CS2353 OBJECT ORIENTED ANALYSIS AND DESIGN

UNIT I

- **Unit one covers a host of many topics central to today's technologies.**
- **These skills are essential in today's professional community.**
- **We will talk about the Unified Process, agile approaches (replace high-level design with frequent redesign) UML,**
- **Later on we will advance into more complex concepts that address framework design and architectural analysis.**

OBJECTS

Knowing an object oriented language is a necessary but insufficient to create object systems.

Knowing how to "thing in objects" is critical.

The proverb "owing a hammer doesn't make one an architect" is especially true with respect to object technology.

Analysis: "build the right thing"

- **Analysis focuses on user requirements (functional or non-functional)**

Design: "build the thing right"

- **Design focuses on how to provide the required functionality.**

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 (or concepts) in the problem domain

Analysis

Usually performed by a systems analyst.

- **This is a person whose job it is to find out what an organization (or person) needs in terms of a software system.**

Analysis looks at the software as a black box of functionality.

- **An analyst will never care about the internals of the system, merely how a user would interact with it from the outside.**

Design

Usually performed by an architect or designer

- **An architect looks at overall system structure (at a high level)**

  For example, an architect on a distributed system might decide how the software components will be distributed. An architect also looks at modularity, and non-functional requirements (such as performance and scalability)

  A designer looks at system structure (at lower levels)

  A designer will look at the classes that make up a module, and how they will interact to perform some function of the system

Design Patterns

Patterns are themes that recur in many types of software systems

- **It is common for software systems (even those whose purpose is quite different) will share some common challenges. Often, the solution for the problem in one context can also be used in another context. Thus, the designs (and even implementations) can often be reused from other software systems**
-

Thinking in Objects and UML

The Unified Modeling Language (UML) is a standard diagramming notation; sometimes referred to as a blueprint. It is NOT OOA/OOD or a method. Only a notation for capturing objects and the relationships among objects (dependency; inheritance; realizes; aggregates,). UML is language-independent

Analysis and design provide software "blueprints" captured in UML. Blueprints serve as a tool for thought and as a form of communication with others. But it is far more essential to 'think' in terms of objects as providing 'services' and accommodating 'responsibilities.'

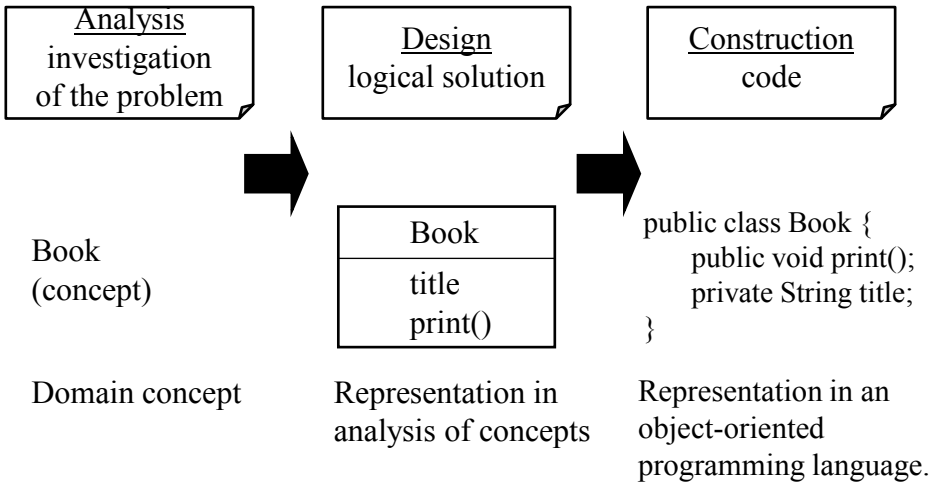
Object-Oriented Analysis (Overview)

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 (or concepts) in the problem domain. For example, concepts in a Library Information System include *Book*, and *Library*. High level views found in the application domain. Often called domain objects; entities.

Object-Oriented Design

Emphasizes a conceptual solution that fulfills the requirements.
Need to define software objects and how they collaborate to meet the requirements.
For example, in the Library Information System, a *Book* software object may have a *title* attribute and a *getChapter* method. What are the methods needed to process the attributes? Designs are implemented in a programming language. In the example, we will have a *Book* class in Java.
From Design to Implementation

Thinking in Terms of Objects and UML – 5
From Design to Implementation



Can you see the services / responsibilities in the Book class? [13]

Of course, design (solution to requirements) 'assumes' a robust requirements analysis has taken place. Use Cases are often used to capture stories of requirements and are often views as 'constituting' the functional requirements, but NOT the software quality factors (non-functional requirements). Use Cases are not specifically designed to be object-oriented, but rather are meant to capture how an application will be used. Many methods for capturing requirements. We will concentrate on Use Cases.

Our Approach:

We need a Requirements Analysis approach with OOA/OOD need to be practiced in a framework of a development process. We will adopt an agile approach (light weight, flexible) in the context of the Unified Process, which can be used as a sample iterative development process. Within this process, the principles can be discussed.

Why the Unified Process:

The Unified Process is a popular iterative software development process. Iterative and evolutionary development involves relatively early programming and testing of a partial system, in repeated cycles. It typically also means that development starts before the exact software requirements have been specified in detail; Feedback (based on measurement) is used to clarify, correct and improve the evolving specification:

This is in complete contrast to what we usually mean by engineering!

What is the Unified Process?

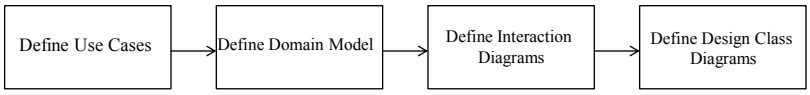
The UP is very flexible and open and can include other practices from other methods such as Extreme Programming (XP) or Scrum for example. e.g. XP's test-driven development, refactoring can fit within a UP project; So can Scrum's daily meeting. Being pragmatic in adapting a particular process to your needs is an important skill: all projects are different.

The Rush to Code

Critical ability to develop is to think in terms of objects and to artfully assign responsibilities to software objects. Talk at great length about encapsulation and assigning methods to objects where the data is defined. One cannot design a solution if the requirements are not understood. One cannot implement the design if the design is faulty.

- **Analysis: - investigate the problem and the requirements.**
  - **What is needed? Required functions? Investigate domain objects.**
  - **Problem Domain**
  - **The Whats of a system.**
  - **Do the right thing (analysis)**
- **Design:**
  - **Conceptual solution that meets requirements.**
  - **Not an implementation**
  - **E.g. Describe a database schema and software objects.**
  - **Avoid the CRUD activities and commonly understood functionality.**
  - **The Solution Domain**
  - **The 'Hows' of the system**
  - **Do the thing right (design)**
- **OOA: we find and describe business objects or concepts in the problem domain**
- **OOD: we define how these software objects collaborate to meet the requirements.**
  - **Attributes and methods.**
- **OOP: Implementation: we implement the design objects in, say, Java, C++, C#, etc.**

- Using the model below, develop a discussion outlining the four activities listed and present the major features of each.



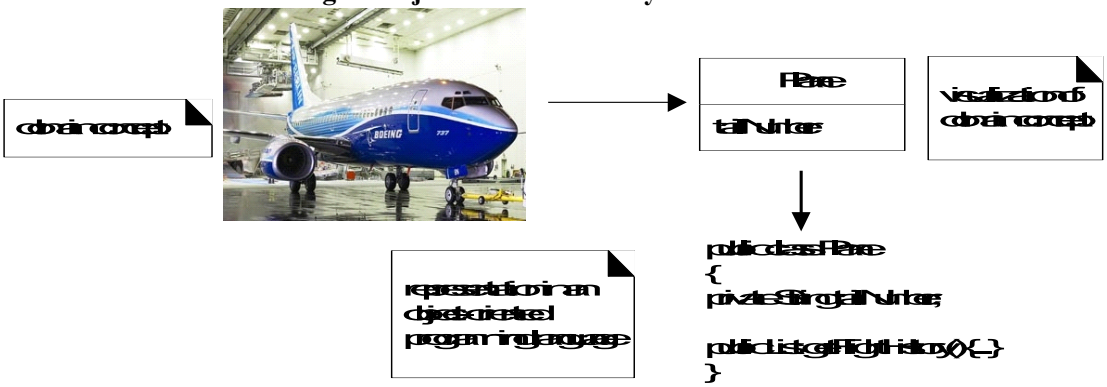
A short definition and example of a domain model, interaction diagram, and class diagram is sufficient, but be prepared to discuss each of these.

Also, have a general idea about use cases – what they are designed to do and what they are not designed to do.

What is Analysis and Design

- **Analysis:**
  - **investigate problem and is requirements; solution comes later**

- ask and answer questions
- Example: Grade Book. Some questions?
- *Requirements Analysis*
  - investigate requirements
- *Object-oriented Analysis*
  - investigate objects used in and by domain



A Short Example

Dicey: In which a program simulates a player tossing two dice.

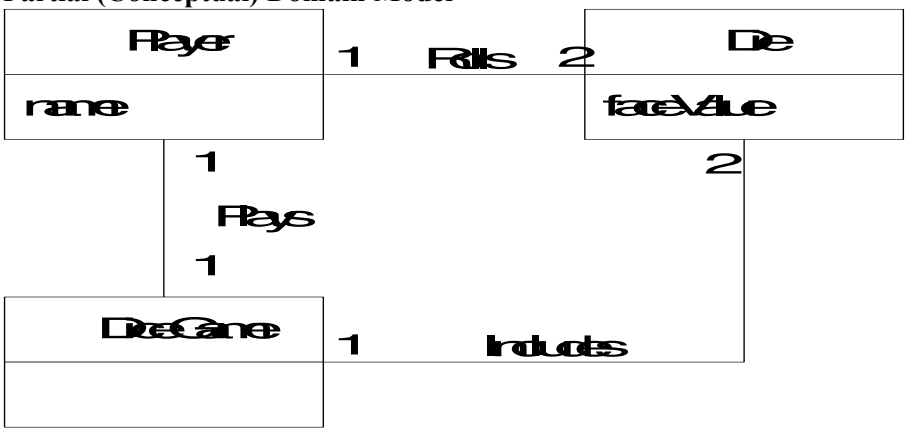
Define Use Cases

- **these are user scenarios, stories, goals**

Define a Domain Model

- **this is a description of the domain from the point of view of the objects involved**
- **identify the concepts, attributes and associations**
- **result is called the *domain model***

Partial (Conceptual) Domain Model



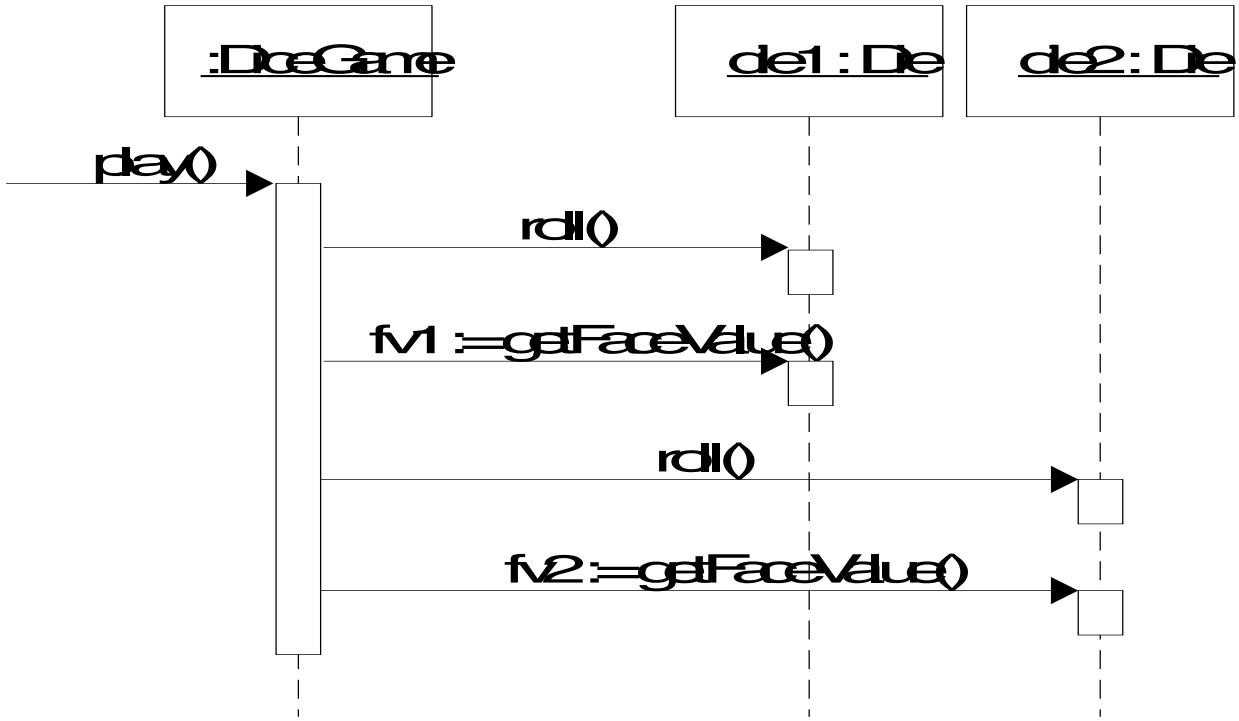
It is not description of the software, it is a visualization of the concepts or mental models of a real world domain.

It is also called conceptual object model.

Assigning Responsibilities

In a program, objects collaborate – pass each other messages, make data available to one another Common notation is called a *sequence diagram*. A sequence diagram shows the *flow of messages* between objects If we know where the message goes we know who is "responsible" for "responding to" the message. Answering the message involves knowing "how" to answer the message How close is "flow of message" to the concept of "function call"?

Sequence diagram – shows messages between objects

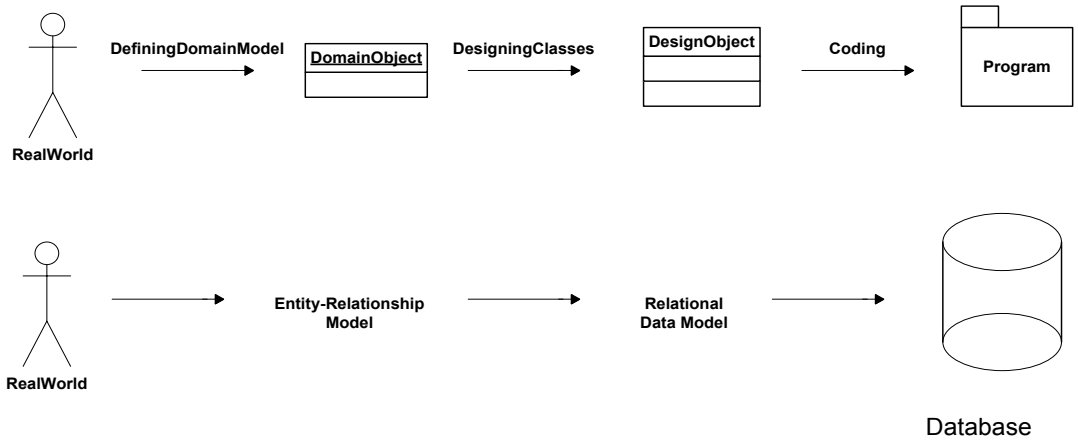


Defining Classes

- **The preceding is a *dynamic* view of collaborating classes.**
- **The following is a *static* view of collaborating classes; more often called a class diagram.**
- **Differs from the domain model, with similarities.**
- **Follows the sequence diagram in the order of things.**

Partial design class diagram

Bridging the Gap



What is UML?

The *Unified Modeling Language* is a visual language for *specifying*, *construction* and *documenting* the artifacts of systems. (OMG) . A lot more than we ever need to know including lots of software for drawing pictures .

How to Apply UML?

As a sketch: informal, hand-drawn documents, used for exploration

As a blueprint: detailed design documents, developed by tools that either *forward-* or *reverse engineer code.*

- ***forward:* tool takes a picture and produces executable code (mostly stubs).**
- ***reverse*: tool takes executable code and draws a picture.**

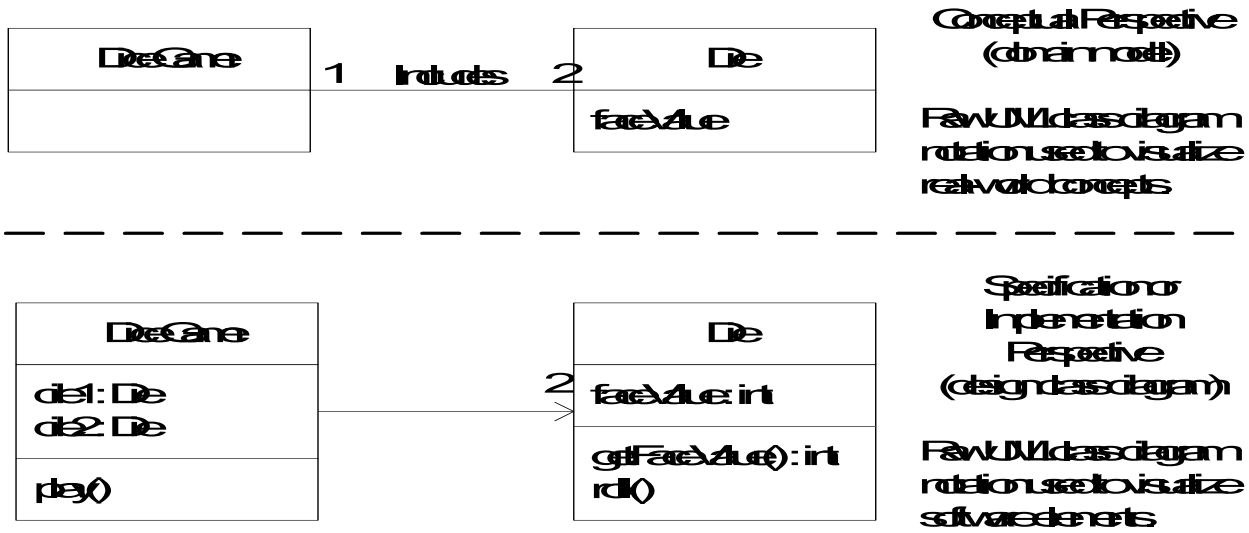
As a Programming Language: Tools produce complete executables.

As a Agile Programming: UML as a sketch.

Three Perspectives in Applying UML

- **UML does not dictate modeling perspective; you can use a class diagram for real-world concepts or Java classes.**
- **None the less, perspectives exist:**
  - **Conceptual class : Describe the real world concept or thing**
  - **software class : Describe software abstractions using specifications or interfaces – no commitment to a particular implementation**
  - **Implementation class : Describe an implementation in a particular language like Java.**

Different perspectives with UML



Different Perspectives of "Class".

- **Rectangular boxes are classes. But they can be physical things, abstract concepts, software things, events,**
- **A software design methods or methodologies superimposes structure/naming conventions on the various UML objects.**
- **For example, in the Unified Process (UP), Domain Model, a box represents a conceptual class. In the Design Model they are called Design Classes.**
- **Our book follows UP:**
  - **conceptual class, software class, implementation class.**

UML (Unified Markup Language) Standard Diagrams

Any complex system is best understood by making some kind of diagrams or pictures. These diagrams have a better impact on our understanding. We prepare UML diagrams to understand a system in better and simple way. A single diagram is not enough to cover all aspects of the system. So UML defines various kinds of diagrams to cover most of the aspects of a system. We can also create our own set of diagrams to meet our requirements. Diagrams are generally made in an incremental and iterative way.

Categories of Diagrams

There are two broad categories of diagrams and then are again divided into sub-categories:

- **Structural Diagrams**
- **Behavioral Diagrams**

Structural Diagrams

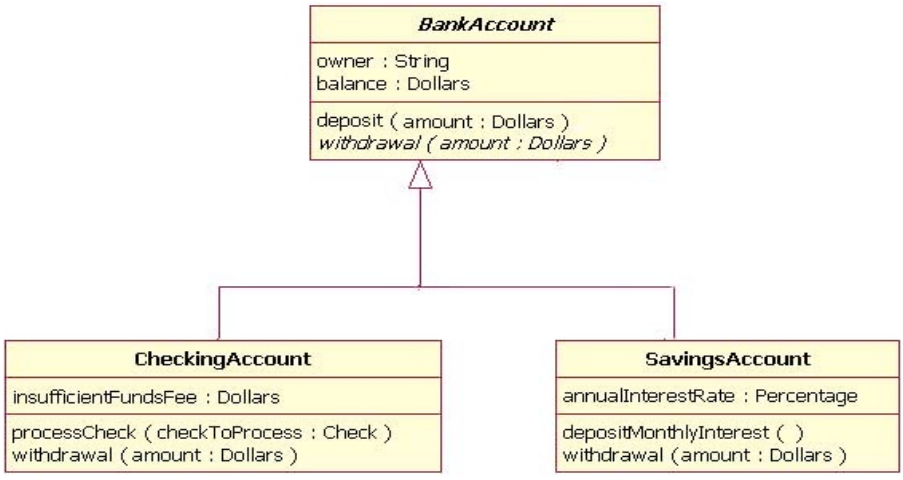
The *structural diagrams* represent the static aspect of the system. These static aspects represent those parts of a diagram which forms the main structure and therefore stable. These static parts are represents by classes, interfaces, objects, components and nodes.

The four structural diagrams are:

- **Class diagram**
- **Object diagram**
- **Component diagram**
- **Deployment diagram**

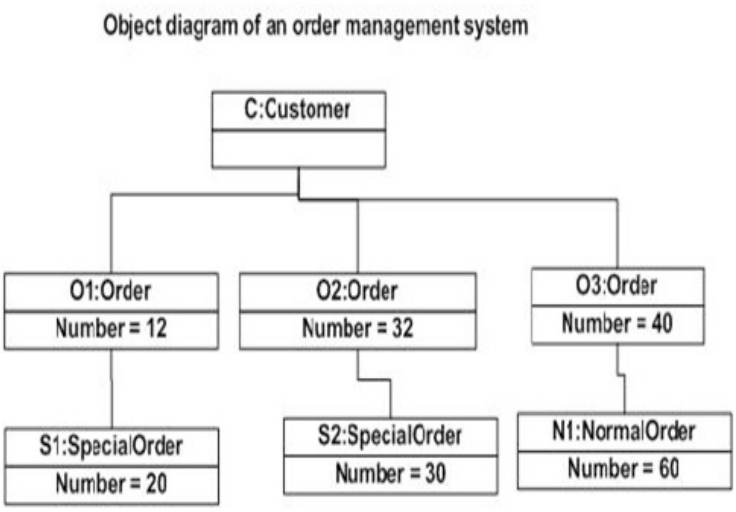
Class Diagram

Class diagrams are the most common diagrams used in UML. Class diagram consists of classes, interfaces, associations and collaboration. Class diagrams basically represent the object oriented view of a system which is static in nature. Active class is used in a class diagram to represent the concurrency of the system. Class diagram represents the object orientation of a system. So it is generally used for development purpose. This is the most widely used diagram at the time of system construction.



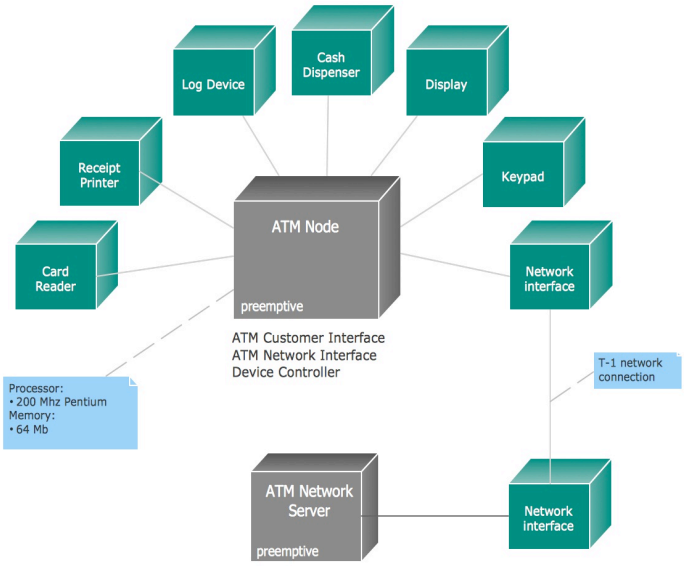
Object Diagram

Object diagrams can be described as an instance of class diagram. So these diagrams are more close to real life scenarios where we implement a system. Object diagrams are a set of objects and their relationships just like class diagrams and also represent the static view of the system. The usage of object diagrams is similar to class diagrams but they are used to build prototype of a system from practical perspective.



Component diagrams

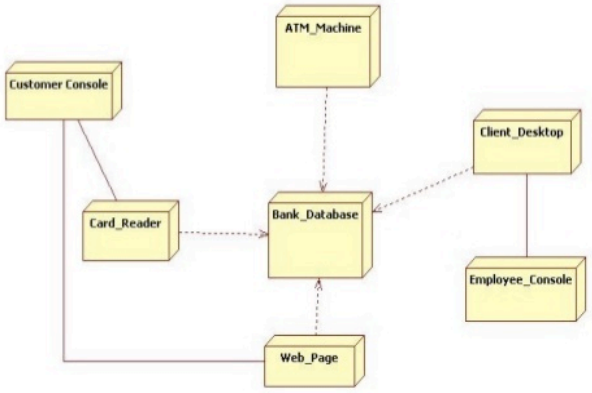
Component diagrams represent a set of components and their relationships. These components consist of classes, interfaces or collaborations. So Component diagrams represent the implementation view of a system. During design phase software artifacts (classes, interfaces etc) of a system are arranged in different groups depending upon their relationship. Now these groups are known as components. Finally, component diagrams are used to visualize the implementation.



Deployment Diagram

Deployment diagrams are a set of nodes and their relationships. These nodes are physical entities where the components are deployed. Deployment diagrams are used for visualizing deployment view of a system. This is generally used by the deployment team.

Deployment for ATM Machine



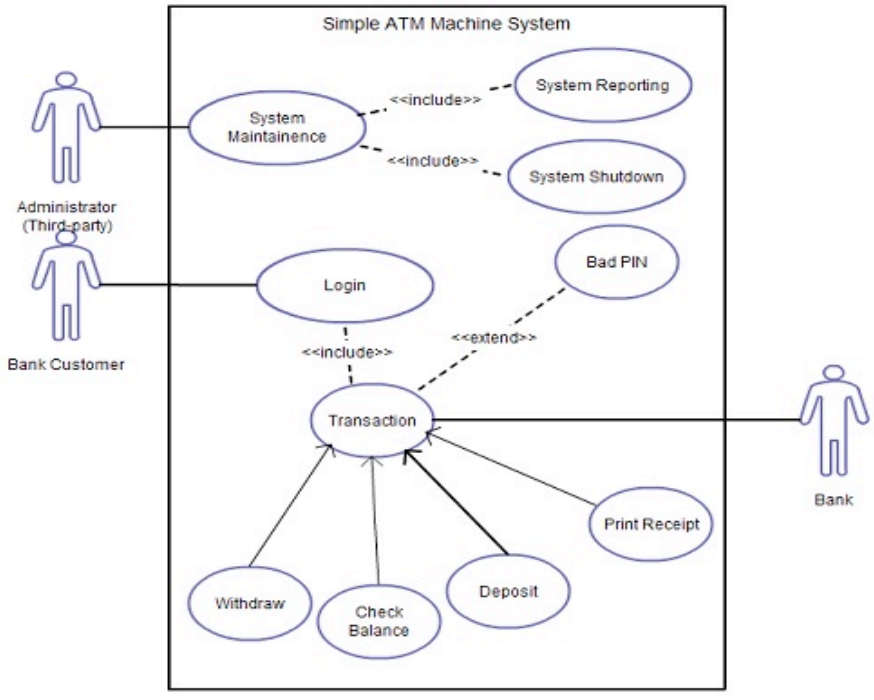
Behavioral Diagrams

Any system can have two aspects, static and dynamic. So a model is considered as complete when both the aspects are covered fully. Behavioral diagrams basically capture the dynamic aspect of a system. Dynamic aspect can be further described as the changing/moving parts of a system. UML has the following five types of behavioral diagrams:

- **Use case diagram**
- **Sequence diagram**
- **Collaboration diagram**
- **State chart diagram**
- **Activity diagram**

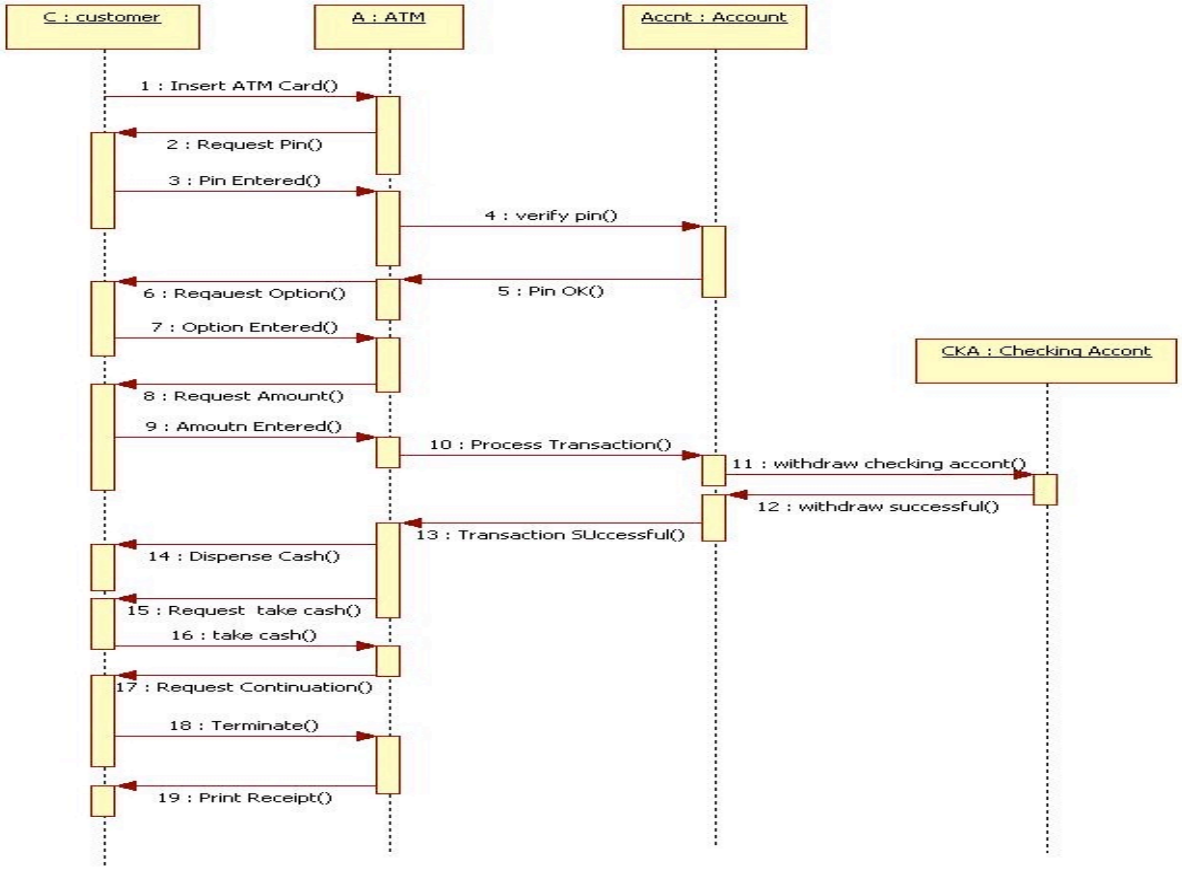
Use case Diagram

Use case diagrams are a set of use cases, actors and their relationships. They represent the use case view of a system. A use case represents a particular functionality of a system. So use case diagram is used to describe the relationships among the functionalities and their internal/external controllers. These controllers are known as actors.



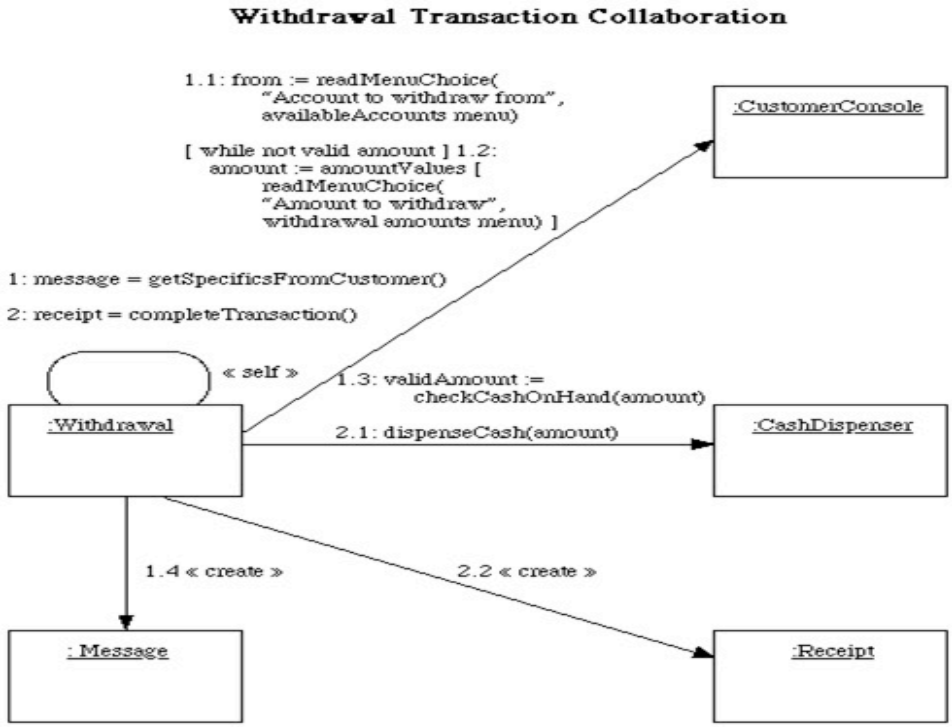
Sequence Diagram

A sequence diagram is an interaction diagram. From the name it is clear that the diagram deals with some sequences, which are the sequence of messages flowing from one object to another. Interaction among the components of a system is very important from implementation and execution perspective. Sequence diagram is used to visualize the sequence of calls in a system to perform a specific functionality.



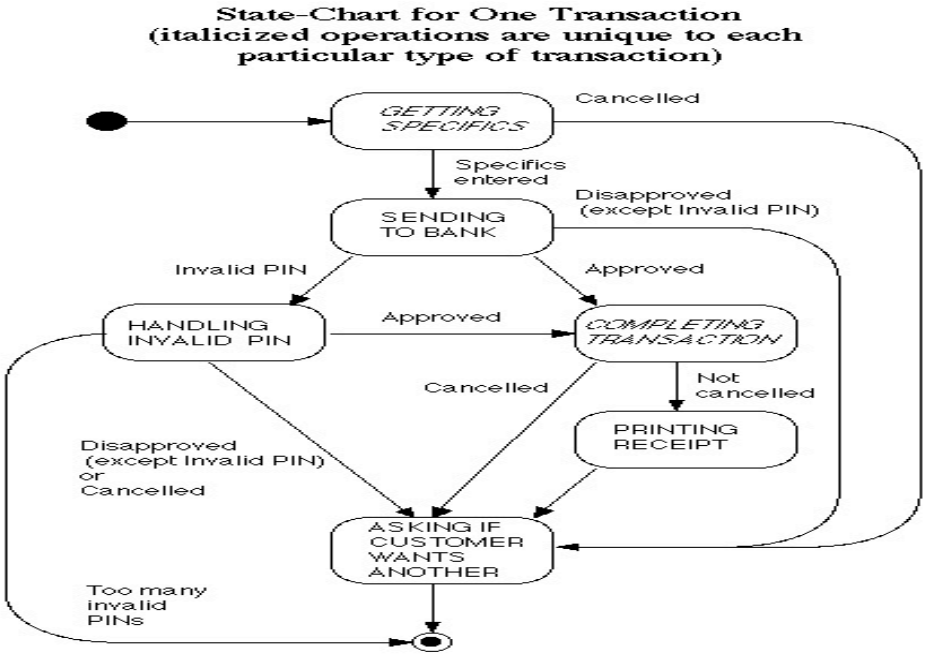
Collaboration Diagram

Collaboration diagram is another form of interaction diagram. It represents the structural organization of a system and the messages sent/received. Structural organization consists of objects and links. The purpose of collaboration diagram is similar to sequence diagram. But the specific purpose of collaboration diagram is to visualize the organization of objects and their interaction.



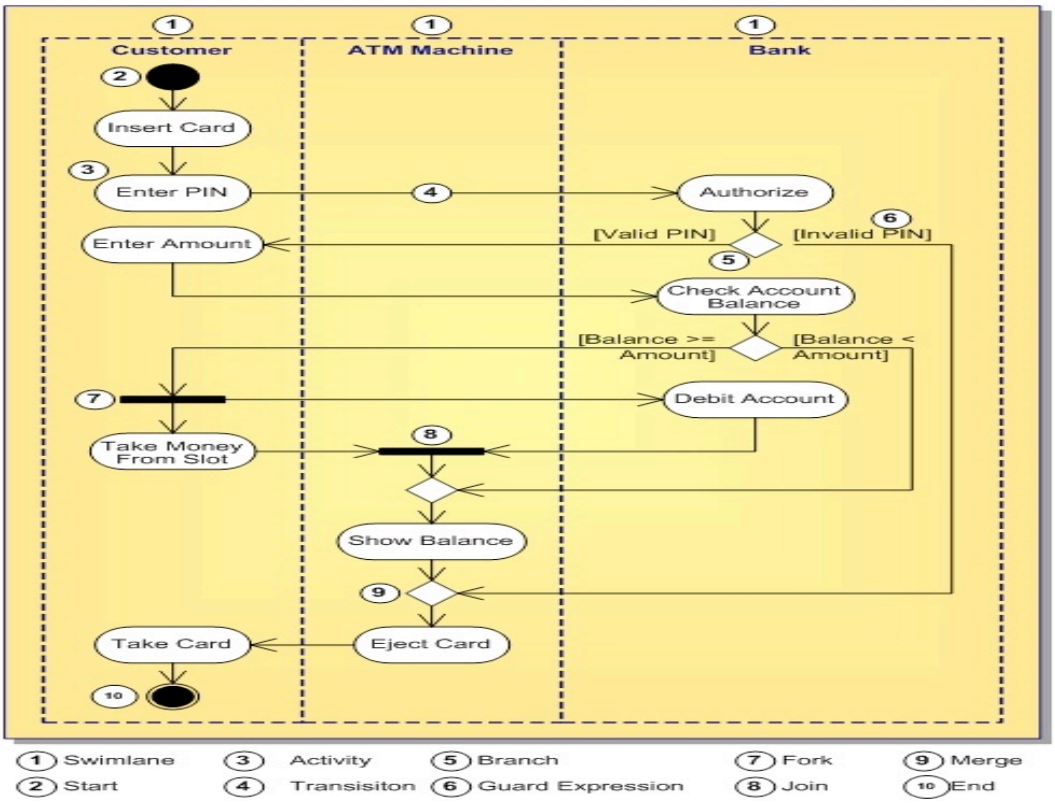
State chart Diagram

Any real time system is expected to be reacted by some kind of internal/external events. These events are responsible for state change of the system. State chart diagram is used to represent the event driven state change of a system. It basically describes the state change of a class, interface etc.



Activity Diagram

Activity diagram describes the flow of control in a system. So it consists of activities and links. The flow can be sequential, concurrent or branched. Activities are nothing but the functions of a system. Numbers of activity diagrams are prepared to capture the entire flow in a system. Activity diagrams are used to visualize the flow of controls in a system. This is prepared to have an idea of how the system will work when executed.



Different phases of unified process.

Iterative development and The Unified process

The Unified Process is a popular and iterative software development process for building object oriented system. In particular, the Rational Unified Process, as modified at Rational Software, is widely adopted by industry.

The Most Important Concept

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 *time boxed*.

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.

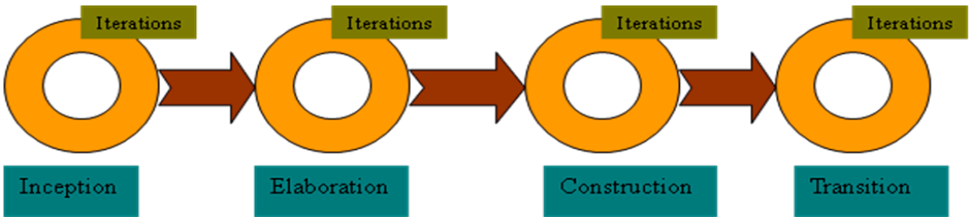
Phases in RUP

RUP is divided into four phases, named:

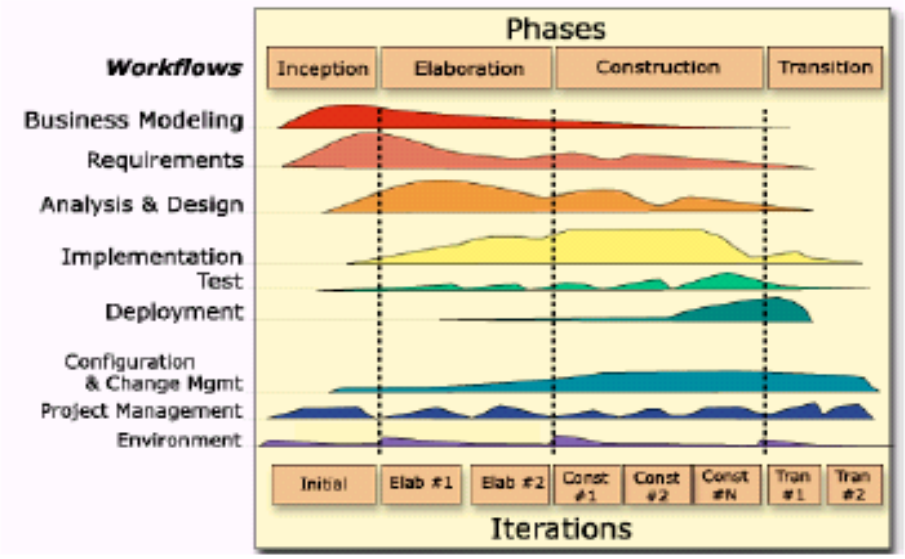
- **Inception**
- **Elaboration**
- **Construction**
- **Transition**

Iterations

- **Each 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

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

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 Prepare the project environment.

Inception - Exit criteria

An initial business case containing at least a clear formulation of the product vision - the core requirements - in terms of functionality, scope, performance, capacity, technology base. Success

criteria (example: revenue projection). An initial risk assessment. An estimate of the resources required to complete the elaboration phase.

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.

Elaboration - Entry criteria

The products and artifacts described in the exit criteria of the previous phase. The plan approved by the project management, and funding authority, and the resources required for the elaboration phase have been allocated. Define the architecture. Project plan is defined. The process, infrastructure and development environment are described.

Elaboration - Activities

Validate the architecture. Baseline the architecture. To provide a stable basis for the bulk of the design and implementation effort in the construction phase.

Elaboration - Exit criteria

A detailed software development plan, with an updated risk assessment, a management plan, a staffing plan, a phase plan showing the number and contents of the iteration , an iteration plan, and a test plan. The development environment and other tools . A baseline vision, in the form of a set of evaluation criteria for the final product . A domain analysis model, sufficient to be able to call the corresponding architecture 'complete'. An executable architecture baseline.

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.

Construction - Entry criteria

The product and artifacts of the previous iteration. The iteration plan must state the iteration specific goals. Risks being mitigated during this iteration. Defects being fixed during the iteration.

Construction – Activities

Develop and test components. Manage resources and control process. Assess the iteration

Construction - Exit Criteria

The same products and artifacts, updated. A release description document, which captures the results of an iteration. Test cases and results of the tests conducted on the products, An iteration plan, detailing the next iteration Objective measurable evaluation criteria for assessing the results of the next iteration(s).

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.

Transition - Entry criteria

The product and artifacts of the previous iteration, and in particular a software product sufficiently mature to be put into the hands of its users.

Transition – Activities

Test the product deliverable in a customer environment. Fine tune the product based upon customer feedback Deliver the final product to the end user .Finalize end-user support material

Transition - Exit criteria

An update of some of the previous documents, as necessary, the plan being replaced by a "post-mortem" analysis of the performance of the project relative to its original and revised success criteria; A brief inventory of the organization's new assets as a result this cycle.

Advantages of RUP

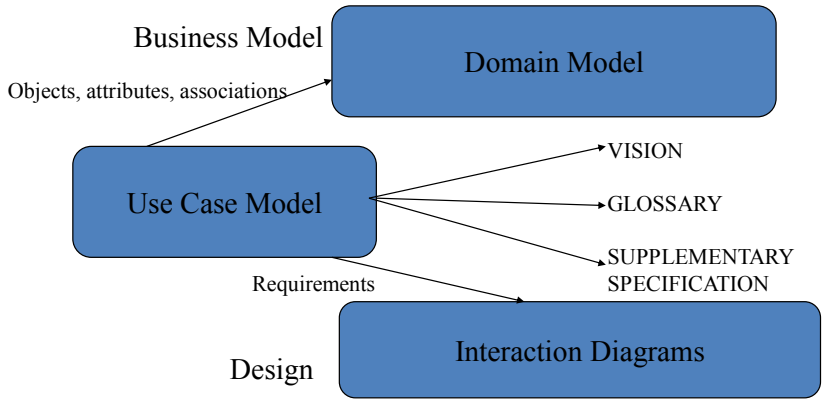
The RUP puts an emphasis on addressing very early high risks areas. It does not assume a fixed set of firm requirements at the inception of the project, but allows to refine the requirements as the project evolves. It does not put either a strong focus on documents The main focus remains the software product itself, and its quality.

Drawbacks of RUP

RUP is not considered particularly "agile". It fails to provide any clear implementation guidelines. RUP leaves the tailoring to the user entirely.

Use-Case Model

Use Case Relationships



2

Use case diagram

- **Use case concepts introduced by Ivar Jackson in object oriented s/w engineering.**
- **Use cases represent specific flow of events in the system**
- **Use cases defines the outside(actor) and inside(use case) of the system's behavior.**
- **Use case diagram is graph of actors, set of use cases enclosed by a system boundary.**

Use Cases are not Diagrams

- **Use Cases may have a diagram associated with them, and a use case diagram is an easy way for an analyst to discuss a process with a subject matter expert (SME).**
- **But use cases are primarily text. The text is important. The diagram is optional.**

Why Use Cases?

- **Simple and familiar story-telling makes it easier, especially for customers, to contribute and review goals.**
- **Use cases keep it simple**
- **They emphasize goals and the user perspective.**
- **New use case writers tend to take them too seriously.**

Actors or Use Case First?

- **An actor is a role that user plays with respect to the system.**
- **Typically, both actors and use cases are identified early and then examined to see if more use cases can be found from the actors, or more actors found by examining the use cases.**

How Use Cases look like?

- **Capture the specific ways of using the system as dialogues between an actor and the system.**
- **Use cases are used to**
  - **Capture system requirements**
  - **Communicate with end users and Subject Matter Experts**

Test the system

Three common use case formats

- **Brief – one paragraph summary, usually of the main success scenario.**
- **Casual – informal paragraph format. Multiple paragraphs that cover various scenario.**
- **Fully dressed – all steps and variations are written in detail , and there are supporting sections , like**

preconditions and success guarantees.

Use cases Template

Use case section	comment
Name	Start with verb
Scope	The system under design
Level	User goal or sub function
Primary Actor	Calls on the system to deliver its service
Stakeholders and Interests	Who cares about this use case, and what do they want?
preconditions	What must be true on start and worth telling the reader?
Success Guarantee	What must be true on successful completion, and worth telling the reader?
Main Success Scenario	A typical, unconditional happy path scenario of success
Extensions	Alternate scenarios of success
Special requirement	Related non functional requirement

Case Study I: NextGen POS System

- **NextGen point-of-sale (POS) system – Computerized application used to record sales and handle payments typically used in a retail store.**
- **Components – Hardware: computer and bar code scanner & Software**
- **Interfaces to various service applications, such as a third party tax calculator and inventory control**
- **Must be relatively fault-tolerant – Even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments**
- **Increasingly must support multiple and varied client-side terminals and interfaces – Thin-client Web browser terminal – Regular personal computer with graphical user interface – Touch screen input – Wireless PDAs, etc.**

USE CASE : Process Sale (FULLY DRESSED VERSION)

- **Primary Actor: Cashier**
- **Stakeholders and Interests:**
  - **Cashier: Wants accurate and fast entry, no payment errors…**
  - **Salesperson: Wants sales commissions updated. …**
- **Preconditions: Cashier is identified and authenticated.**
- **Success Guarantee (Post conditions):**
  - **Sale is saved. Tax correctly calculated.…**

Main success scenario (or basic flow):…

<u>Use case</u>

- **Main success scenario (or basic flow):**

1. **The Customer arrives at a POS checkout with items to purchase.**
2. **cashier starts new sale**
3. **Cashier enters item identifier**
4. **System records sale line item and presents item description, price and running total.**

Cashier repeats step 3-4 until indicates done.

5. **System presents total with taxes calculated.**
6. **Cashier tells customer the total, and asks for payment.**
7. **Customer pays and system handles payment.**
8. **System logs completed sale and send sale and payment info to the external accounting system and inventory system.(to update inventory)**
9. **System presents receipt.**
10. **Customer leaves with receipt and goods.**

- **Extensions (or alternative flows):**
  - **If invalid identifier entered. Indicate error.**
  - **If customer didn't have enough cash, cancel sales transaction.**
  - **Customer paying by debit card … invalid pin**
  - **Printer out of paper.**
- **Special requirements: Touch screen UI, …**
- **Technology and Data Variations List:**

- **Identifier entered by bar code scanner,…**

- **Open issues: What are the tax law variations?...**
  Things that are in Use Cases
- **Create a written document for each Use Case**
  - **Clearly define intent of the Use Case**
  - **Define Main Success Scenario (Happy Path)**
  - **Define any alternate action paths**
  - **Use format of Stimulus: Response**
  - **Each specification must be testable**
  - **Write from actor's perspective, in actor's vocabulary**

Elements in the Preface

- **Only put items that are important to understand before reading the Main Success Scenario. These might include:**
- **Name (*Always needed for identification*)**
- **Primary Actor**
- **Stakeholders and Interests List**
- **Preconditions**
  Success guarantee (Post Conditions)
  Naming Use Cases
- **Must be a complete process from the viewpoint of the end user.**
- **Usually in verb-object form, like Buy Pizza**
- **Use enough detail to make it specific**
- **Use active voice, not passive**
- **From the viewpoint of the actor, not the system**

Golden Rule of Use-Case Names

- **Each use case should have a name that indicates what value (or goal) is achieved by the actor's interaction with the system**
- **Here are some good questions to help you adhere to this rule:**
  - **Why would the actor initiate this interaction with the system?**
    What goal does the actor have in mind when undertaking these actions?
    What value is achieved and for which actor?

Use Case Name Examples

- **Excellent - Purchase Concert Ticket**
- **Very Good - Purchase Concert Tickets**
- **Good - Purchase Ticket (insufficient detail)**
- **Fair - Ticket Purchase (passive)**
- **Poor - Ticket Order (system view, not user)**
- **Unacceptable - Pay for Ticket (procedure, not process)**

CRUD

- **Examples of bad use case names with the acronym CRUD. (All are procedural and reveal nothing about the actor's intentions.)**
- **C - actor Creates data**
- **R - actor Retrieves data**
- **U - actor Updates data**
- **D - actor Deletes data**

Identify Actors

- **We cannot understand a system until we know who will use it**
  - **Direct users**
  - **Users responsible to operate and maintain it**
  - **External systems used by the system**
  - **External systems that interact with the system**

Types of Actors

- **Primary Actor**
  - **Has goals to be fulfilled by system**
- **Supporting Actor**
  - **Provides service to the system**
- **Offstage Actor**
  - **Interested in the behavior, but no contribution**
- **In diagrams, Primary actors go on the left and others on the right.**

Define Actors

- **Actors should not be analyzed or described in detail unless the application domain demands it.**
- **Template for definition:**
  - **Name**
  - **Definition**
- **Example for an ATM application:**
  Customer: Owner of an account who manages account by depositing and withdrawing funds
  Working with Use Cases
- **Determine the actors that will interact with the system**
- **Examine the actors and document their needs**
- **For each separate need, create a use case**
- **During Analysis, extend use cases with interaction diagrams**

Preconditions

- **Anything that must always be true before beginning a scenario is a precondition.**
- **Preconditions are assumed to be true, not tested within the Use Case itself.**
- **Ignore obvious preconditions such as the power being turned on. Only document items necessary to understand the Use Case.**

Success Guarantees

- **Success Guarantees (or Post conditions) state what must be true if the Use Case is completed successfully.**
- **This may include the main success scenario and some alternative paths. For example, if the happy path is a cash sale, a credit sale might also be regarded a success.**
- **Stakeholders should agree on the guarantee.**

Scenarios

- **The Main Success Scenario, or "happy path" is the expected primary use of the system, without problems or exceptions.**

Alternative Scenarios or Extensions are used to document other common paths through the system and error handling or exceptions

Documenting the Happy Path

- **The Success Scenario (or basic course) gives the best understanding of the use case**
- **Each step contains the activities and inputs of the actor and the system response**
- **If there are three or more items, create a list**
- **Label steps for configuration management and requirements traceability**
- **Use present tense and active voice**
- **Remember that User Interface designers will use this specification**

Note: Do not use the term "happy path" in formal documents.

Documenting Extensions

- **Use same format as Happy Path**
- **Document actions that vary from ideal path**
- **Include error conditions**
- **Number each alternate, and start with the condition:**

3A. Condition: If [actor] performs [action] the system …

- **If subsequent steps are the same as the happy path, identify and label as (same)**
- **Steps not included in alternate course are assumed not to be performed.**

Two Parts for Extensions

- **Condition**
  - **Describe the reason for the alternative flow as a condition that the user can detect**
- **Handling**
  - **Describe the flow of processing in the same manner as the happy path, using a numbering system consistent with the original section.**

Special Requirements

If a non-functional requirement , quality attribute, or constraint affects a use case directly, describe it as a special requirement

Technology and Data Variations List

- **Often there are technical differences in how things are done even though what is done is the same. These things can be described in the Technology and Data Variations List.**

- **For example, if a card reader cannot read the magnetic stripe on a credit card, the cashier might be able to enter it on the keyboard.**

Types of Use Cases

- **The most common Use Cases are High Level Use Cases and Expanded Essential Use Cases in analysis, and Expanded Real Use Cases in design. The next slide gives definitions.**
- **In addition, Use Case diagrams may be used in discussions with stakeholders while capturing their requirements.**

Elaborating Use Cases

- **High Level Use Case (Brief)**
  - **Name, Actors, Purpose, Overview**
- **Expanded Use Case (Fully Dressed)**
  - **Add System Events and System Responses**
- **Essential Use Case (Black Box)**
  - **Leave out technological implications**
- **Real Use Case (White Box)**
  - **Leave in technology**

Technology

- **The distinction between an essential (black box) use case that leaves out technology and a real (white box) use case that includes technology is fundamental.**
- **For example, in an Automated Teller Machine, an *essential* use case can mention identification or validation, but only a *real* use case can mention a key pad or card reader.**

Post conditions

- **Post conditions (or success guarantees) state what always must be true for a use case to succeed. Avoid the obvious, but clearly document any that are not obvious. This is one of the most important parts of a use case.**

Conditions and Branching

Stick to the "Happy Path," "Sunny Day Scenario," Typical Flow, or Basic Flow (*all names for the same basic idea*) in the main section and defer all conditional sections and branching to the extensions or alternate flows

Extension Use Cases

- **Users appreciate simplicity, so most use cases leave out alternate courses**
- **You can do this by extending the use case while leaving the original use case alone**

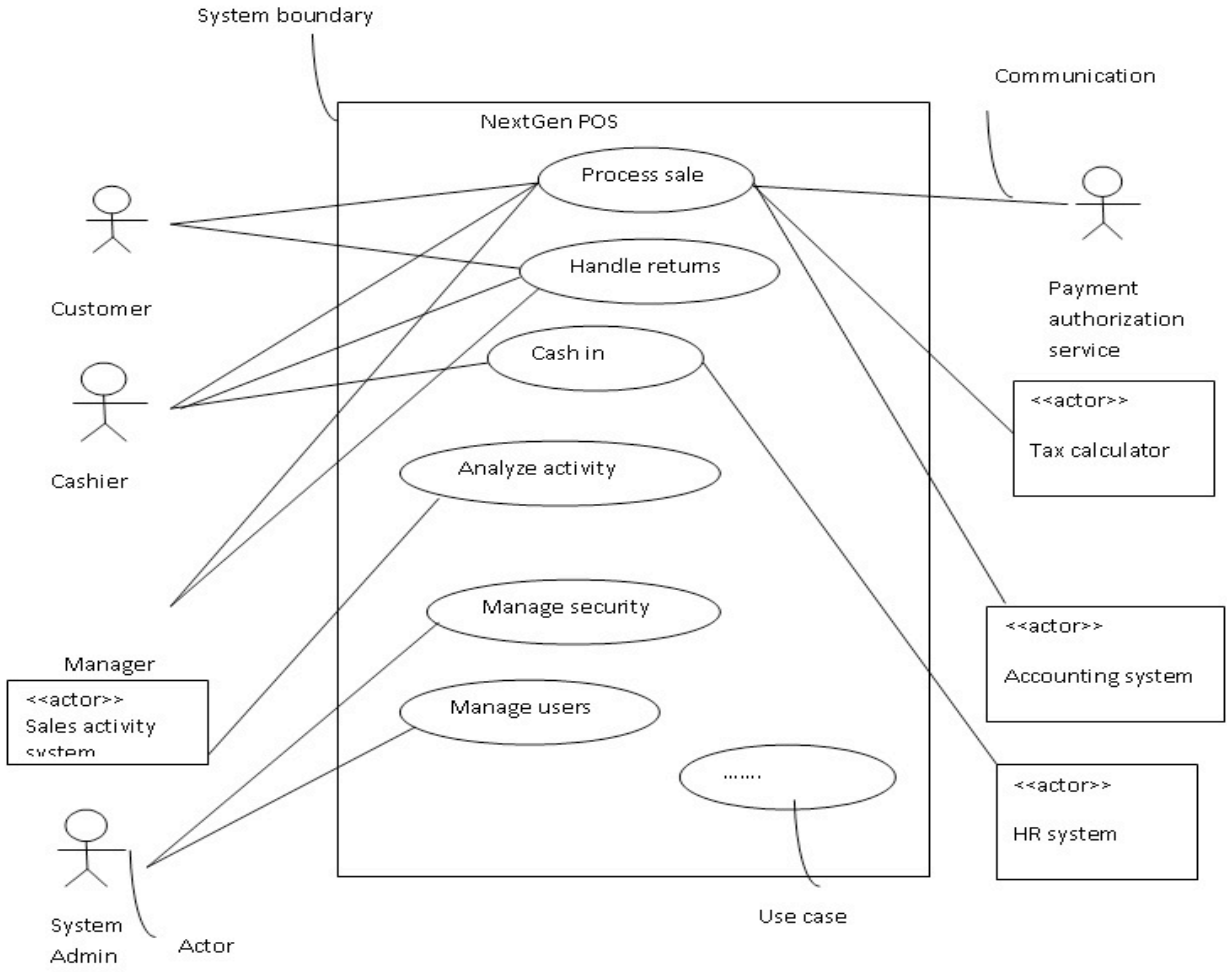
Feature Lists

- **Older methods of detailing requirements tended to have many pages of detailed feature lists. Usually the details could not be seen in context.**
- **Current philosophy is to use a higher level of detail with use cases instead of a list.**
- **High level System Feature Lists are acceptable when they can give a succinct summary of the system.**

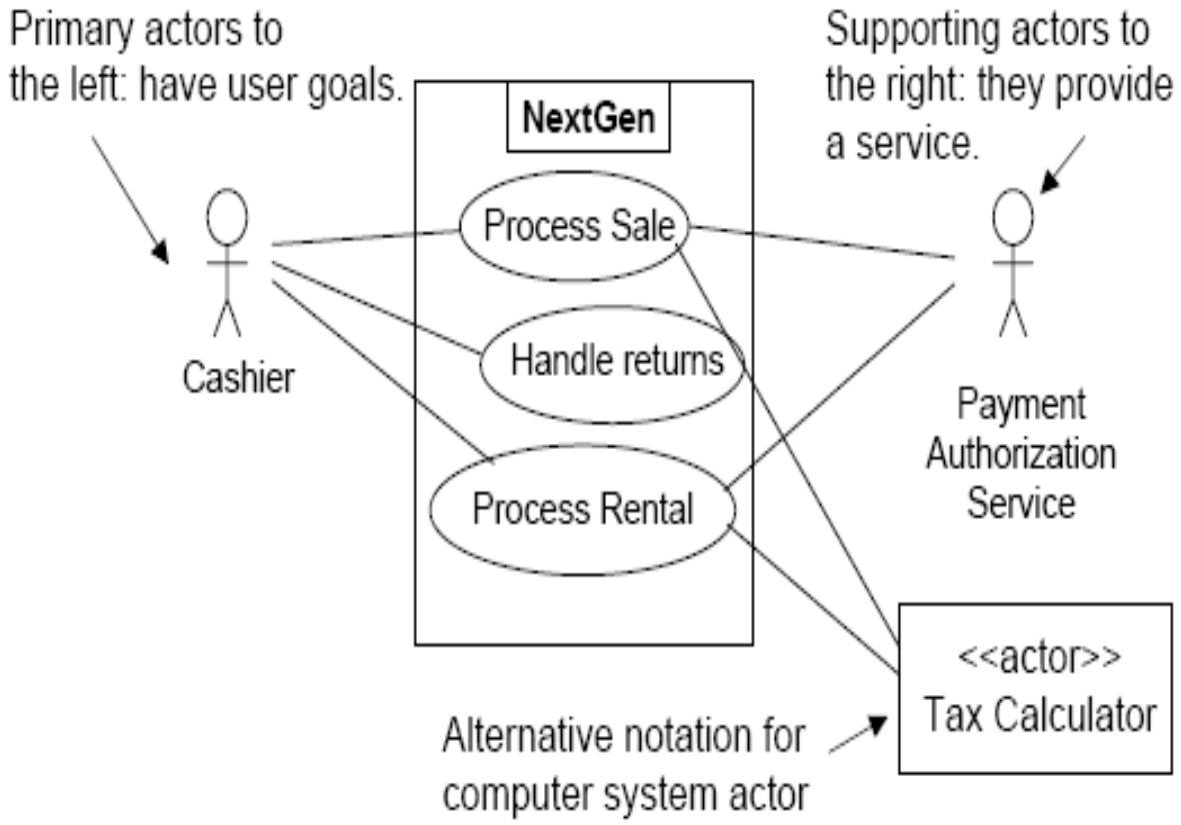
Use Cases not an OO idea

- **Use Cases are not an Object-Oriented methodology. They are common in structured development as well.**
- **However, the Unified Process encourages use-case driven development.**

Partial use case diagram – POS system

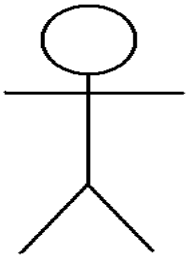


Overview



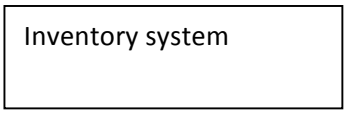
- **A use case diagram identifies transactions between actors and a system as individual use cases**

Actor



- An actor is an idealized user of a system
- Actors can be users, processes, and other systems
- Many users can be one actor, in a common role
- One user can be different actors, based on different roles
- An actor is labeled with the name of the role

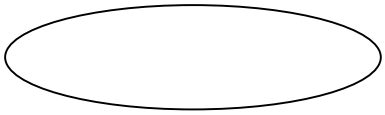
Non-human Actor



- Actors can be users, processes, and other systems.
- Show non human actors in a different manner, usually as a rectangle

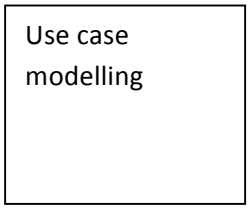
Non human actors are usually not primary users, and thus are usually shown on the right, not the left

Use Case



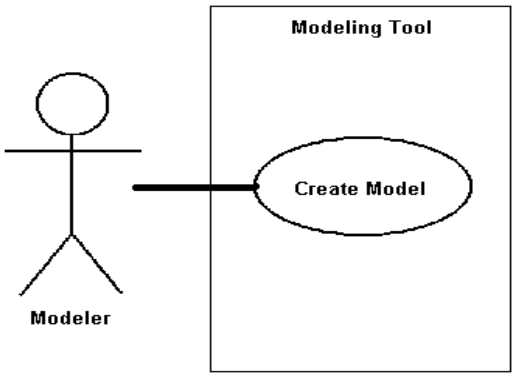
- A use case is a coherent unit of externally visible functionality provided by a system and expressed by a sequence of messages
- Additional behavior can be shown with parent-child, extend and include use cases
- It is labeled with a name that the user can understand

System



- A system is shown as a rectangle, labeled with the system name
- Actors are outside the system
- Use cases are inside the system
- The rectangle shows the scope or boundary of the system

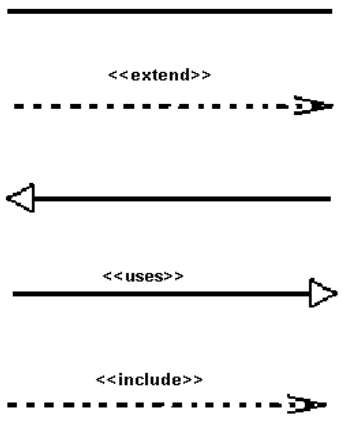
Association Relationship



- An association is the communication path between an actor and the use case that it participates in
- It is shown as a solid line
- It does not have an arrow, and is normally read from left to right
- Here, the association is between a Modeler and the Create Model use case

Relationships in Use Cases

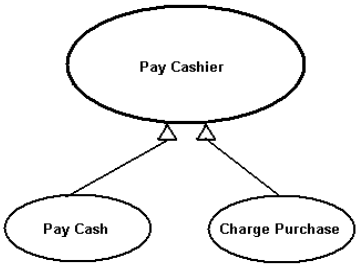
- There are several Use Case relationships:
- Association
- Extend
- Generalization
- Uses
- Include



Extend Relationship

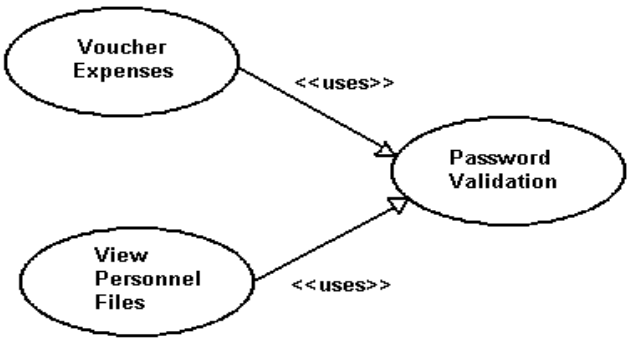
- Extend puts additional behavior in a use case that does not know about it.
- It is shown as a dotted line with an arrow point and labeled <<extend>>
- In this case, a customer can request a catalog when placing an order

Use Case Generalization



- Generalization is a relat e and a more specific use case that inherits and extends features to it
- It is shown as a solid line with a closed arrow point
- Here, the payment process is modified for cash and charge cards

Uses Relationship



- When a use case uses another process, the relationship can be shown with the uses relationship
- This is shown as a solid line with a closed arrow point and the <<uses>> keyword
- Here different system processes can use the logon use case

Include Relationship

- Include relationships insert additional behavior into a base use case
- They are shown as a dotted line with an open arrow and the key word <<include>>
- Shown is a process that I observed in an earlier career

Use Case Example- ATM