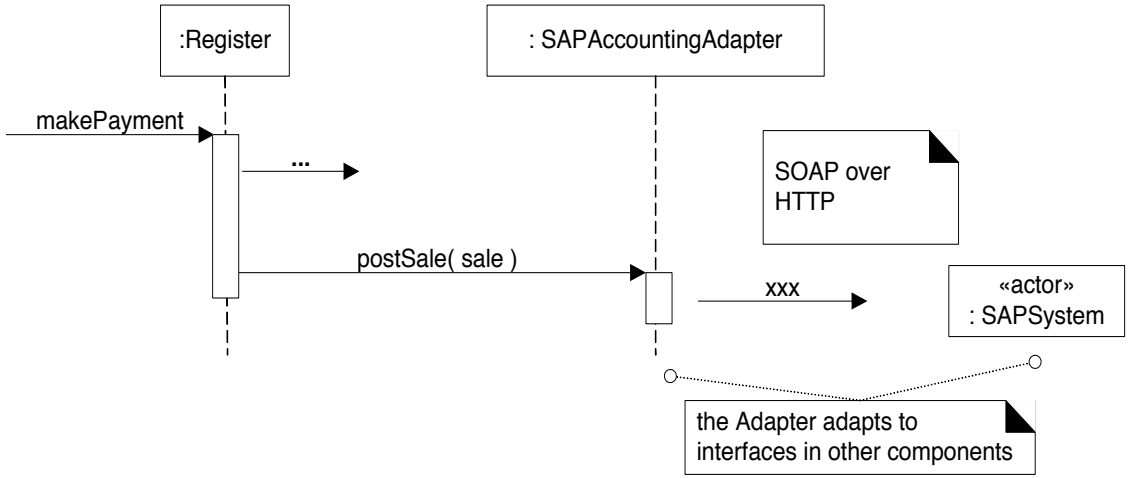
THE ADAPTER PATTERN

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

To review : the NextGen POS system needs to support several kinds of external third-party services, including tax calculators, credit authorization services, inventory systems, and accounting systems among others. Each has a different API, which can't be changed.

A solution is to add a level of indirection with objects that adapt the varying external interfaces to a consistent interface used within the application. The solution is illustrated in the Figure.

As illustrated in Figure, a particular adapter instance will be instantiated for the chosen external service, such as SAP for accounting, and will adapt the *postSale* request to the external interface, such as a SOAP XML interface over HTTPS for an intranet Web service offered by SAP.



USING AN ADAPTER

A resource aapter that hides an external system may also be considered a Façade object, as it wraps access to the subsystem or system with a single object. The motivation to call it a resource adapter especiallly exists when the wrapping object provides adaptation to varying external interfaces.

Singleton (GoF)

The *ServicesFactory* raises a problem in the design of who creates the factory and how it is accessed.

First, observe that only one instance of the factory in needed within the process. Second, quick reflection suggests that the methods of this factory may need to be called from various places in the code, as different places need access to the adapters for calling on the external services. Thus visibility problem arises.

One solution for the visibility problem is pass the *ServicesFactory* instance around as a parameter to wherever a visibility need is discovered for if, or to initialize the objects that need visibility to it, with a permanent reference. This is possible but inconvenient; an alternative is the Singleton pattern.

It is desirable to support global visibility or a single access point to a single instance of a class rather than some other form of visibility. This is true for *ServicesFactory* instance.

Name : Singleton

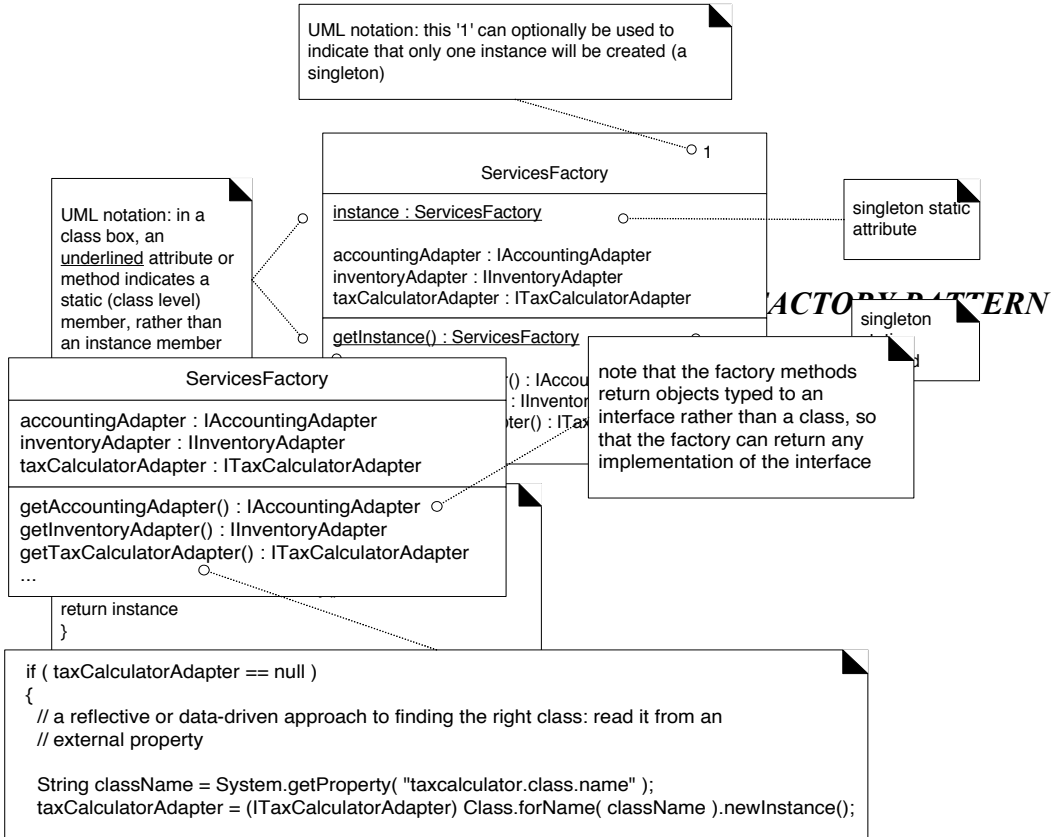
Problem : Exactly one instance of a class is allowed-it is a “singleton”. Objects need a global and single point of access.

Solution : Define a static method of the class that returns the singleton.

THE SINGLETON PATTERN IN THE SERVICESFACTORY CLASS

Factory

This is also called Simple Factory or Concrete Factory. This pattern is not a GoF design pattern, but extremely widespread.



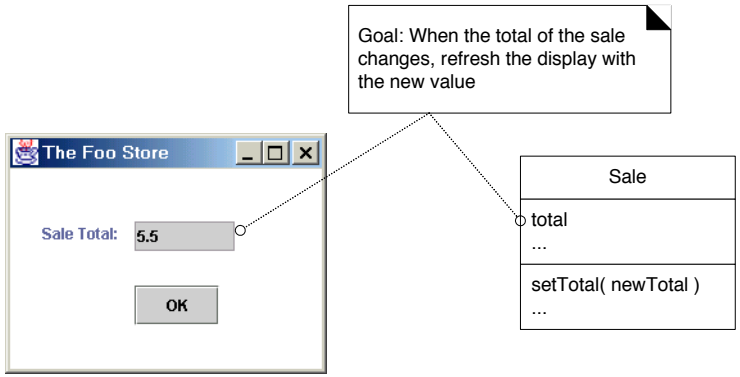
It is also a simplification of the GoF Abstract Factory pattern, and often described as a variation of Abstract Factory, although that's not strictly accurate.
The adapter raises a new problem in the design: In the prior Adapter pattern solution for the external services with varying interfaces. This point underscores another fundamental design principle: Design to maintain a **separation of concerns.** This is modularized or separate distinct concerns into different areas. Choosing a domain object to create the adapters does not support the goal of a separation of concerns, and lower its cohesion.
A common alternative in this case in to apply the Factory pattern, in which a Pure Fabrication "factory" object is defined to create objects.
Factory objects have several disadvantages:

- ♣ 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.

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?
Solution : Create a pure fabrication object called Factory that handles the creation.

Observer:

Another requirement for iteration is adding the ability of GUI window to refresh its display of the sale total when the total changes.



UPDATING THE INTERFACE WHEN THE SALE TOTAL CHANGES

To review, the Model-View Separation principle discourages such solutions. It states that "model" objects should not know about view or presentation objects such as a window. It promotes low coupling from other layers to the presentation layer of objects.

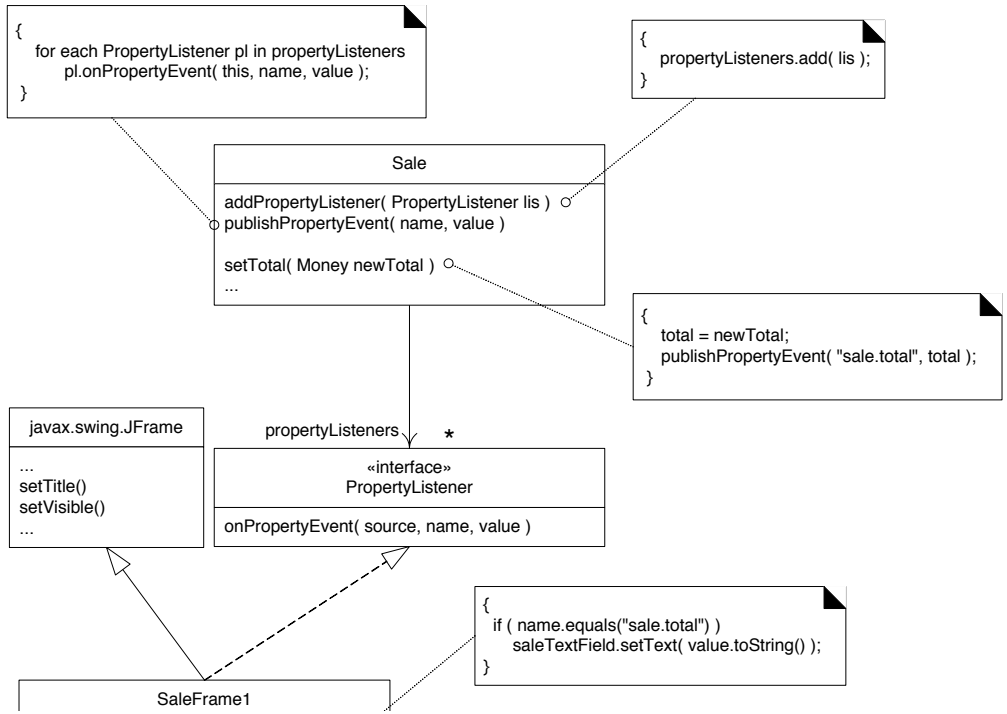
A consequence of supporting this low coupling is that it allows the replacement of the view or presentation layer by a new one or of particular windows by new windows without impacting the non-UI objects.

Thus Model-View Separation supports Protected Variations with respect to a changing user interface. To solve this design problem, the observer pattern can be used.

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 subscribers. What to do?

Solution : 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.



3. Explain Creator and Information Expert with an Example?(May/June 2013)

Creator:

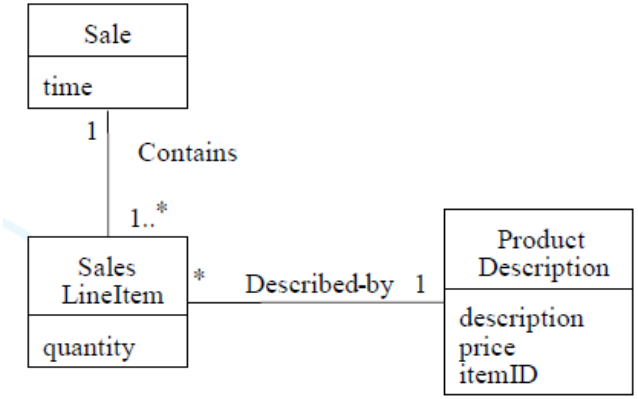
Problem: Who should be responsible for creating a new instance of some class?

The creation of objects is one of the most common activities in an object-oriented system. It is useful to have a general principle for the assignment of creation responsibilities. Assigned well, the design can support low coupling, increased clarity, encapsulation and reusability.

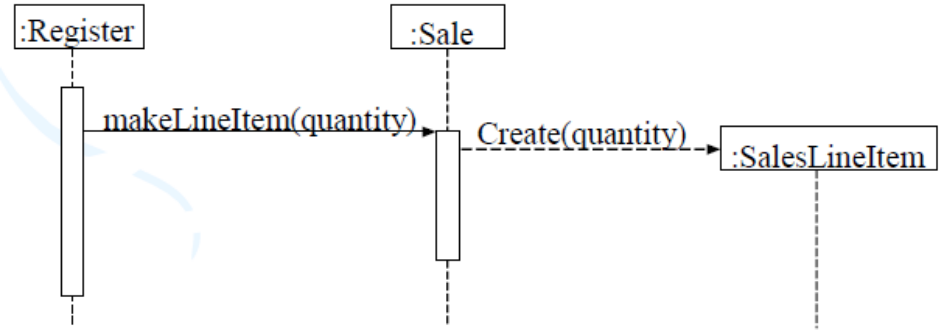
Solution: Assign class B the responsibility to create an instance of class A if one of these is true (the more the better):

- ♣ B "contains" or compositely aggregates A.
- ♣ B records A.
- ♣ B closely uses A.
- ♣ B has the initializing data for A that will be passed to A when it is created. Thus B is an Expert with respect to creating A.

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



PARTIAL DOMAIN MODEL



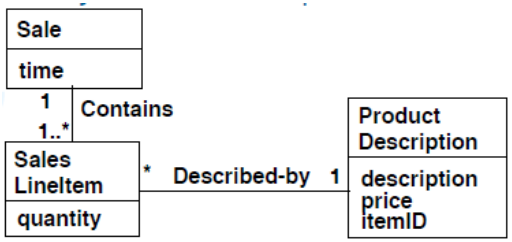
CREATING A SALESLINEITEM

Information Expert:

Problem: What is a general principle of assigning responsibilities to objects?

A design model may define hundreds of thousands of software classes, and an application may require hundreds of thousands of responsibilities to be fulfilled. During object design, when the interactions between objects are defined, choices about the assignment of responsibilities to software classes can be made.

Solution: Assign a responsibility to the information expert-the class that has the information necessary to fulfill the responsibility.



ASSOCIATION OF SALE

4. Explain Low Coupling and Controller with an Example? (May/June 2013)

Low Coupling

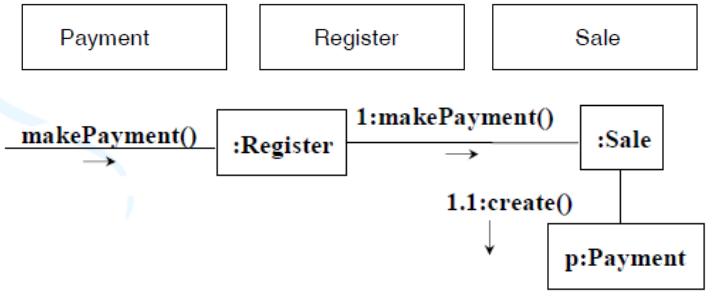
Problem: How to support low dependency, low change impact, and increased reuse?

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on another elements. An element with low or weak coupling is not dependent on too many other elements. These elements include classes, subsystems, and systems.

A class with high or strong coupling relies on many other classes. Such classes may be undesirable; some suffer from the following problems:

- ♣ 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.

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



SALE CREATES PLACEMENT

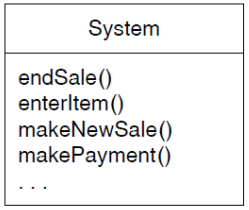
Controller:

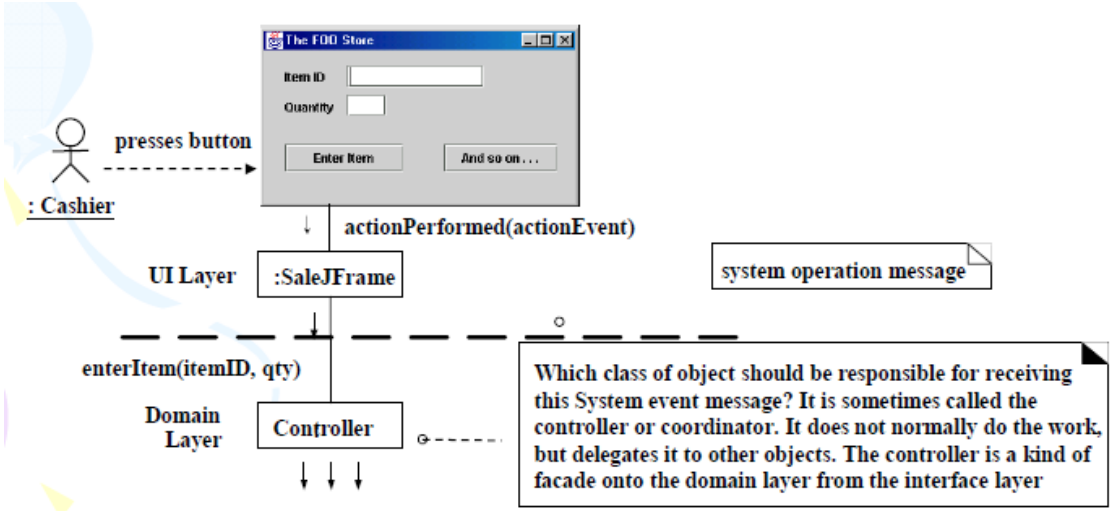
Problem: What first object beyond the UI layer receives and coordinates (controls) a system operation?

A Controller is the first object beyond the UI layer that is responsible for receiving or handling a system operation message.

Solution: assign the responsibility to a class representing one of the following choices:

- ♣ 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.*
- ♣ Represents a use case scenario within which the system event occurs, often named <UseCaseName> Handler, <UseCaseName> Coordinator, or <UseCaseName> Session.
  - → Use the same controller class for all system events in the same use case scenario.
  - → Informally, asession is an instance of a conversation with an actor. Sessions can be of any length but are often organized in terms of use cases.





WHAT OBJECT SHOULD BE THE CONTROLLER FOR ENTERITEM?

5. Explain factory and observer patterns.

Factory Pattern

Consider the problem discussed in the Adapter pattern: who creates the adapters? How to determine which class of adapter to create? If some domain object creates them, the responsibilities of the domain object are going beyond pure application logic and into concerns related to connectivity with external software components.

Goal: Design to maintain a separation of concerns. If a domain object is chosen, a lower cohesion results.

Apply the (concrete) Factory pattern, in which a Pure Fabrication "factory" object is defined to create objects. Thus, we

Separate the responsibility of complex creation into cohesive helper objects

Hide potentially complex creation logic.

Context/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, etc.?

Solution: Create a pure fabrication object called a *Factory* that handles the creation.

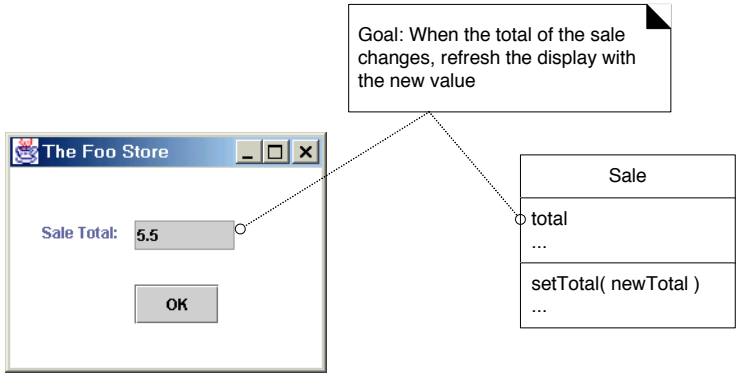
The code for getTaxCalculatorAdapter()

```
{
        if (taxCalculatorAdapter == null)
        {
          String className = System.getProperty("taxcalculator.class.name");
          TaxCalculatorAdapter = (ITaxCalculatorAdapter) Class.forName(className).newInstance();
```

```
        }
        return taxCalculatorAdapter;
    }
```

Observer:

Another requirement for iteration is adding the ability of GUI window to refresh its display of the sale total when the total changes.

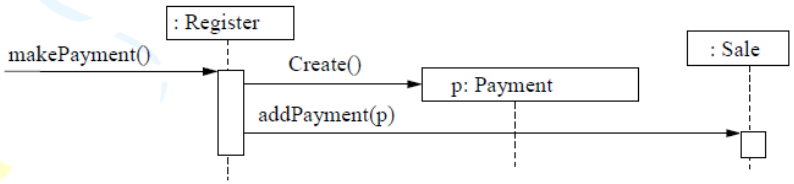


6. Explain the concept of High Cohesion.(May/June 2012)

High Cohesion

Problem: How to keep objects focused, understandable, and manageable and as a side effect, support low coupling?

Cohesion is a measure of how strongly related and focused the responsibilities of an element are. An element with highly 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.



REGISTER CREATES PAYMENT

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 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.

7. Explain the concept of strategy pattern.

Strategy Pattern

Since the behavior of pricing varies by 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-discounted price from the *Sale*, and then apply the discounting rule.

A strategy object is attached to a context object – the object to which it applies the algorithm, e.g., Sale. When a *getTotal* message is send to a *Sale*, it delegates some of the work to its strategy object.

8. Explain the concept of bridge pattern.

The **bridge pattern** is a design pattern used in software engineering which is meant to *"decouple an abstraction from its implementation so that the two can vary independently"*.[1] The *bridge* uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes.

When a class varies often, the features of object-oriented programming become very useful because changes to a program's code can be made easily with minimal prior knowledge about the program. The bridge pattern is useful when both the class and what it does vary often. The class itself can be thought of as the *implementation* and what the class can do as the *abstraction*. The bridge pattern can also be thought of as two layers of abstraction.

When there is only one fixed implementation, this pattern is known as the Pimpl idiom in the C++ world.

The **bridge pattern** is often confused with the adapter pattern. In fact, the **bridge pattern** is often implemented using the class **adapter pattern**, e.g. in the Java code below.

Variant: The implementation can be decoupled even more by deferring the presence of the implementation to the point where the abstraction is utilized.

9. Explain in detail about all types of Design Patterns with an example scenario.

Adapter - Convert the interface of a class into another interface clients expect. / Adapter lets classes work together, that could not otherwise because of incompatible interfaces.
Bridge - Compose objects into tree structures to represent part-whole hierarchies. / Composite
Composite - Compose objects into tree structures to represent part-whole hierarchies. / Composite lets clients treat individual objects and compositions of objects uniformly.
Decorator - add additional responsibilities dynamically to an object.
Flyweight - use sharing to support a large number of objects that have part of their internal state in common where the other part of state can vary.
Memento - capture the internal state of an object without violating encapsulation and thus providing a mean for restoring the object into initial state when needed.

10. Explain polymorphism, pure fabrication and protected variations?

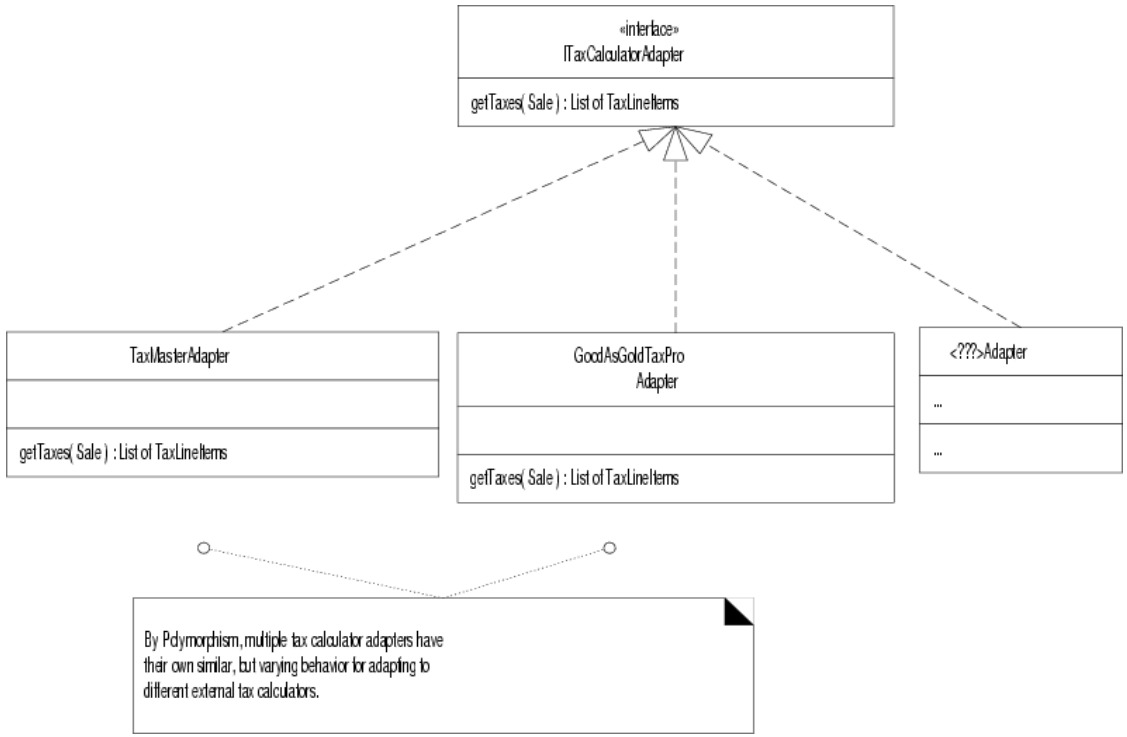
Polymorphism:

Problem: How handle alternatives based on type? How to create pluggable software components?

Alternatives based on type-Conditional variation is a fundamental theme in programs. If a program is designed using if-then-else or case statement conditional logic, then if a new variation arises, it requires modification of the case logic-often in many places. This approach makes it difficult to easily extend a program with new variations because changes tend to be required in several places-wherever the conditional logic exists.

Pluggable software components-Viewing components in client-server relationships, how can you replace one server component with another, without affecting the client.

Solution: 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.



POLYMORPHISM IN ADAPTING TO DIFFERENT EXTERNAL TAX CALCULATORS

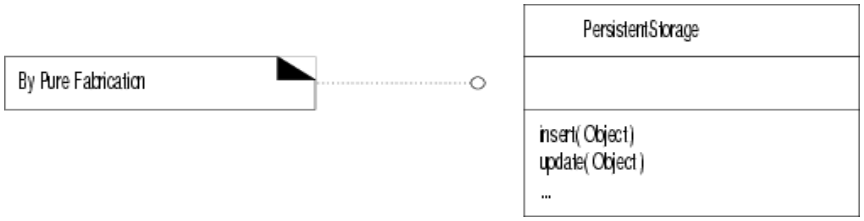
Pure Fabrication:

Problem: What object should have the responsibility, when you do not want to violate high cohesion and low coupling, or other goals, but solutions offered by expert are not appropriate?

OO designs are characterized by implementing as software classes representations of concepts in the real-world problem domain to lower the representational gap. There are many situations in which assigning responsibilities only to domain layer software classes leads to problems in terms of poor cohesion or coupling, or low reuse potential.

Solution: Assign a high cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept-something made up, to support high cohesion, low coupling and reuse.

Such a class is a fabrication of the imagination. The responsibilities assigned to this fabrication support high cohesion and low coupling, so that the design of the fabrication is very clean, or pure-hence pure fabrication.

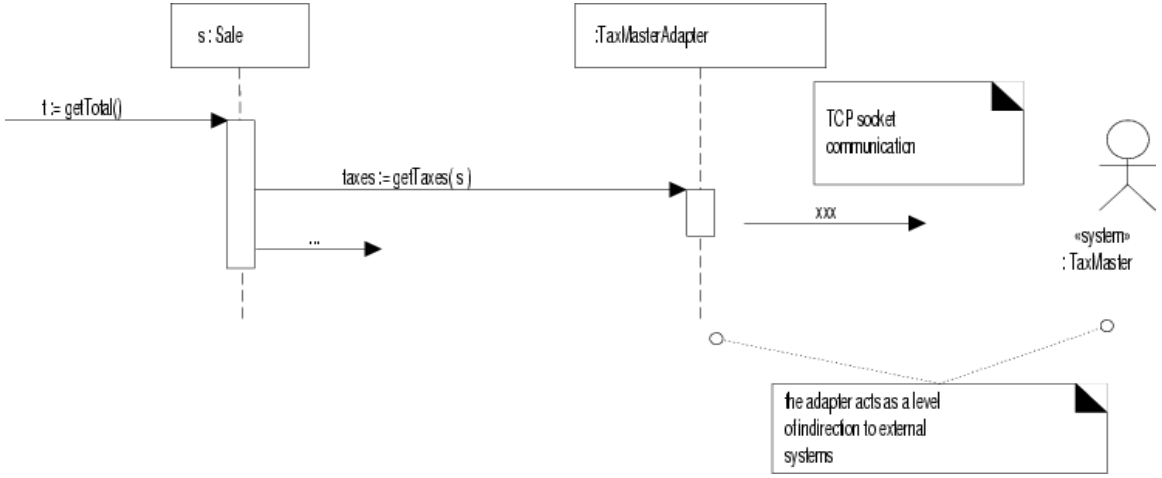


Indirection:

Problem: Where to assign a responsibility, to avoid direct coupling between two or more things?

How to de-couple objects so that low coupling is supported and reuse potential remains higher?

Solution: Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled. The intermediary creates an indirection between the other components.



INDIRECTION VIA THE ADAPTER

Protected Variations:

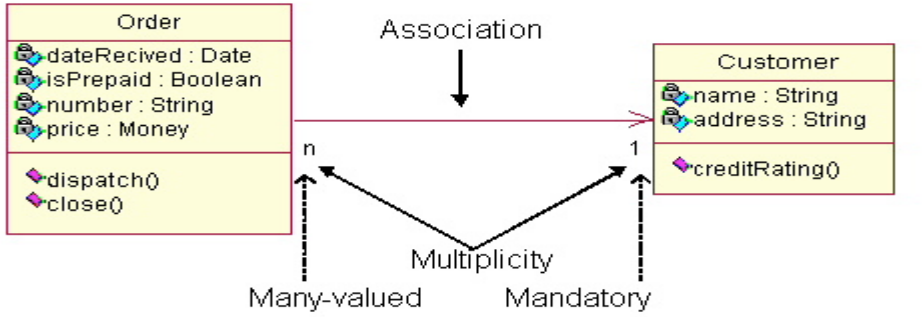
Problem: How to design objects, subsystems and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?

Solution: Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

UNIT –III
PART A

1. What is an association?

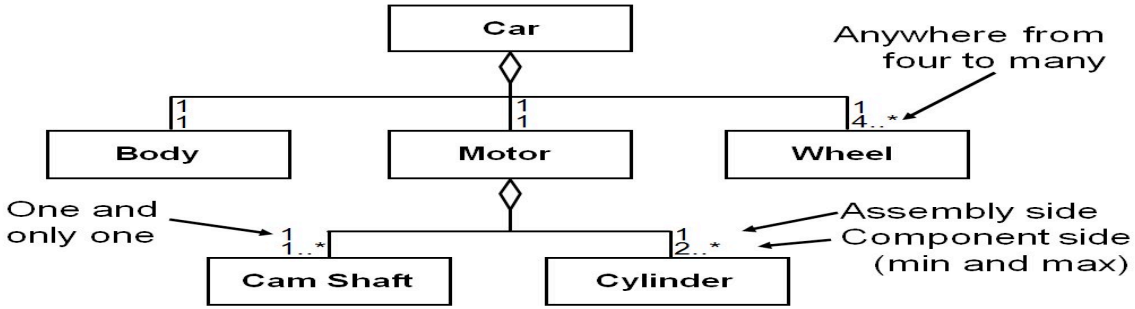
An Association represents a family of links. Binary associations (with two ends) are normally represented as a line, with each end connected to a class box. Higher order associations can be drawn with more than two ends. In such cases, the ends are connected to a central diamond.



2. What is an Aggregation? (Nov/Dec 2013) (May/June 2014)

Aggregation is a variant of the "has a" or association relationship; aggregation is more specific than association. It is an association that represents a part-whole or part-of relationship. As a type of association, an aggregation can be named and have the same adornments that an association can. However, an aggregation may not involve more than two classes.

Aggregation Example



3. **What about attributes in Code?**

The recommendation that attributes in the domain model be mainly datatypes does not imply that C# or Java attributes must only be of simple, primitive datatypes. The domain model is a conceptual perspective, not a software one. In the design model, attributes may be of any type.

4. Define business Modeling

When developing a single application, this includes domain object modeling. When engaged in large scale business analysis or business process reengineering, this include dynamic modeling of the business process across the entire enterprise.

5. Define Elaboration (May/June 2014)

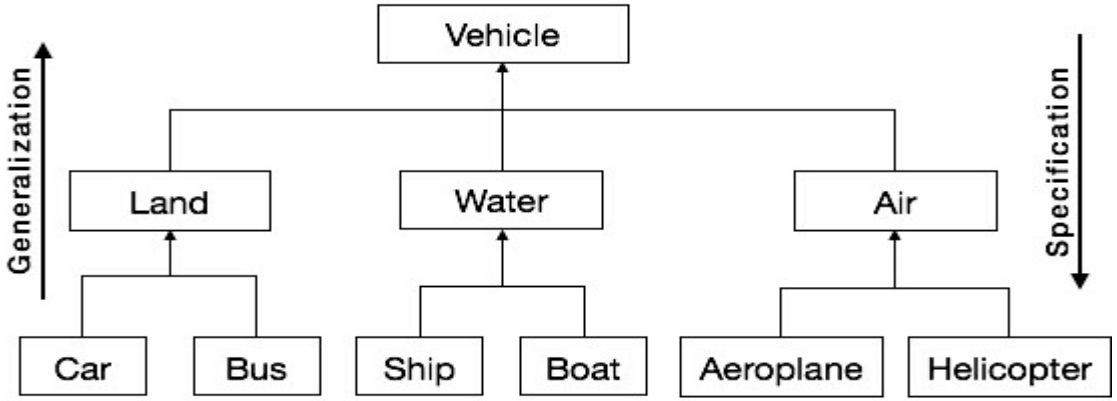
Elaboration is the initial series of iterations during which the team does serious investigation, implements the core architecture clarifies most requriments, and tackles the high –risk issues. Domain model, Design model and data model are the beginning artifacts in elaboration. Achieve a stable version of the majority of the requirements.

6. Define inception step.

Inception is the initial short step to establish a common vision and basic scope for the project. It will include analysis of perhaps 10% of the use cases, analysis of the critical non-functional requirement, creation of a business case, and preparation of the development environment so that programming can start in the following elaboration phase.

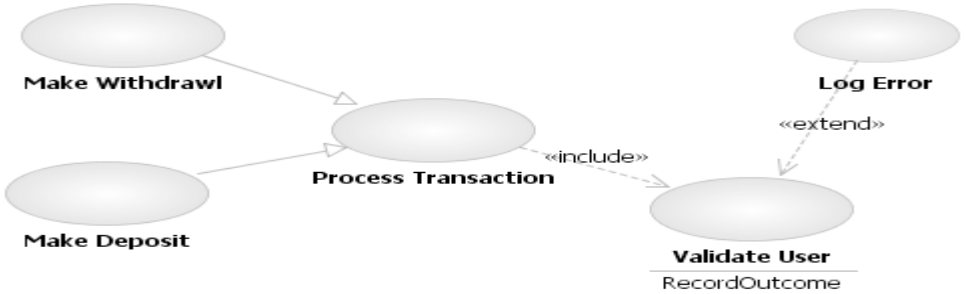
7. What is generalization relationship?

It is a relationship in which one model element (the child) is based on another model element Generalization relationships are used in class, component, deployment, and use-case diagrams to indicate that the child receives all of the attributes, operations, and relationships that are defined in the parent.



8. What is exclude relationship?

In UML modeling, you can use an extend relationship to specify that one use case (extension) extends the behavior of another use case (base).



9. What is composition? (Nov/Dec 2013) (May/June 2014)

Composition is a special type aggregation where the 'has-a' relationship is more strong. I.e. the part entity cannot exist without the whole entity. For example an university has departments which cannot exist on their own with the containing 'university' entity

Composition: every car has an engine.

Aggregation: cars may have passengers, they come and go

10. Define the characteristics of aggregation.

It does not imply ownership. It is an asymmetric relationship. It is a transitive relationship. It implies stronger coupling behavior (copy, delete, etc.). An aggregation is a special form of association that models a whole-part relationship between an aggregate (the whole) and its parts. Aggregation is used to model a compositional relationship between model elements. Containment of the aggregated class is by reference.

11. Differentiate Aggregation and containment.

Aggregation is the relationship between the whole and a part. We can add/subtract some properties in the part (slave) side. It won't affect the whole part.

Best example is Car, which contains the wheels and some extra parts. Even though the parts are not there we can call it as car.

But, in the case of containment the whole part is affected when the part within that got affected. The human body is an apt example for this relationship. When the whole body dies the parts (heart etc) are died.

12. Define Conceptual Classes

• Symbol — words or images representing the concept
• Intention — definition of the concept
• Extension — the set of examples to which the concept applies.

13. How to Create a Domain Model — Three Steps

Find the domain concepts
Draw them in a UML class diagram
Add associations and attribute.

14. List the relationships used in class diagram (Nov/Dec 2014) (May/June 2014)

Objects (of certain class), with
attributes
operations
Links between Objects,
Aggregations between Objects.

15. What is qualified association?

A qualified association has a qualifier that is used to select an object from a larger set of related objects based upon the qualifier key. It reduces the multiplicity at the target end of the association, usually down from many to one because it implies the selection of usually one instance from a larger set.

EX: If a ProductCatalog contains many ProductDescriptions and each one can be selected by an itemID.

16. What are the 3 relationships that can be shown in UML diagram? Define them

1. Association: how are objects associated? This information will guide us in designing classes.

2. Super-Sub structure: How are objects organized into super classes and sub classes? This information provides us the direction of inheritance.

3. Aggregation and a part of structure: What is the composition of complex classes? This information guides us in defining mechanisms that properly manage object within object.

17. What are the advantages of inception?

- Estimation or plans are expected to be reliable.
- After inception, design architecture can be made easily because all the use cases are written in detail.
- The life-cycle objectives of the project are stated, so that the needs of every stakeholder are considered.
- Scope and boundary conditions, acceptance criteria and some requirements are established.

18. What are the three strategies to find conceptual classes?

There are three strategies.
1. Reuse or modify existing models.
2. Use a category list.
3. Identify noun phrases.

19. 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.

20. When are Description Classes useful?

Add a Description Class When:

1. There needs to be a description about an item or service, independent of the current existence of any examples of those items or services.
2. Deleting instances of things they describe results in a loss of information that needs to be maintained.
3. It reduces redundant or duplicated information.

PART B

1. **Explain in detail about the next generation POS system.**

The Case Study:

- The text uses the development of a Point of Sale (POS) System as a case study to demonstrate object oriented software development.
- A POS system is a replacement for a cash register that adds many computer services to the traditional cash register. Most retail stores now use POS systems.

Architectural Layers:

- User Interface
  - *(minor topic)*
- Application Logic and Domain Objects
  - Sale and Payment *(main topics)*
- Technical Services
  - Log and Persistence *(secondary topics)*

Iterative Learning/Development:

- The text is organized in three iterations.
- Each iteration will deliver a product to the customer, with subsequent iterations adding features to the earlier ones.
- Each iteration will do analysis and design only on the features for the current release of the software.

2. **Explain with an example, how use case modeling is used to describe functional requirements.(April/May 2011)**

Contracts may be defined for **system operations** that the system as a black box offers in its public interface to handle incoming system events. System operations can be identified by discovering these system events.

Use-Case Model: System Operation Contracts

No new system operations are being considered in this iteration, and thus contracts are not required. In any event, contracts are just an option to consider when the detailed precision they offer is an improvement over the descriptions in the use cases.

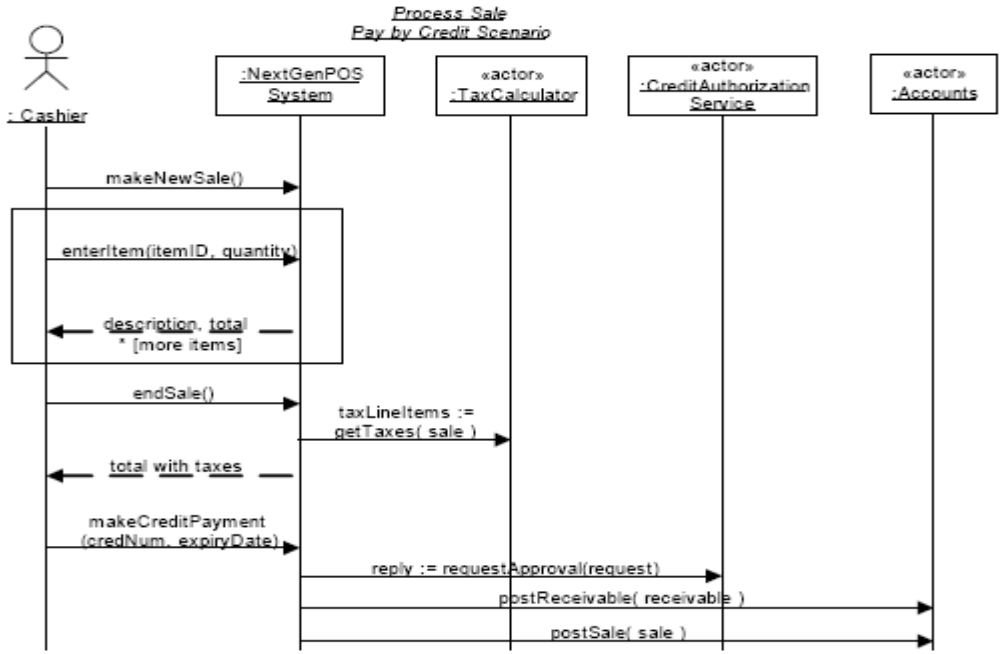


Figure 21.1 An SSD scenario that illustrate some external systems

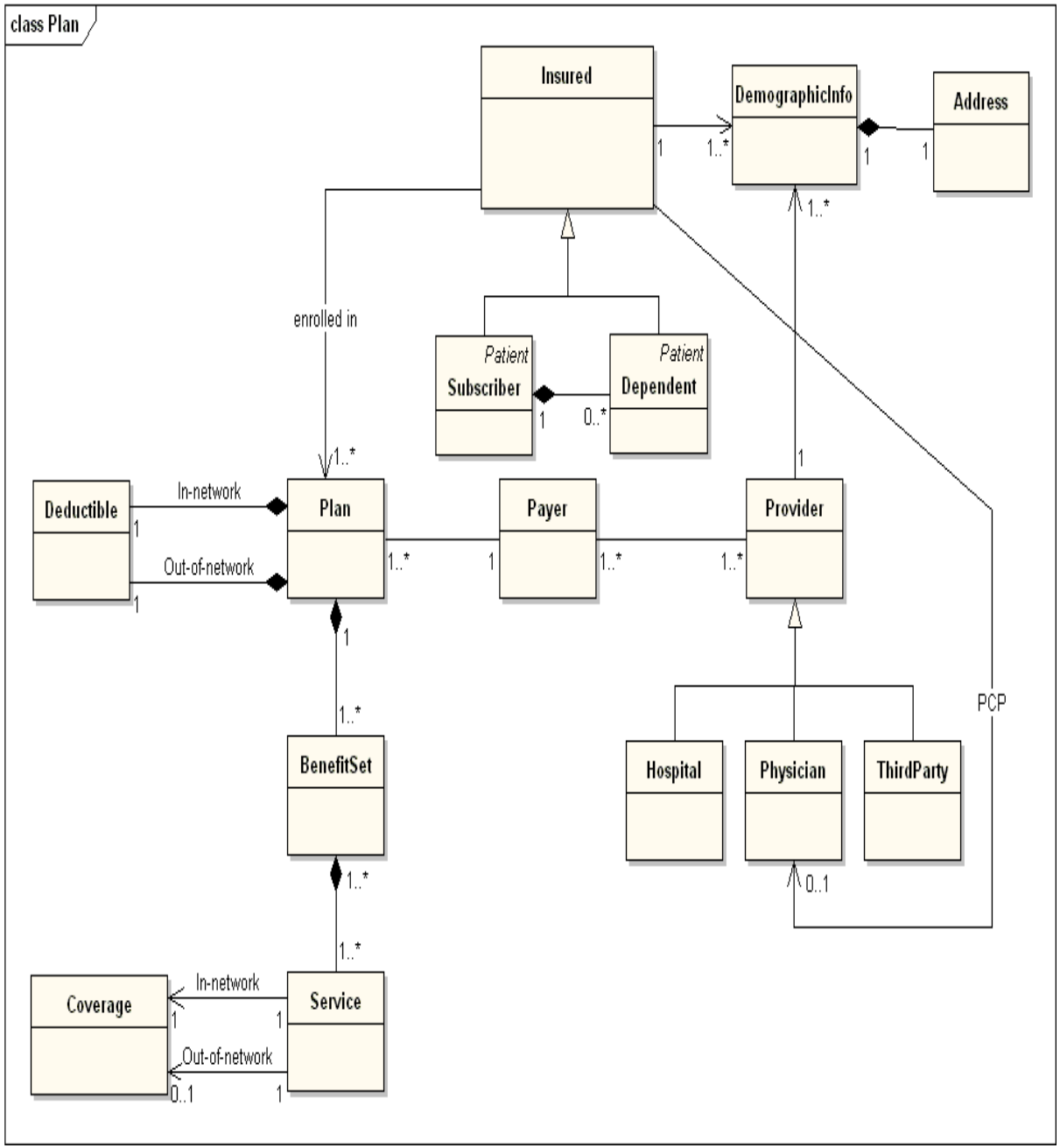
System operations handle input system events.

The entire set of system operations, across all use cases, defines the public system interface, viewing the system as a single component or class. In the UML, the system as a whole can be represented by a class.

3. **Explain the Domain Model and define all the classes involved in the model?**
   A **domain model** in problem solving and software engineering is a conceptual model of all the topics related to a specific problem. It describes the various entities, their attributes, roles, and relationships, plus the constraints that govern the problem domain. It does not describe solutions to the problem.

   The domain model is created in order to represent the vocabulary and key concepts of the problem domain. The domain model also identifies the relationships among all the entities within the scope of the problem domain, and commonly identifies their attributes. A domain model that encapsulates methods within the entities is more properly associated with object oriented models. The domain model provides a structural view of the domain that can be complemented by other dynamic views, such as use casemodels.

An important advantage of a domain model is that it describes and constrains the scope of the problem domain. The domain model can be effectively used to verify and validate the understanding of the problem domain among various stakeholders. It defines a vocabulary and is helpful as a communication tool. It can add precision and focus to discussion among the business team as well as between the technical and business teams.


class Plan
Insured
DemographicInfo
Address
1
1..*
1
1
1..*
enrolled in
Patient
Subscriber
Patient
Dependent
1
0..*
1..*
1
Deductible
In-network
Out-of-network
1
1
Plan
Payer
Provider
1..*
1
1..*
1..*
1
1..*
BenefitSet
1..*
1..*
Hospital
Physician
ThirdParty
PCP
0..1
Coverage
In-network
Out-of-network
1
1
0..1
1
Service

Sample domain model for a health insurance plan

A well thought-out domain model serves as a clear depiction of the conceptual fabric of the problem domain and therefore is invaluable to ensure all stakeholders agree on the scope and meaning of the concepts in the problem domain. An accurate domain model can also serve as an essential input to solution implementation within a software development cycle since the model elements comprising the problem domain can serve as key inputs to code construction, whether that construction is achieved manually or through automated code generation approaches. It is important, however, not to compromise the richness and clarity of the business meaning depicted in the domain model by expressing it directly in a form influenced by design or implementation concerns.

The domain model is one of the central artifacts in the project development approach called feature-driven development (FDD).

In domain-driven design, the Domain Model (domain entities and actors) covers all layers involved in modelling a business domain, including (but not limited to) Service Layer,Business Layer, and Data Access Layer thus ensuring effective communication at all levels of engineering. It is considered an effective tool for software development, especially when domain knowledge is iteratively provided by domain experts (such as Business Analysts, Subject Matter Experts and Product Owners.)

In UML, a class diagram is used to represent the domain model.

"A domain model captures the most important types of objects in the context of the business. The domain model represents the 'things' that exist or events that transpire in the business environment." – I. Jacobsen

Why do a domain model?

Gives a conceptual framework of the things in the problem space

Helps you think – focus on semantics

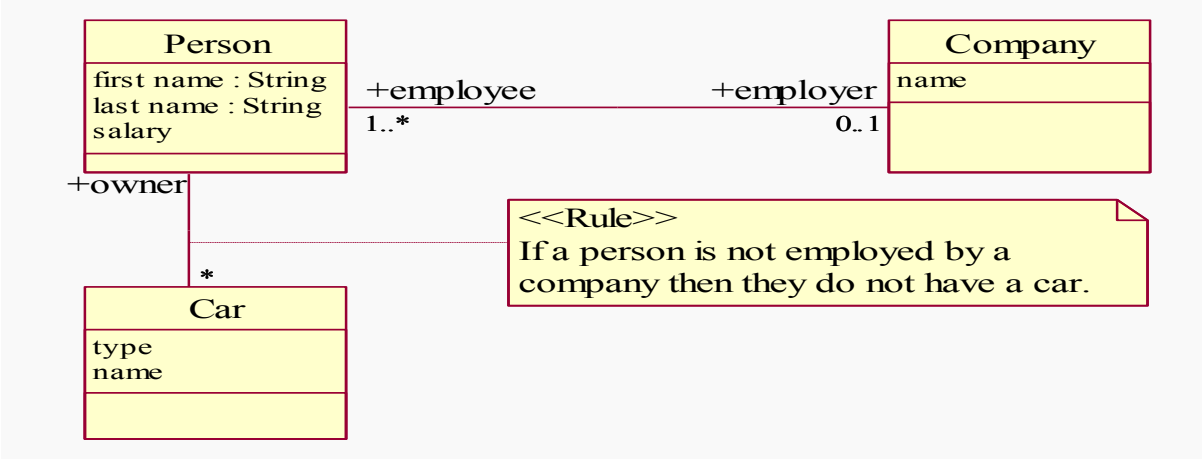
Provides a glossary of terms – noun based

It is a static view - meaning it allows us convey time invariant business rules

Foundation for use case/workflow modelling

Based on the defined structure, we can describe the state of the problem domain at any time.

Simple domain model



Features of a domain model

The following features enable us to express time invariant static business rules for a domain:-

Domain classes – each domain class denotes a type of object.

Attributes – an attribute is the description of a named slot of a specified type in a domain class; each instance of the class separately holds a value.

Associations – an association is a relationship between two (or more) domain classes that describes links between their object instances. Associations can have roles, describing the multiplicity and participation of a class in the relationship.

Additional rules – complex rules that cannot be shown with symbology can be shown with attached notes.

DOMAIN CLASSES:

Each domain class denotes a type of object. It is a descriptor for a set of things that share common features. Classes can be:-

Business objects - represent things that are manipulated in the business e.g. *Order*.

Real world objects – things that the business keeps track of e.g. *Contact*, *Site*.

Events that transpire - e.g. *sale* and *payment*.

A domain class has attributes and associations with other classes (discussed below). It is important that a domain class is given a good description

How do I make a domain model?

Perform the following in very short iterations:

Make a list of candidate domain classes.

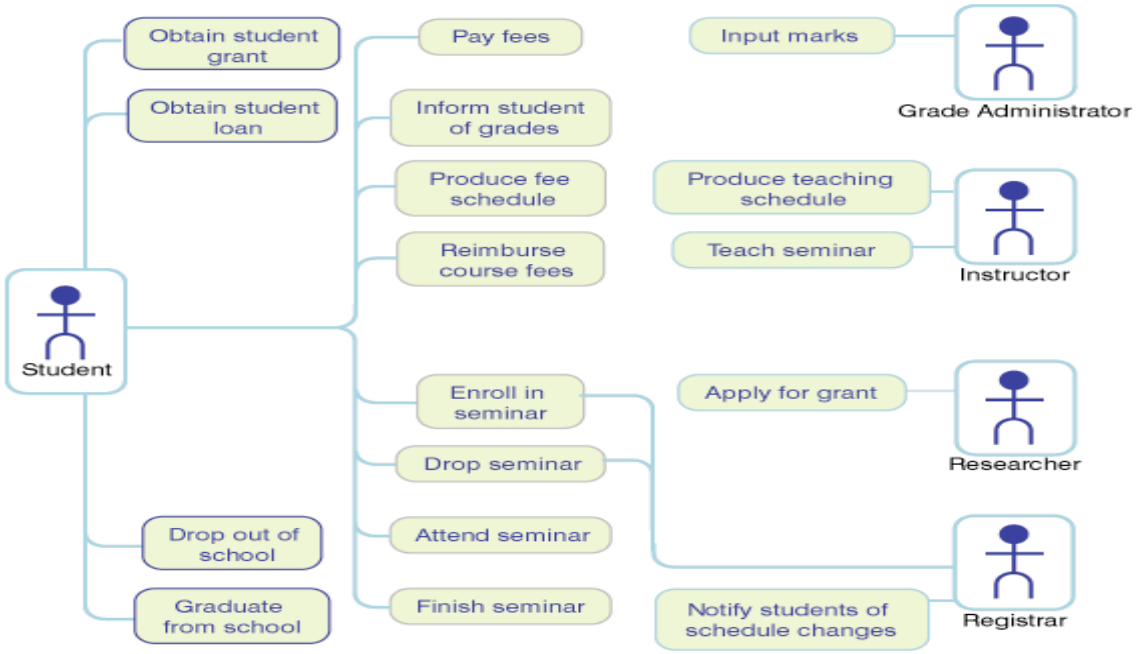
Draw these classes in a UML class diagram.

If possible, add brief descriptions for the classes.

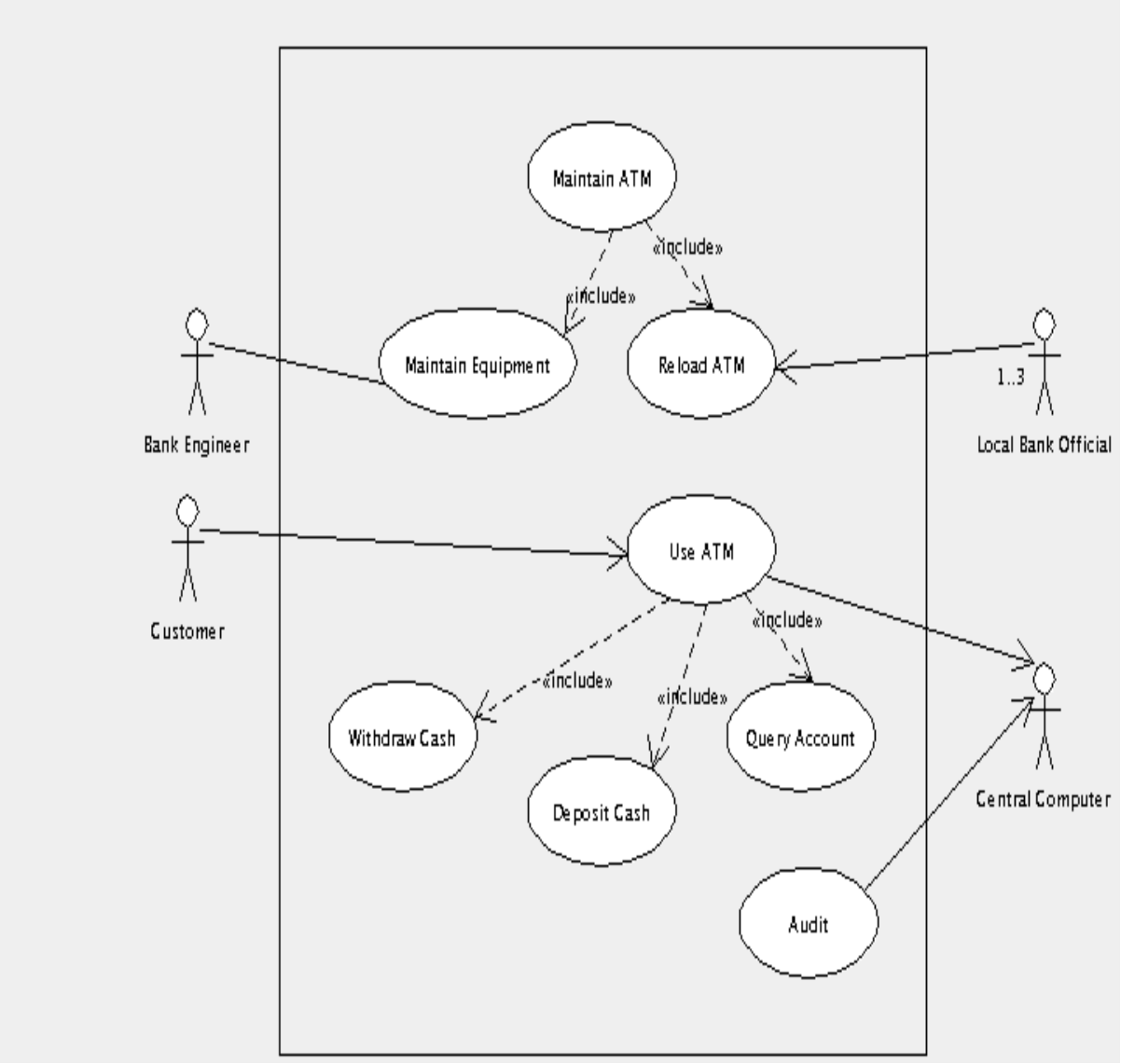
Identify any associations that are necessary.
Decide if some domain classes are really just attributes.
Where helpful, identify role names and multiplicity for associations.
Add any additional static rules as UML notes that cannot be conveyed with UML symbols.
Group diagrams/domain classes by category into packages.
Concentrate more on just identifying domain classes in early iterations !
Identifying domain classes?
An obvious way to identify domain classes is to identify nouns and phrases in textual descriptions of a domain.
Consider a use case description as follows:-

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

3. UML activity diagram for the Enroll in University use case.



4. Generate a Use Cases diagram for Example ATM System and explain?



4. Explain about Association and different types about associations.
   Aggregation is a kind of association used to model whole-part relationships between things.

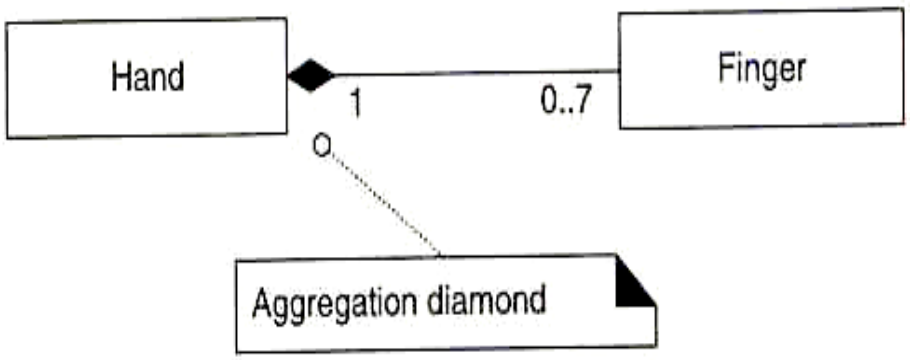
The whole is called **Composite**.

For instance, physical assemblies are organized in aggregation relationships such as a *Hand* aggregates *Fingers*.

Aggregation is shown in the UML with a hollow or filled diamond symbol at the composite end of whole-part association.

Aggregation is a property of an association role.

Aggregation Notation



Composite Aggregation – Filled Diamond

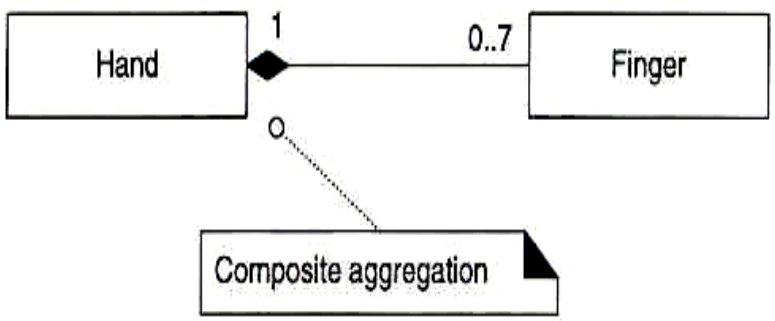
Means that the part is a member of only one composite object, and that there is an existence and disposition dependency of the part on the composite.

Composition is signified with a filled diamond.

It implies that the composite solely owns the part, and that they are in a tree structure part hierarchy;it is the most common form of aggregation shown in models.

Example Composition Aggregation

A finger is a part of at most one hand (we hope!), thus the aggregation diamond is filled to indicate composite aggregation.



Composite Aggregation : In The Design Model
Composition and its existence dependency implication indicates that composite software objects create the part software objects.

For example, *Sale* creates *SalesLineItem*.

Composite Aggregation: In The Domain Model
It does not represent software objects, the notion of the whole creating the part is seldom relevant (a real sale does not create a real sales line item).

However, there is still an analogy.

For Example, in a "human body" domain model,one thinks of the hand as including the fingers, so if one says, "A hand has come into existence," we understand this to also mean that fingers have come into existence as well.

Multiplicity At Composite End
If the multiplicity at the composite end is exactly one, the part may not exist separate from some composite.

For Example : if the finger is removed from the hand, it must be immediately attached to another composite object (another hand, a food,…); at least, that is what the model is declaring, regardless of the medical merits of this idea!

If the multiplicity at the composite end is 0…1,then the part may be removed from the composite, and still exist apart from membership in any composite.

So in previous example if you want fingers floating around by themselves, use 0…1.

Shared Aggregation – Hollow Diamond
Means that the multiplicity at the composite end may be more than one.

It is signified with Hollow Diamond.

Implies that the part may be simultaneously in many composite instances.

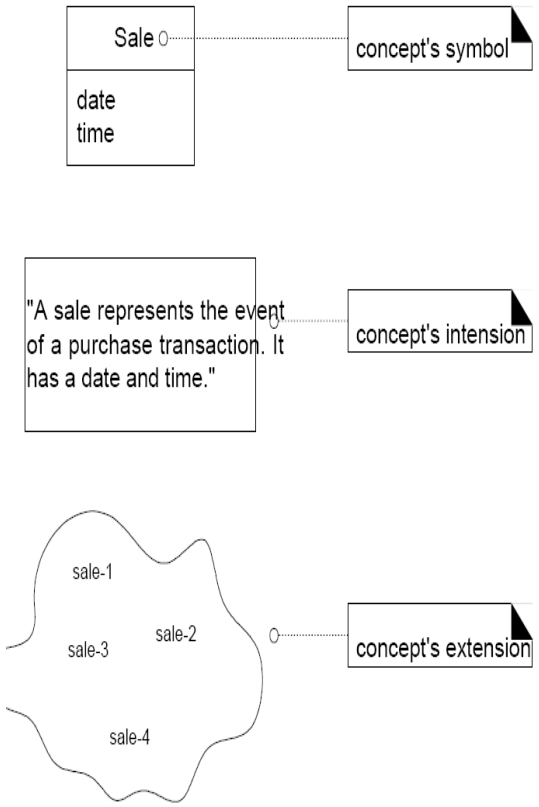
Shared aggregation seldom exists in physical aggregates, but rather in nonphysical concepts.

Shared Aggregation : Example

A UML package may be considered to aggregate its elements. But an element may be referenced in more than one package.

5. **How to find conceptual classes?(April/May 2011)**
   Conceptual Classes
   Informally, a conceptual class is an idea, thing, or object.
   More formally, a conceptual class may be considered in terms of its symbol, intension, and extension
   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.We may choose to name it by the symbol *Sale.* The intension of a *Sale* maystate that it "represents the event of a purchase transaction, and has a date andtime." The extension *of Sale* is all the examples of sales; in other words, the setof all sales.



A conceptual class has a symbol, intension, and extension.

The three strategies to find conceptual classes

Reuse or modify existing models(First,Best,and easiest approach). There are published ,well crafted domain models and data models for many common domains ,such as inventory,finance,health,banking,and so forth.

Use a Category List

Identify noun phrases.

Conceptual Class Category List We can kick start the creation of a domain model by making a list of candidate conceptual classes. The following table contains many common categories (which are usually worth considering as meeting business information needs)

6. Explain the concept of description classes?

Creating domain model

- *What might these be for the library conceptual classes? Monopoly?*
- *Issues*
- *Naming: try to use the names stakeholders use*
- *Attributes vs. conceptual classes*
  - Unless concept is just a number or a string in the real world, represent as a conceptual class
  - Ex: "store" may only have a name in your model
- *Not all concepts have a tangible real-world analog*
  - Ex: message, OS process

Description concepts

- *Metadata: describes or explains other data*
- *Examples*
  - object-oriented class definition
  - attribute list: ex: product has price, ID, description, etc.
  - describing artifacts for your project
- *Maintains information even if all instances are deleted*
- *Can reduce redundant information*

Description concepts

- *Where might this be useful for a library?*
  - Definition of library's cataloging system
  - For each category/subcategory in the Library of Congress system, what is the subject? Where is the book located?

Associations

- *Association = relationship between instances of 2 classes*

- *Too many associations make domain model confusing*
- *Many, but not all, associations affect implementation*
- *Need to remember the association for more than an instant*
  - Ex: Location of player's piece in Monopoly is essential (save until the next turn)
  - Ex: The dice roll determines where to move, but can be forgotten immediately

Associations

- Multiplicity *(like cardinality in an entity-relationship diagram)*
  - How many object instances are involved in one instance of the relationship
  - Ex: Class Section is held in Classroom
    - More than one section is held in the same classroom
    - We may want a section to use only one classroom (for scheduling), or may need to allow multiple classrooms (ex: labs)

Associations

- *Can also find by using standard list of categories for associations*
  - Ex: A is a physical or logical part of B
    - Drawer is part of Register (PoS) -- 1 to 1 (not needed for first iteration of PoS)
    - Square is part of Board (Monopoly) -- ?
    - Seat is part of Airplane -- ?

Associations

  - Ex: A and B are related transactions
    - Sale is paid by Cash payment (PoS) -- (see fig. 9.17)
    - A Cancellation cancels a Reservation
  - Ex: A uses/manages/owns B
    - Cashier works on Register(PoS)
    - Player owns Piece (Monopoly)
    - Pilot flies Airplane

Attributes

- *Indicate information required by requirements*
  - Should be of a primitive/built-in type (object attributes should be associations)

- *Ex: (Monopoly)*
  - Die has faceValue
  - Player, Piece, and Square each has name attribute

Attributes

- *Ex: (PoS) For ProcessSale use case, Product description should include:*
  - itemID (for searching for item purchased)
  - price (for receipt total)
  - description (include on receipt, on register's display)

7. **Explain in detail about domain model refinement?**

   Objectives

   Add association classes to the Domain Model.

   Add aggregation relationships.

   Model the time intervals of applicable information.

   Choose how to model roles.

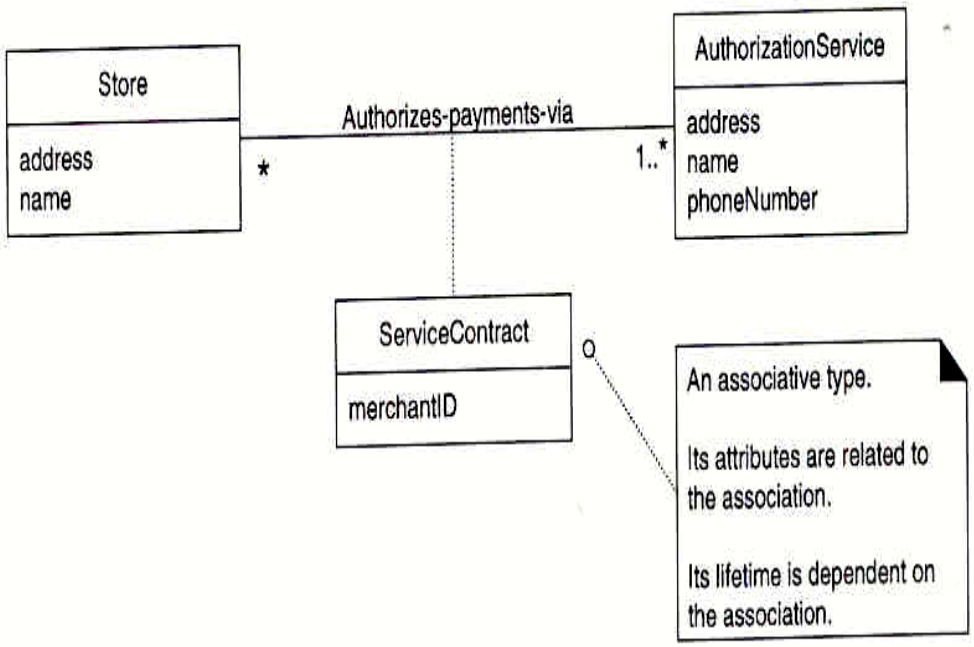
   Organize the Domain Model into packages.

   Association Classes

   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 *AuthorizationSerice*.

   In the UML,this is illustrated with a dashed line from the association to the association class

   This figure visually communicates the idea that a *ServiceContract* and its attributes are related to the association between *Store* and *AuthorizationService,* and that the lifetime of the *ServiceContract* is dependent on the relationship.

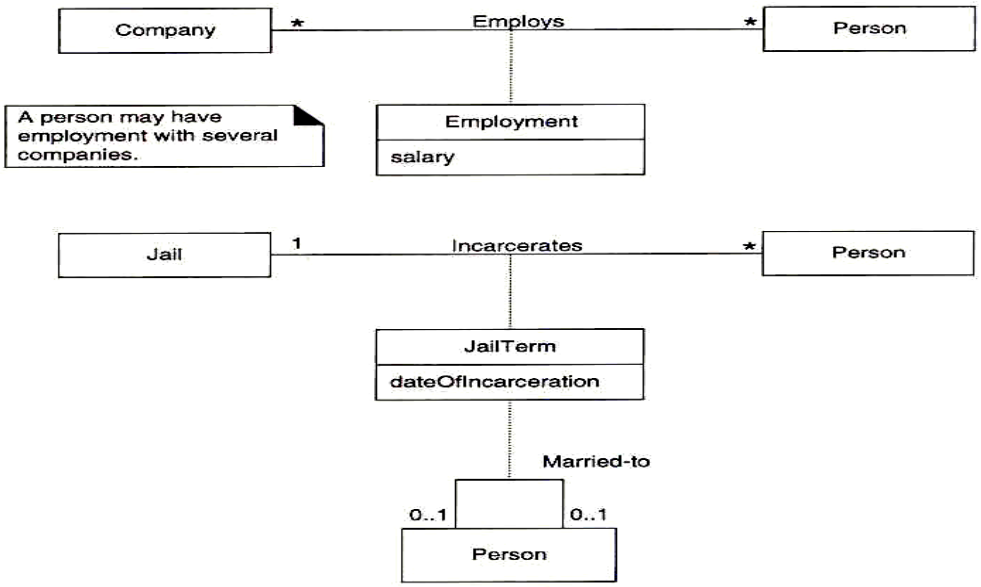


Guidelines For Adding Association Classes

An attribute is related to an association.

Instances of the association class have a life-time dependency on the association.

There is a many-to-many association between two concepts, and information associated with the association itself.



Roles In Association

They are a relatively accurate way to express the notation that the same instance of a person takes on multiple (and dynamically changing) roles in various association.
I , a person, simultaneously or in sequence, may take on the role of writer, object designer, parent, and so on.

Roles As Concepts

Provides ease and flexibility in adding unique attributes, associations, and additional semantics.
The implementation of roles as separate classes is easier because of limitations of current popular object – oriented programming languages ,
it is not convenient to dynamically mutate an instance of one class into another, or dynamically add behavior and attributes as the role of a person changes.

Derived Elements

Can be determined from others.
Attributes and associations are most common derived elements.

Guidelines For Derived Elements

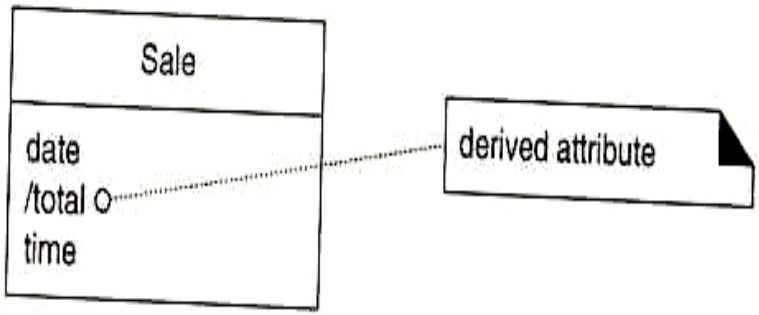
Avoid showing derived elements in a diagram, since they add complexity without new information.

However, add a derived element when it is prominent in the terminology, and excluding it impairs comprehension.

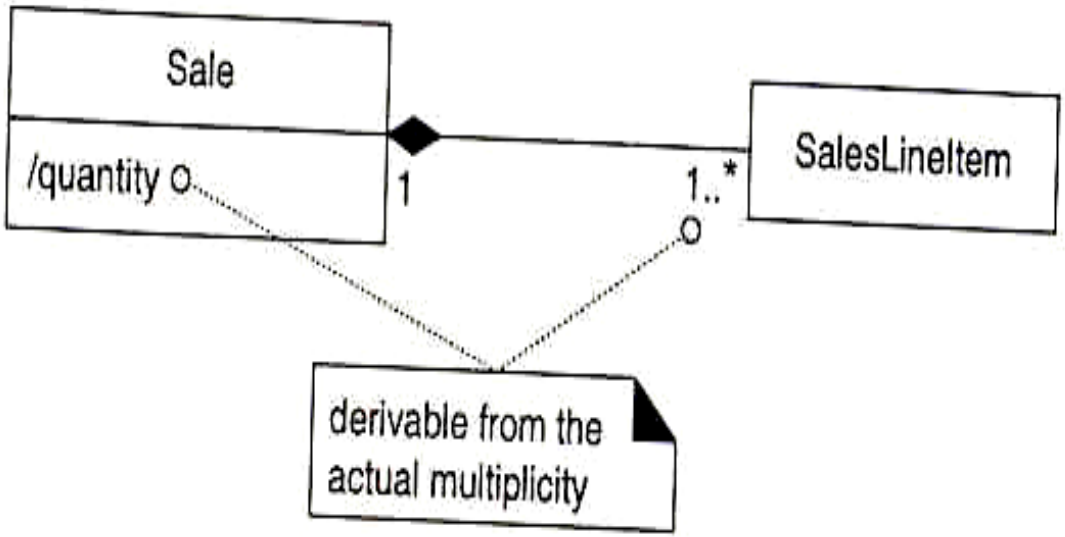
Derived Elements – Example

A Sale *total* can be derived from *SalesLineItem* and *ProductSpecification* information.

In the UML, it is shown with a "/" preceding the element name.



Derived Attribute Related To Multiplicity

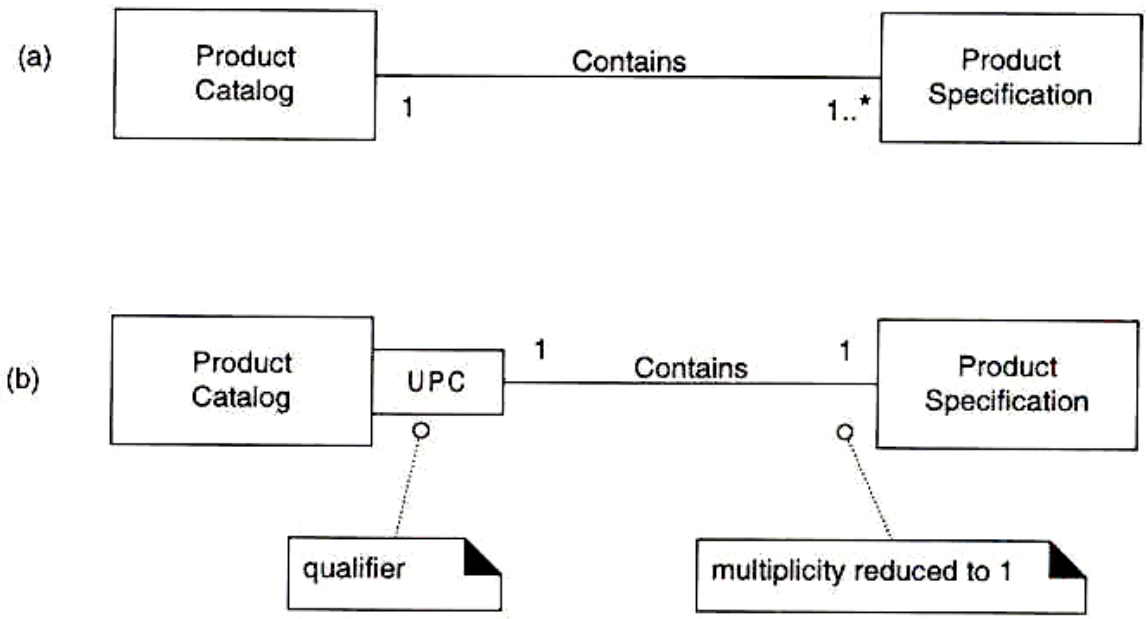


Qualified Associations

A **qualifier** may be used in association; it distinguishes the set of objects at the far end of the association based on the qualifier value.

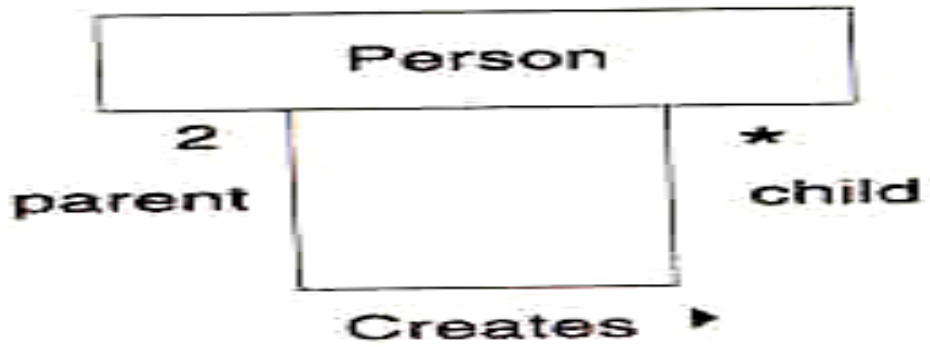
An association with a qualifier is a **qualified association**.

For example, *ProductSpecifications* may be distinguished in a *ProductCatalog* by their ItemID.



Reflexive Associations

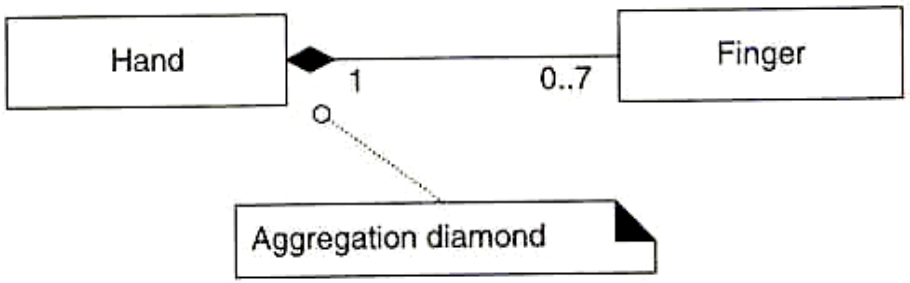
A concept may have an association to itself; this is know as a reflexive association.



8. Discuss the concept of aggregation and composition?

- **Aggregation** is a kind of association used to model whole-part relationships between things.
- The whole is called **Composite**.
- For instance, physical assemblies are organized in aggregation relationships such as a *Hand* aggregates *Fingers*.
- Aggregation is shown in the UML with a hollow or filled diamond symbol at the composite end of whole-part association.
- Aggregation is a property of an association role.

Aggregation Notation

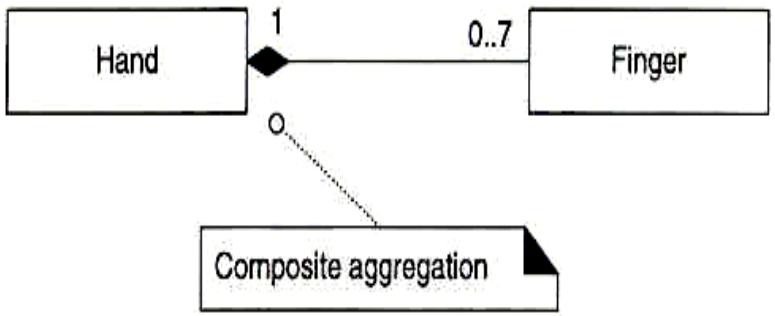


Composite Aggregation – Filled Diamond

- Means that the part is a member of only one composite object, and that there is an existence and disposition dependency of the part on the composite.
- Composition is signified with a filled diamond.
- It implies that the composite solely owns the part, and that they are in a tree structure part hierarchy;it is the most common form of aggregation shown in models.

Example Composition Aggregation

A finger is a part of at most one hand (we hope!), thus the aggregation diamond is filled to indicate composite aggregation.



Composite Aggregation : In The Design Model

- Composition and its existence dependency implication indicates that composite software objects create the part software objects.
- For example, *Sale* creates *SalesLineItem*.

Composite Aggregation: In The Domain Model

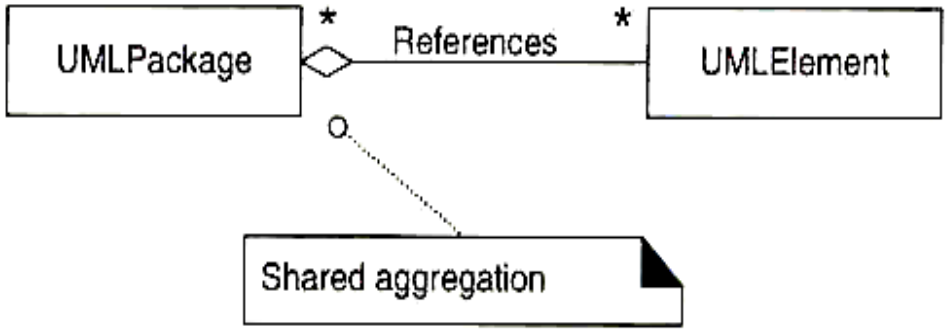
- It does not represent software objects, the notion of the whole creating the part is seldom relevant (a real sale does not create a real sales line item).
- However, there is still an analogy.
- For Example, in a "human body" domain model,one thinks of the hand as including the fingers, so if one says, "A hand has come into existence," we understand this to also mean that fingers have come into existence as well.

Multiplicity At Composite End

- If the multiplicity at the composite end is exactly one, the part may not exist separate from some composite.
- For Example : if the finger is removed from the hand, it must be immediately attached to another composite object (another hand, a food,…); at least, that is what the model is declaring, regardless of the medical merits of this idea!
- If the multiplicity at the composite end is 0…1,then the part may be removed from the composite, and still exist apart from membership in any composite.
- So in previous example if you want fingers floating around by themselves, use 0…1.

Shared Aggregation – Hollow Diamond

- Means that the multiplicity at the composite end may be more than one.
- It is signified with Hollow Diamond.
- Implies that the part may be simultaneously in many composite instances.
- Shared aggregation seldom exists in physical aggregates, but rather in nonphysical concepts.
- **Shared Aggregation : Example**
- A UML package may be considered to aggregate its elements. But an element may be referenced in more than one package.



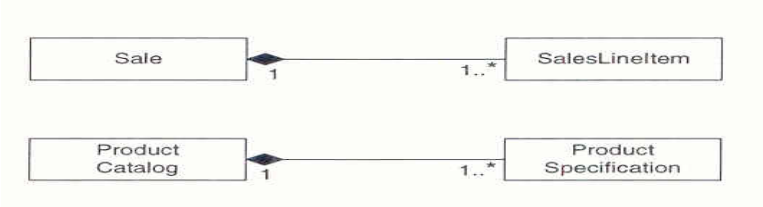
Guidelines For When To Show Aggregation

- 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.
- Operation applied to the composite propagate to the parts, such as destruction, movement, recording.

A Benefit Of Showing Aggregation

- It clarifies the domain constraints regarding the eligible existence of the part independent of the whole. In composite aggregation, the part may not exist outside the lifetime of the whole.
  - During design work, this has an impact on the create – delete dependencies between the whole and part software classes and database elements (in terms of referential integrity and cascading delete paths.

Aggregation In POS Application



9. **Describe the strategies used to identify conceptual classes. Describe the steps to create a domain model used for representing conceptual classes. (Nov/Dec 2013) (May/June 2014)**

Conceptual Classes

Informally, a conceptual class is an idea, thing, or object.

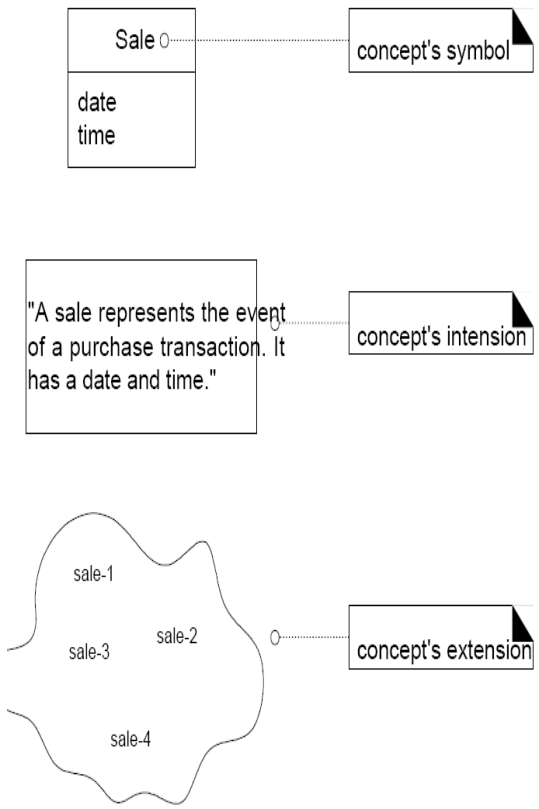
More formally, a conceptual class may be considered in terms of its symbol, intension, and extension

- 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. We may choose to name it by the 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 sales.



A conceptual class has a symbol, intension, and extension.

The three strategies to find conceptual classes

a) Reuse or modify existing models(First,Best,and easiest approach). There are published ,well crafted domain models and data models for many common domains ,such as inventory,finance,health,banking,and so forth.
b) Use a Category List
c) Identify noun phrases.

Conceptual Class Category List We can kick start the creation of a domain model by making a list of candidate conceptual classes. The following table contains many common catgories(which are usually worth considering as meeting business information needs)

Conceptual Class Category	Examples
physical or tangible objects	*Register* *Airplane*
specifications, designs, or descriptions of things	*ProductSpecification* *FlightDescription*
places transactions	*Store* *Airport* *Sale, Payment* *Reservation*
transaction line items	*SalesLineItem*
roles of people containers of other things	*Cashier* *Pilot* *Store, Bin* *Airplane*

10. **Write briefly about elaboration and discuss the differences between elaboration and inception with examples(May/June 2014)**

Elaboration:

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. In the UP, "risk" includes business value. Therefore, early work may include implementing scenarios that are deemed important, but are not especially technically risky.

Tasks performed in elaboration:

the core, risky software architecture is programmed and tested the majority of requirements are discovered and stabilized the major risks are mitigated or retired

key ideas and best practices that 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

Artifacts May Start in Elaboration:

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	Descriptions of the user interface, paths of navigation, usability models, and so forth.

Inception Vs. Elaboration

Inception lays the groundwork for the project

1st inception iteration:

we are starting from zero

So…there is no preexisting project state to refine

hence the concept of incremental change doesn't apply yet

somewhat unique because

typically there is only 1 iteration in inception phase

some people don't even call it an iteration

The Concept of First Iterations

Inception:

often 1 iteration long…

and it’s the 1st iteration

Like in program loops, the first iteration is different

A mathematical analysis:

Assume all iterations have equal effect

Iteration 11 would change the project by 10% What about iteration 3? 2? 1?

There seems to be a broader truth here…

Inception Vs. Elaboration II

Iterations in the elaboration phase:

They gradually seem more and more routine

Why, mathematically speaking?

But when they finally are routine…

we no longer call it “elaboration”

(…it’s construction – see next figure)

Implementation begins in the elaboration phase

(What is implementation?)

Key UP idea: after each elaboration iteration