

1.1 INTRODUCTION TO OOAD:

- ❑ **Analysis** emphasizes an *investigation* of the problem and requirements, rather than a solution. **Design** emphasizes a *conceptual solution* that fulfils the requirements, rather than its implementation.
- ❑ Analysis and design have been summarized in the phrase ***do the right thing (analysis), and do the thing right (design)***.
- ❑ During **object-oriented analysis**, there is an emphasis on finding and describing the objects—or concepts—in the problem domain. For example, in the case of the library information system, some of the concepts include *Book*, *Library*, and *Patron*.
- ❑ During **object-oriented design**, there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. For example, in the library system, a *Book* software object may have a *title* attribute and a *getChapter* method.

The various steps in object oriented analysis are as given below:



1. **Define Use Cases**

Requirements analysis may include a description of related domain processes; these can be written as **use cases**.

2. **Define a Domain Model**

Object-oriented analysis is concerned with creating a description of the domain from the perspective of classification by objects. A decomposition of the domain involves an identification of the concepts, attributes, and associations that are considered noteworthy. The result can be expressed in a **domain model**.

3. **Define Interaction Diagrams**

Object-oriented design is concerned with defining software objects and their collaborations. A common notation to illustrate these collaborations is the **interaction diagram**. It shows the

flow of messages between software objects, and thus the invocation of methods.

4. Define Design Class Diagrams

In addition to a *dynamic* view of collaborating objects shown in interaction diagrams, it is useful to create a *static* view of the class definitions with a **design class diagram**. This illustrates the attributes and methods of the classes.

UML:

The Unified Modeling Language (UML) 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.

Three ways to apply UML:

1. **UML as sketch:** informal and incomplete diagrams created to explore the different parts of problem of solution space.
 - a. Very quick.
 - b. Preferred by agile modellers
2. **UML as blueprint:** Relatively detailed design diagrams used for reverse engineering or code generation.
3. **UML as executable programming language:** complete executable specification of a software system in UML.
 - a. Compile models directly into executable code
 - b. Eliminates costly coding activity.

Three Perspectives to Apply UML

The UML describes raw diagram types, such as class diagrams and sequence diagrams. It does not superimpose a modeling perspective on these. For example, the same UML class diagram notation can be used to draw pictures of concepts in the real world or software classes in Java.

This insight was emphasized in the Syntropy object-oriented method [CD94]. That is, the same notation may be used for three perspectives and types of models (Figure 1.6):

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

1.2 UNIFIED PROCESS



The **Unified Process** has emerged as a popular software development process for building object-oriented systems.

The various characteristics of a unified process:

- ✚ Iterative and incremental
- ✚ Time boxed
- ✚ Architecture centric
- ✚ Risk focussed

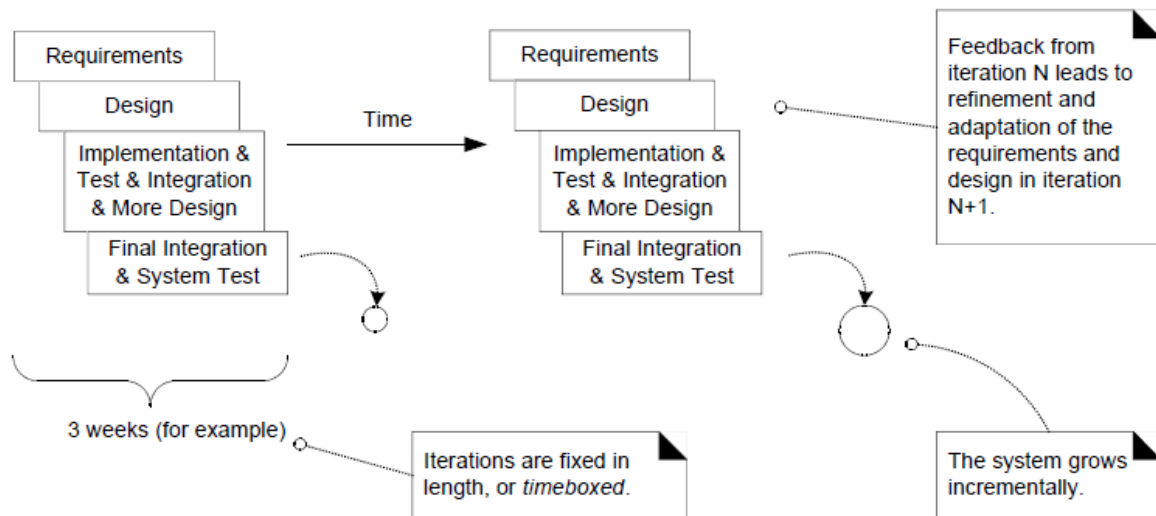
UNIFIED PROCESS -Iterative and incremental processes:

- ▶ In this approach, development is organized into a series of short, fixed-length (for example, four week) mini-projects called **iterations**.
- ▶ The outcome of each is a tested, integrated, and executable system.
- ▶ Iteration includes its own requirements analysis, design, implementation, and testing activities.
- ▶ The iterative lifecycle is based on the **successive enlargement and refinement of a system through multiple iterations, with cyclic feedback and adaptation** as core drivers to converge upon a suitable system.
- ▶ The system grows incrementally over time, iteration by iteration, and thus this approach is also known as **iterative and incremental development**.
- ▶ Before all requirements are finalized, or the entire design is speculatively defined, leads to rapid feedback—feedback from the users, developers, and tests

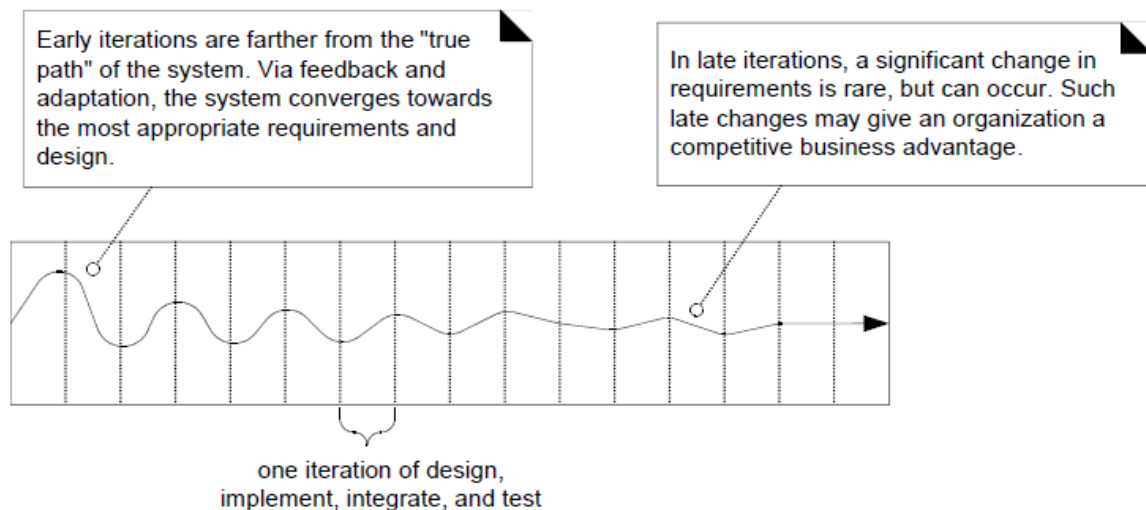
Iterative SW Development Process



Each iteration is a mini-waterfall process



- Consequently, work proceeds through a series of structured **build-feedback-adapt cycles**.



Benefits of Iterative Development

Benefits of iterative development include:

- Early rather than late mitigation of high risks (technical, requirements, objectives, usability, and so forth)
- Early visible progress

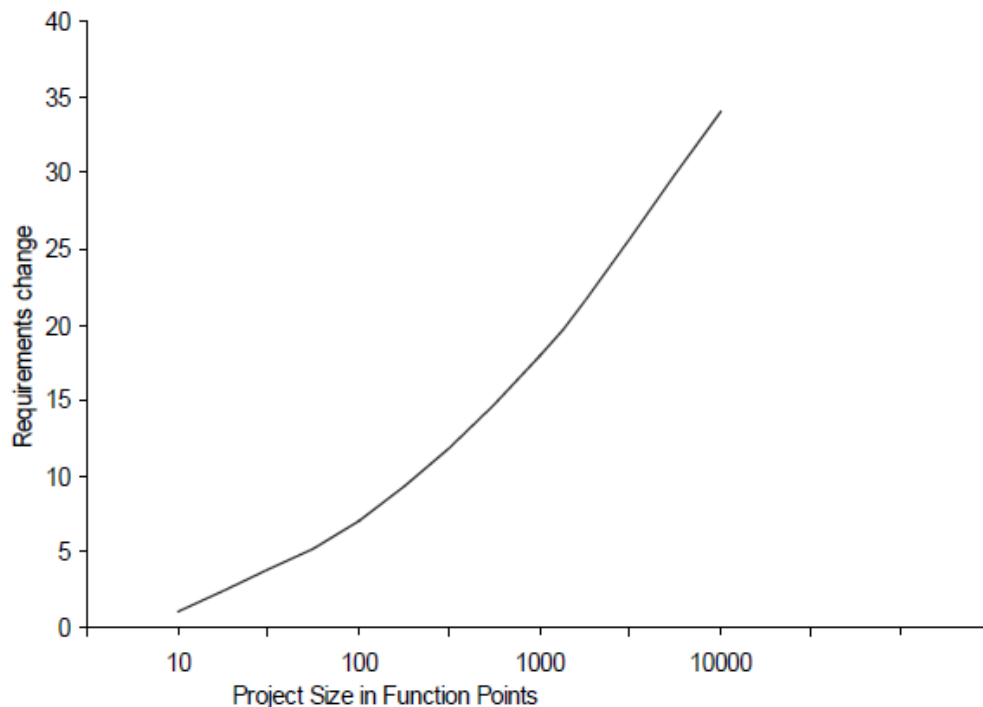
- Early feedback, user engagement, and adaptation, leading to a refined system that more closely meets the real needs of the stakeholders
- managed complexity; the team is not overwhelmed by "analysis paralysis"
- The learning within iteration can be methodically used to improve the development process itself, iteration by iteration.

UNIFIED PROCESS –Time boxed:

- ❑ The UP (and experienced iterative developers) recommends an iteration length between two and six weeks.
- ❑ Small steps, rapid feedback, and adaptation are central ideas in iterative development whereas long iterations subvert the core motivation for iterative development and increase project risk.
- ❑ A key idea is that iterations are **time boxed**, or fixed in length.
- ❑ For example, if the next iteration is chosen to be four weeks long, then the partial system should be integrated, tested, and stabilized by the scheduled date—date slippage is discouraged.
- ❑ If it seems that it will be difficult to meet the deadline, the recommended response is to remove tasks or requirements from the iteration, and include them in a future iteration, rather than slip the completion date.

Thus, any analysis, modeling, development, or management practice based on the assumption that things are long-term stable (i.e., the waterfall) is fundamentally flawed. *Change* is the constant on software projects. Iterative and evolutionary methods assume and embrace change and adaptation of *partial and evolving* specifications, models, and plans based on feedback.

Requirements change increases with project size
Manufacture of a SW 'product' is not predictable
Waterfall process too rigid



UNIFIED PROCESS - Architecture Centric

The Unified Process insists that architecture sit at the heart of the project team's efforts to shape the system. Since no single model is sufficient to cover all aspects of a system, the Unified Process supports multiple architectural models and views.

UNIFIED PROCESS - Risk Focused

The Unified Process requires the project team to focus on addressing the most critical risks early in the project life cycle. The deliverables of each iteration, especially in the Elaboration phase, must be selected in order to ensure that the greatest risks are addressed first.

PHASES IN A UNIFIED PROCESS:



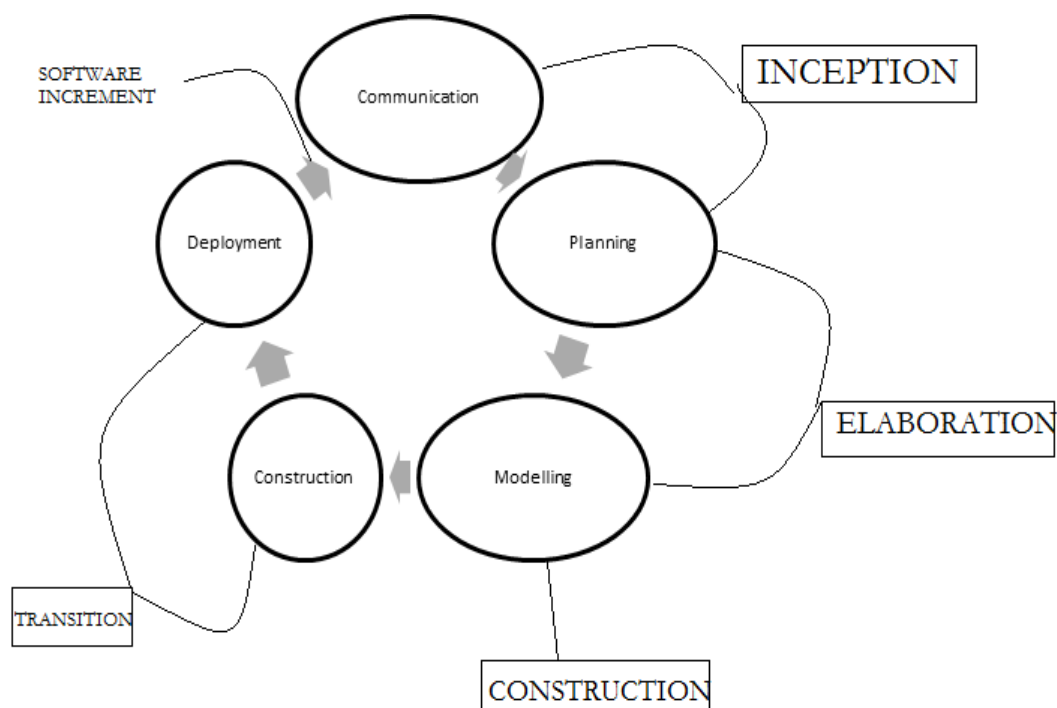
The various phases of an iterative life cycle model of software development are:

- Communication
- Planning
- Modelling
- Construction
- Deployment

The various phases of a unified process are:

- Inception
- Elaboration
- Construction
- Transition

Both these factors are mapped together as given in the following diagram:



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



- 1) **Inception**— this phase involves approximate vision, business case, scope, vague estimates.
 - i) Inception is not a requirements phase; rather, it is a kind of feasibility phase,
 - ii) Here, investigation is done to support a decision to continue or stop.
 - iii) It involves communication and planning.
- 2) **Elaboration**— it involves refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates.
 - i) Elaboration is not the requirements or design phase; rather, it is a phase where the core architecture is iteratively implemented, and high risk issues are mitigated.
 - ii) The elaboration phase is where the project starts to take shape. In this phase the problem domain analysis is made and the architecture of the project gets its basic form.
 - iii) It involves planning and modelling.
- 3) **Construction**—iterative implementation of the remaining lower risk and easier elements, and preparation for deployment.
 - i) In this phase, the main focus is on the development of components and other features of the system. This is the phase when the bulk of the coding takes place.
- 4) **Transition**— it mainly focuses on beta tests, deployment.
 - (i) The primary objective is to 'transit' the system from development into production, making it available to and understood by the end user.
 - (ii) The activities of this phase include training the end users and maintainers and beta testing the system to validate it against the end users' expectations.

- (iii) The product is also checked against the quality level set in the Inception phase.

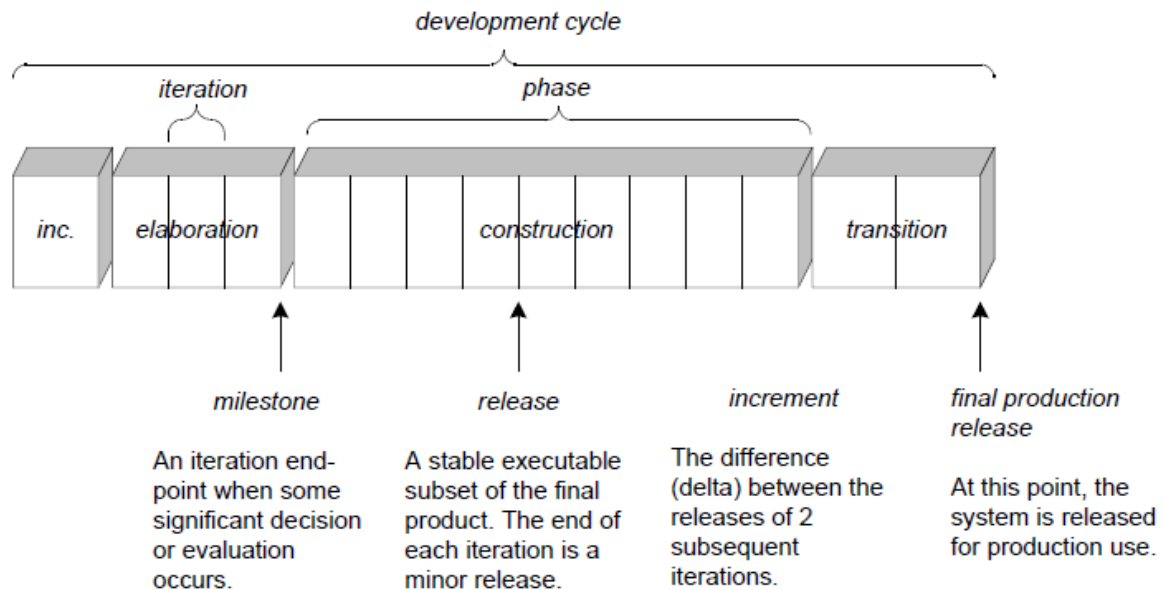


FIG: VARIOUS PHASES IN UNIFIED PROCESS

The UP Disciplines:

- 1) The UP describes work activities, such as writing a use case, within **disciplines** (originally called **workflows**)
- 2) A discipline is a set of activities (and related artifacts) in one subject area, such as the activities within requirements analysis.
- 3) There are several disciplines in the UP. But some important artifacts are:
 - a) **Business Modeling**—when developing a single application, this includes domain object modeling. When engaged in large-scale business analysis or business process reengineering, this includes dynamic modeling of the business processes across the entire enterprise.
 - b) **Requirements**—Requirements analysis for an application, such as writing uses cases and identifying non-functional requirements.
 - c) **Design**—All aspects of design, including the overall architecture, objects, databases, networking, and the like.

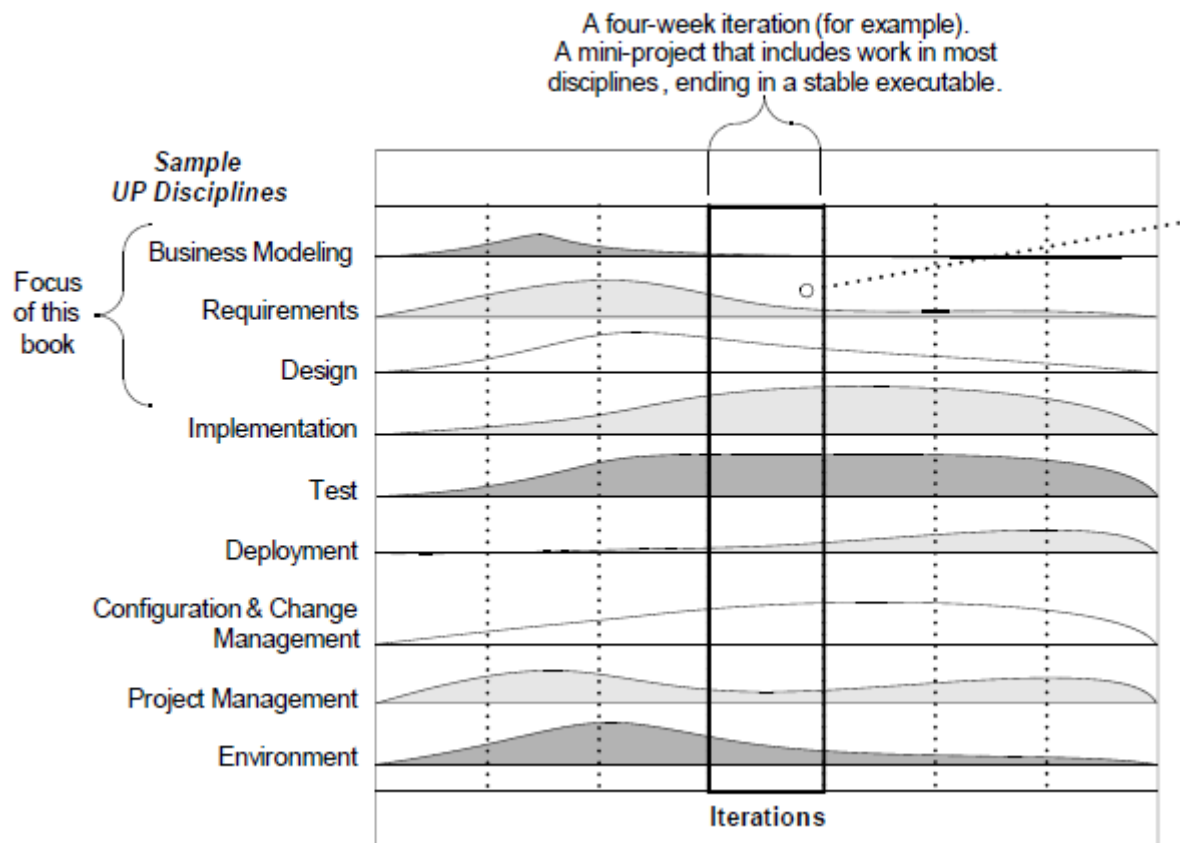
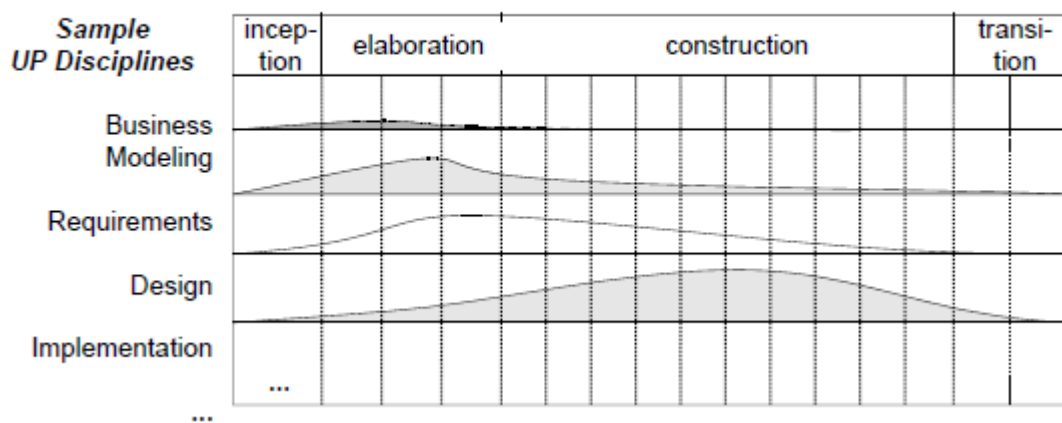


FIG: VARIOUS UNIFIED PROCESS DISCIPLINES

The various phases and disciplines of a Unified Process are mapped as given below:



AGILE UNIFIED PROCESS:

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

The Agile Principles

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

- ▶ A **heavy process** is a pejorative term meant to suggest one with the following qualities :
 - many artifacts created in a bureaucratic atmosphere
 - rigidity and control
 - elaborate, long-term, detailed planning
 - predictive rather than adaptive
 - ▶ A **predictive process** is one that attempts to plan and predict the activities and resource (people) allocations in detail over a relatively long time span, such as the majority of a project. Predictive processes usually have a "waterfall" or sequential lifecycle—first, defining all the requirements; second, defining a detailed design; and third, implementing.
 - ▶ An **adaptive process** is one that accepts change as an inevitable driver and encourages flexible adaptation; they usually have an iterative lifecycle.
 - ▶ An **agile process** implies a light and adaptive process, nimble in response to changing needs.
-

FURPS+ MODEL OF UNIFIED PROCESS:

Requirements are capabilities and conditions to which the system, i.e. the project—must conform. The UP promotes a set of best practices, one of which is manage requirements.

To review the UP best practice *manage requirements*:

...a systematic approach to finding, documenting, organizing, and tracking the changing requirements of a system. [RUP]

Types of Requirements

In the UP, requirements are categorized according to the FURPS+ model, a useful mnemonic with the following meaning

- **Functional**—features, capabilities, security.
- **Usability**—human factors, help, documentation.
- **Reliability**—frequency of failure, recoverability, predictability.
- **Performance**—response times, throughput, accuracy, availability, resource usage.
- **Supportability**—adaptability, maintainability, internationalization, configurability.

Need for FURPS+ model:

It is helpful to use FURPS+ categories (or some categorization scheme) as a checklist for requirements coverage, to reduce the risk of not considering some important facet of the system.

Data dictionary: In unified process, a data dictionary is mainly used. It records requirements related to data, such as validation rules, acceptable values, and so forth.

Prototypes are a mechanism to clarify what is wanted or possible.

1.3 UML DIAGRAMS

- ★ **UML diagram** is a partial graphical representation (view) of a model of a system under design, implementation.
- ★ UML diagram contains **graphical elements** (symbols) - UML nodes connected with edges (also known as paths or flows) - that represent elements in the UML model of the designed system.
- ★ The UML model of the system might also contain other documentation such as use cases written as template texts.

UML diagrams represent two different views of a system model:

- **Static (or structural) view:** emphasizes the static structure of the system using objects, attributes, operations and relationships. The structural view includes class diagrams and composite structure diagrams.
- **Dynamic (or behavioural) view:** emphasizes the dynamic behaviour of the system by showing collaborations among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams and state machine diagrams.

UML diagrams are broadly classified into two categories – STRUCTURAL and BEHAVIORAL diagrams.

Structure diagrams

- ✓ Structure diagrams emphasize the things that must be present in the system being modelled.
- ✓ Since structure diagrams represent the structure, they are used extensively in documenting the software architecture of software systems.
- ✓ For example, the component diagram which describes how a software system is split up into components and shows the dependencies among these components.

The various Structural UML diagrams are:

- ☐ Class diagram
- ☐ Component diagram
- ☐ Deployment diagram
- ☐ Package diagram

Behaviour diagrams

- ★ Behaviour diagrams emphasize what must happen in the system being modelled.
- ★ Since behaviour diagrams illustrate the behaviour of a system, they are used extensively to describe the functionality of software systems.
- ★ As an example, the activity diagram describes the business and operational step-by-step activities of the components in a system.
- ★ The various Behavioural UML diagrams are:
 - Activity diagram
 - State Machine diagram
 - Interaction diagram
 - Use case diagram

Interaction diagrams

- ☐ Interaction diagrams, a subset of behavior diagrams, emphasize the flow of control and data among the things in the system being modelled.
- ☐ For example, the sequence diagram which shows how objects communicate with each other in terms of a sequence of messages.
- ☐ The interaction diagrams are in turn classified as :
 - \$ Sequence diagram
 - \$ Communication/ Collaboration diagram

1.4 USE CASES

Use case view of the system:

- ▶ The **use case view** captures the behavior of a **system**, **subsystem**, or **class** as it appears to an outside user.
- ▶ It partitions the system functionality into transactions meaningful to **actors**—idealized users of a system.
- ▶ The pieces of interactive functionality are called **use cases**.

Use case model:

- ▶ It is a model that describes a system's functional requirements in terms of use cases.
- ▶ It is a model of the system's functionality and environment.

Need for use case model:

- ▶ Writing use cases—stories of using a system—is an excellent technique to understand and describe requirements.
- ▶ The UP defines the **Use-Case Model** within the Requirements discipline.
- ▶ Use Cases are to discover and record functional requirements.

Elements of use case model:

The various elements of use case view are: actors, system, use cases, scenarios and relationships.

Use cases:

- ▶ A use case captures **a contract between the stakeholders of a system about its behavior**.
- ▶ A use case is a sequence of actions a system performs that yields an observable result of value to a particular actor.
- ▶ In UML use-case diagrams, a use case is rendered as an ellipse.

- ▶ Every use case must have a name that distinguishes it from other use cases.
- ▶ Rules of naming Use Case: **Verb + Noun**



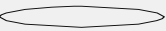


Actors:

- ▶ An actor represents anything that interacts with the system.
- ▶ The **actor** initiates an interaction with the system to accomplish some goal. The system responds, protecting the interests of all the stakeholders.
- ▶ Actors are not only roles played by people, but organizations, software, and machines.
- ▶ Actors are broadly classified into three categories:
 - **Primary actor** has user goals fulfilled through using services of the system.
 - For example, the cashier.
 - **Supporting actor** provides a service (for example, information) to the system. Often a computer system, but could be an organization or person.
 - For example, the automated payment authorization service
 - **Offstage actor** has an interest in the behavior of the use case, but is not primary or supporting
 - For example, a government tax agency.

Scenario:

- ▶ A scenario is a specific sequence of actions and interactions between actors and the system under discussion.

Notation used in a use case diagram:

| Symbol | Meaning / Usage |
|---|---|
|  <actor name> | Actor: human user |
|  <actor name> | Actor: external hardware, machine or system |
|  <use case name> | Use Case |
|  | Association: <i>communication</i> between Actors and Use Cases |
|  <association name> | Association: <i>extends or uses</i> relationships between Use Cases <i>inheritance</i> relationships between Actors. |

Use case Formality types:

- ♦ Use cases are written in different formats, depending on need. Use cases are written in varying degrees *of formality*:
 - ♦ **Brief** - terse one-paragraph summary, usually of the main success scenario.
 - ♦ **Casual** - informal paragraph format. Multiple paragraphs that cover various scenarios.
 - ♦ **Fully dressed** - the most elaborate. All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees //refer to 127/960 in textbook

Use case Template:

| Use Case Section | Comment |
|-------------------------------------|--|
| Use Case Name | Start with a verb. |
| Scope | The system under design. |
| Level | "user-goal" or "subfunction" |
| Primary Actor | Calls on the system to deliver its services. |
| Stakeholders and Interests | Who cares about this use case, and what do they want? |
| Preconditions | What must be true on start, <i>and</i> worth telling the reader? |
| Success Guarantee | What must be true on successful completion, <i>and</i> worth telling the reader. |
| Main Success Scenario | A typical, unconditional happy path scenario of success. |
| Extensions | Alternate scenarios of success or failure. |
| Special Requirements | Related non-functional requirements. |
| Technology and Data Variations List | Varying I/O methods and data formats. |
| Frequency of Occurrence | Influences investigation, testing, and timing of implementation. |
| Miscellaneous | Such as open issues. |

How to find use cases?

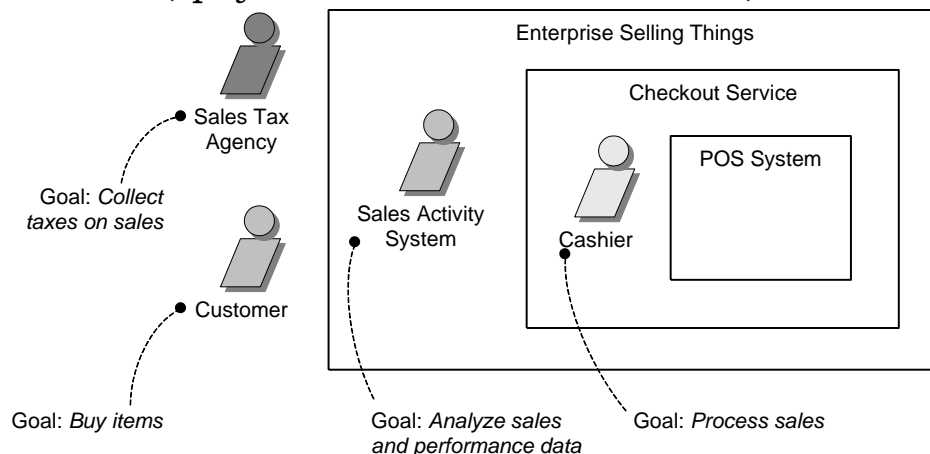
- ♦ 1. **Choose the system boundary.** Is it just a software application, the hardware and application as a unit, that plus a person using it, or an entire organization?

- ♦ 2. **Identify the primary actors.** Those that have user goals fulfilled through using services of the system.
- ♦ 3. For each, **identify their user goals.** Raise them to the highest user goal level that satisfies the EBP guideline.
- ♦ 4. **Define use cases that satisfy user goals;** name them according to their goal. Usually, user goal-level use cases will be one-to-one with user goals, but there is at least one exception, as will be examined.

EXAMPLE SCENARIO:

Process Sale: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.

1. **Identifying System Boundary:** For this case study, the POS system itself is the system under design; **everything outside of it is outside the system boundary**, including the cashier, payment authorization service, and so on.



Steps 2 and 3: Finding Primary Actors and Goals

This is done by answering the following questions:

Who starts and stops the system?

Who does system administration?

Who does user and security management?

Is there a monitoring process that restarts the system if it fails?

Who evaluates system activity or performance?

How is software updates handled? Push or pull update?

Who evaluates logs?

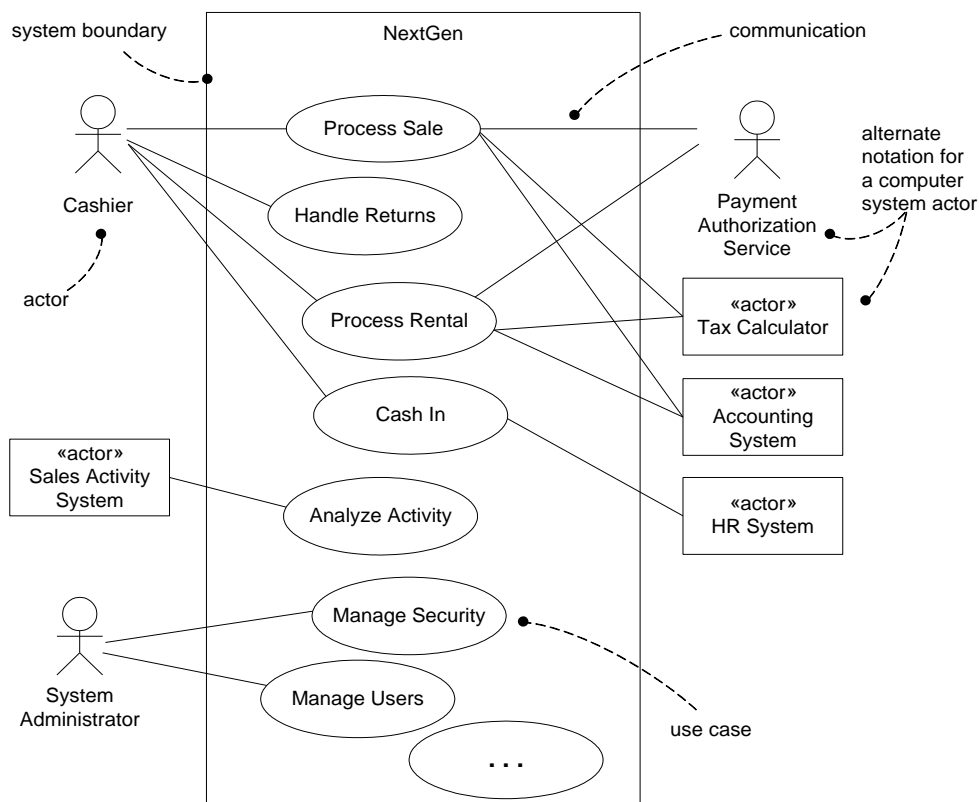
Are they remotely retrieved?

The Actor-Goal list:

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

4. **Finding the use cases:** Name the use case similar to the user goal (name use cases **starting with a verb**).

Thus the use case diagram is drawn as:



USE CASE RELATIONSHIPS:

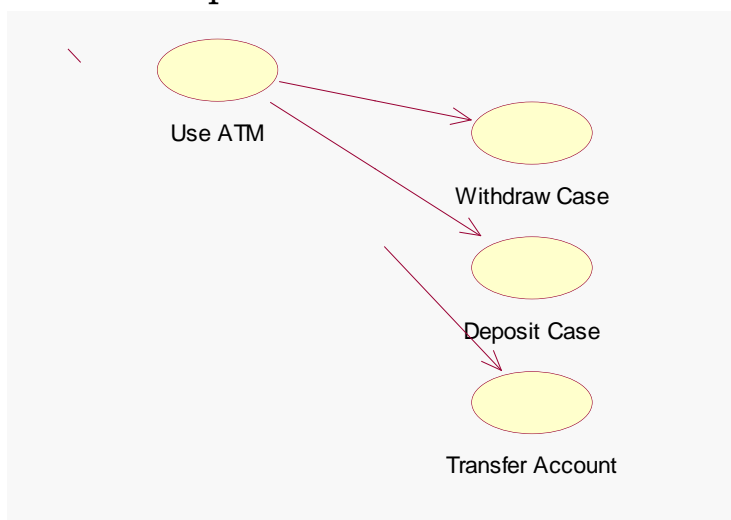
A use case can participate in several relationships, in addition to association with actors.

Table 5-1: Kinds of Use Case Relationships

| <i>Relationship</i> | <i>Function</i> | <i>Notation</i> |
|-------------------------|--|-------------------------|
| association | The communication path between an actor and a use case that it participates in | — |
| extend | The insertion of additional behavior into a base use case that does not know about it | «extend» - - - - -> |
| use case generalization | A relationship between a general use case and a more specific use case that inherits and adds features to it | —> |
| include | The insertion of additional behavior into a base use case that explicitly describes the insertion | «include» - - - - -> |

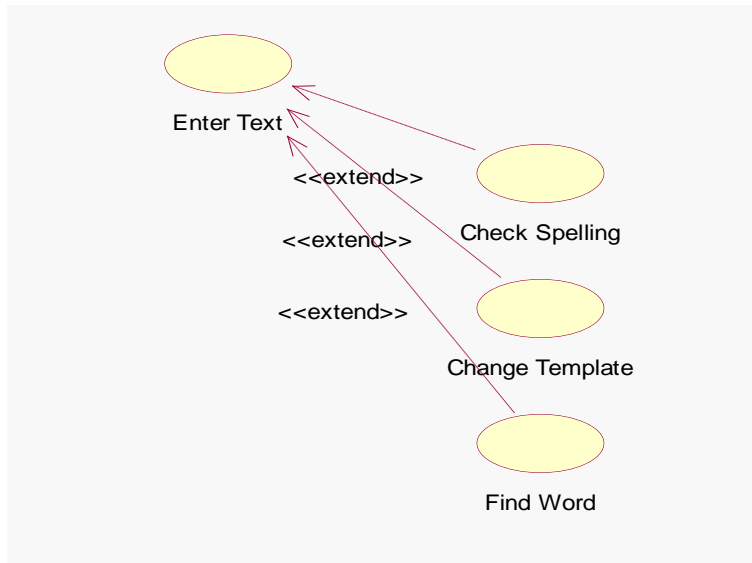
Include relationship:

- ❑ A use case can simply incorporate the behavior of other use cases as fragments of its own behavior. This is called an **include** relationship.
- ❑ In this case, the new use case is not a special case of the original use case and cannot be substituted for it.
- ❑ A include relationship from use case A to use case B indicates that A will also include the behavior as specified by B
- ❑ Example:



Extend relationship:

- ❑ A use case can also be defined as an incremental extension to a base use case. This is called an **extend** relationship.
- ❑ There may be several extensions of the same base use case that may all be applied together.
- ❑ An extends relationship from use case A to use case B indicates that an instance of B may include (subject to specific conditions) the behavior specified by A.



Use case generalisation:

- ✓ A use case can also be specialized into one or more child use cases. This is **use case generalization**.
- ✓ Any child use case may be used in a situation in which the parent use case is expected.

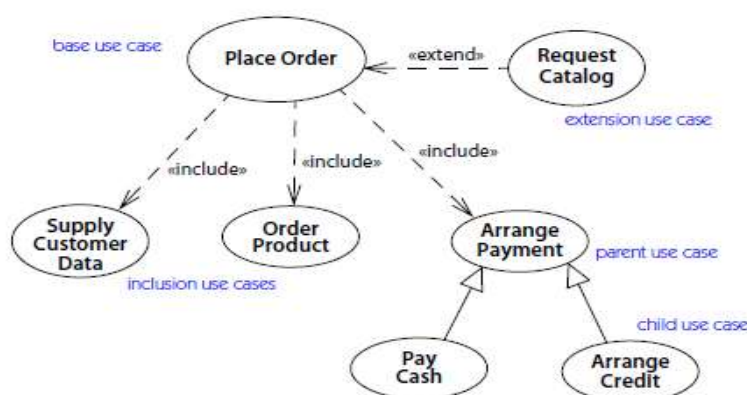


Figure 5-2. Use case relationships

Given above is a use case diagram with all relationships specified.

So in brief, the purposes of use case diagrams can be as follows:

- Used to gather requirements of a system.
 - Used to get an outside view of a system.
 - Identify external and internal factors influencing the system.
 - Show the interacting among the requirements are actors.
-

1.5 CLASS DIAGRAMS

Definition:

In software engineering, a **class diagram** in the Unified Modelling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

Need for class diagrams:

- ★ The class diagram is the main building block of object oriented modelling.
- ★ It is used both for general conceptual modelling of the systematics of the application, and for detailed modelling translating the models into programming code.
- ★ Class diagrams can also be used for data modelling.

Class:

- \$ A **class** represents a discrete concept within the application being modeled.
- \$ A class is the descriptor for a set of objects with similar structure, behavior, and relationships.
- \$ All **attributes** and **operations** are attached to classes or other **classifiers**.
- \$ A **class** defines a set of objects that have state and behavior.
- \$ State is described by **attributes** and **associations**.

Design Class Diagram:

A design class diagram illustrates the specifications for SW classes and interfaces.

Typical information included:

- β Classes, associations, and attributes
- β Interfaces, with their operations and constants
- β Methods
- β Attribute type information
- β Navigability
- β Dependencies

Attributes:

An attribute is a feature associated with an object of a class.

Each attribute must have a name.

If an attribute name is made up of more than one word, most OO languages require that you link it by hyphens or underscores.

attributeName : data_type=Default_value

Method:

A **method** is the implementation of an operation.

Operations are modules of procedural code that respond to messages by providing information.

The lifetime history of an object is described by a **state machine** attached to a class.

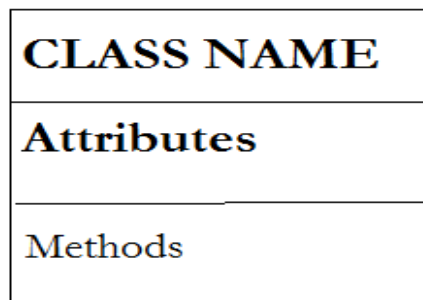
The full UML Syntax for operations is :

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

- This visibility marker is public (+) or private (-);
- The name is a string .
- The parameter-list is the list of parameters for the operation .
- The return-type is the type of the returned value, if there is one .
- The property-string indicates property values that apply to the given operation .

Notations for a class diagram:

A class is shown as solid-outline rectangle containing the class name and occasionally with compartments separated by horizontal lines. When class is shown with three compartments, the middle compartment holds a list of attributes and the bottom compartment holds a list of operations.



Class diagrams in two perspectives:

In conceptual perspective, a class diagram can be used to visualise a domain model.

In software perspective, it is called design class diagram.

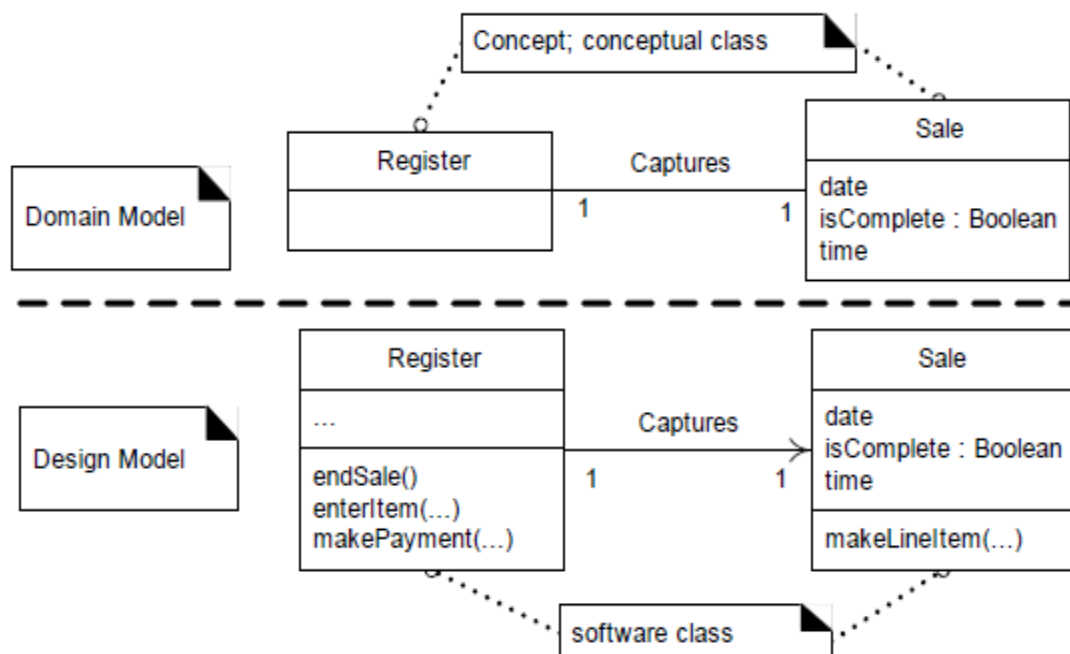


Figure 19.2 Domain model vs. Design Model classes.

Meaning of class in different perspectives:

The classes used in the above described perspectives have three different names. Hence we define:

1. **Conceptual class:** this type of class represents world class entities.
2. **Software classes:** this type of class represents the specification or implementation perspective of a software component.
3. **Implementation classes:** this type of class is implemented using some object oriented programming languages like Java, C++.

Steps in Creating a DCD:

The various steps in creating a design class diagram are as follows:

1. Identify all the classes participating in the SW solution by analyzing the interaction diagrams.
2. Draw them in a class diagram.
3. Duplicate the attributes from the associated concepts in the conceptual model.
4. Add method names by analyzing the interaction diagrams.
5. Add type information to the attributes and methods.
6. Add the associations necessary to support the required attribute visibility.
7. Add navigability arrows to the associations to indicate the direction of attribute visibility.
8. Add dependency relationship lines to indicate non-attribute visibility.

DCD FOR NextGEN POS SYSTEM:**1. Identify Software Classes and Illustrate Them**

- ❑ The first step in the creation of DCDs as part of the solution model is to identify those classes that participate in the software solution.
- ❑ These can be found by scanning all the interaction diagrams and listing the classes mentioned.

❑ For the POS application, these are:

- ▶ Register
- ▶ ProductCatalog
- ▶ Store Payment
- ▶ Sale
- ▶ ProductSpecification
- ▶ SalesLineItem

2. The next step is to draw a class diagram for these classes and include the attributes previously identified in the Domain Model that are also used in the design.

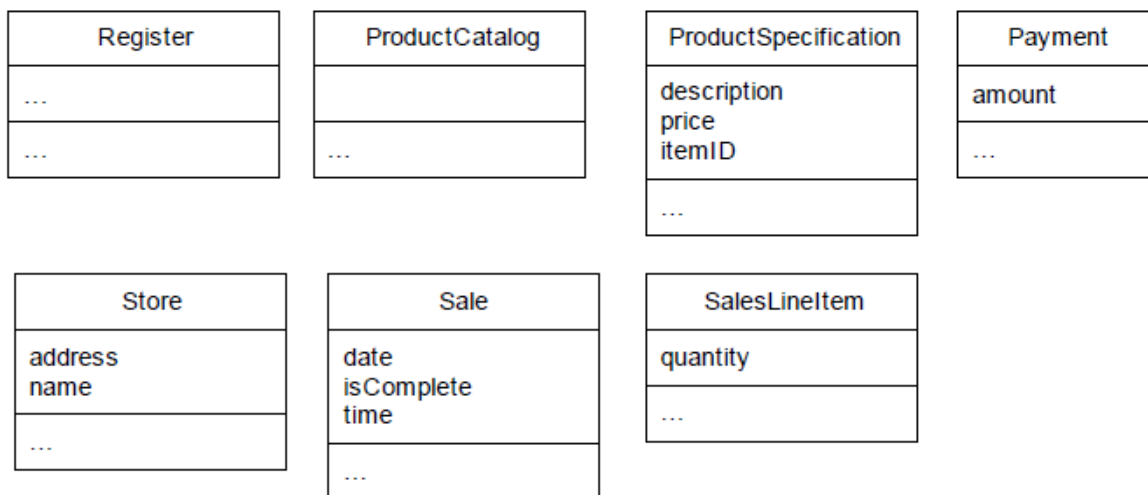


Figure 19.3 Software classes in the application.

3.Add Method Names

- ▶ The methods of each class can be identified by analyzing the interaction diagrams.
- ▶ For example, if the message makeLineItem is sent to an instance of class Sale, then class Sale must define a makeLineItem method.
- ▶ Inspection of all the interaction diagrams for the POS application yields the allocation of methods.

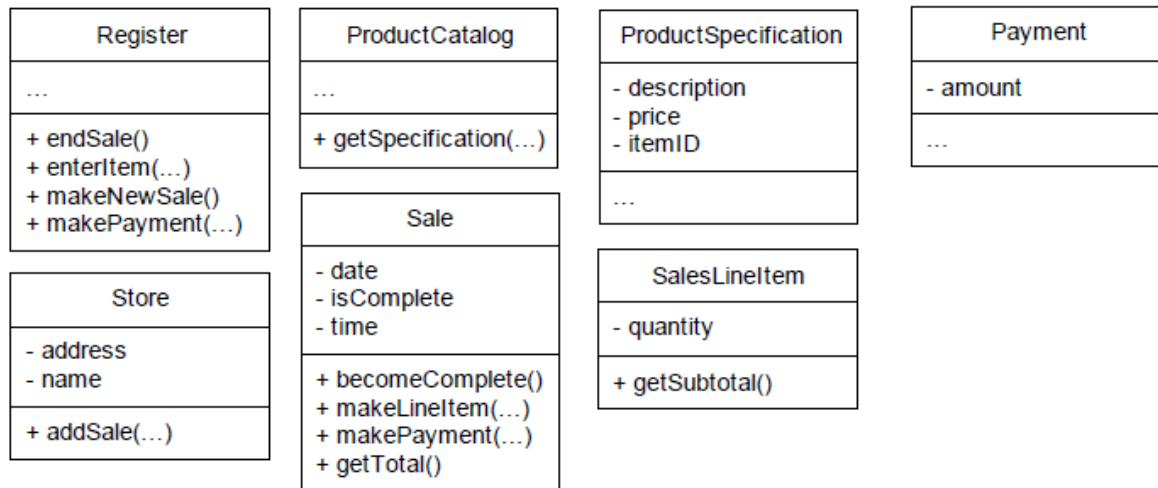


Figure 19.5 Methods in the application.

4. Adding Associations and Navigability

- Each end of an association is called a **role**.
- In the DCDs, the role may be decorated with a navigability arrow. Navigability is a property of the role that indicates that it is possible to navigate uni-directionally across the association from objects of the source to target class.
- Navigability implies visibility

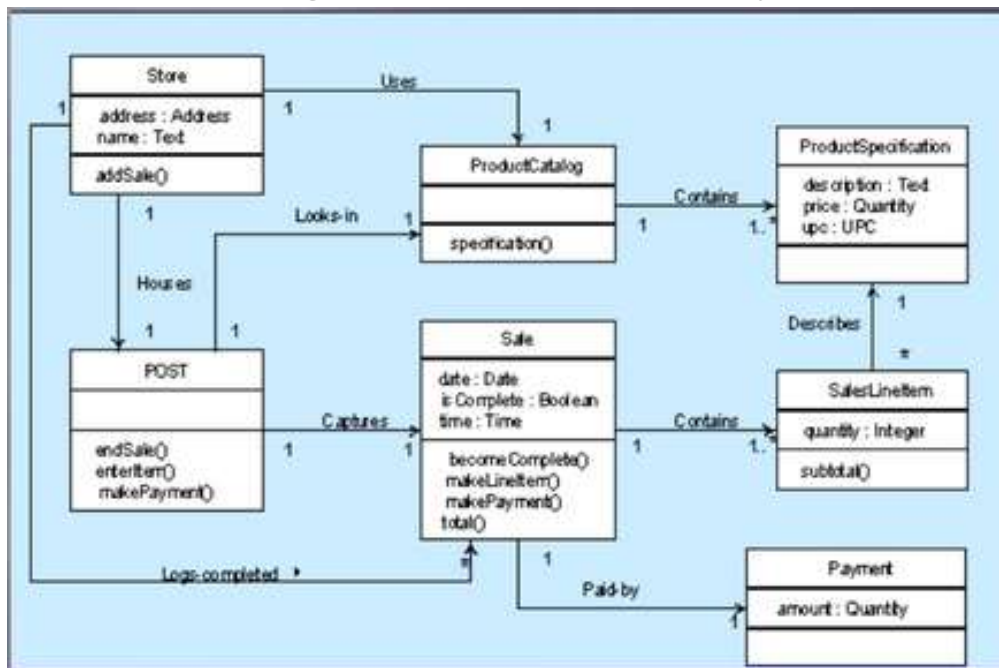
5. Adding Associations

- ❖ The required visibility and associations between classes are indicated by the interaction diagrams.
- ❖ Here are common situations suggesting a need to define an association with a navigability adornment from A to B:
 - A sends a message to B.
 - A creates an instance B.
 - A needs to maintain a connection to B.

6. Adding Dependency Relationships

- ★ The UML includes a general **dependency relationship**, which indicates that one element (of any kind, including classes, use cases, and so on) has knowledge of another element.
- ★ It is illustrated with a dashed arrow line.

SO the class diagram for NextGEN POS system is drawn as:



Class Diagrams in the Unified Process Phases

1. **Inception**—The Design Model and DCDs will not usually be started until elaboration because it involves detailed design decisions, which are premature during inception.
2. **Elaboration**—during this phase, DCDs will accompany the use-case realization interaction diagrams; they may be created for the most architecturally significant classes of the design.
3. It is recommended to generate DCDs regularly from the source code, to visualize the static structure of the system.
4. **Construction**—DCDs will continue to be generated from the source code as an aid in visualizing the static structure of the system.

How to show methods in class diagrams?

- ★ An UML method is an implementation of operation.
- ★ A method can be implemented in many ways including:
 - ❑ In interaction diagrams, by the details and sequences of messages
 - ❑ In class diagram, with an UML note stereotyped with <<method>>

Relationships in Class Diagrams:

A relationship is a general term covering the specific types of logical connections found on class and objects diagrams. UML shows the following relationships:

1) Instance level relationships

- I. **Dependency:** Dependency relationship indicates that one element (of any kind, including classes, use cases, and so on) has knowledge of another element.
 - a. A dependency is a using relationship that states a change in specification of one thing may affect another thing that uses it, but not necessarily the reverse.
- II. **Association:** An **association** describes discrete connections among objects or other instances in a system.
 - i. The most common kind of association is a **binary association** between a pair of classifiers.
 - ii. An **instance** of an association is a **link**.
- III. **Aggregation:** Aggregation is a variant of the "has an" association relationship; aggregation is more specific than association. It is an association that represents a part-whole or part-of relationship
- IV. **Composition :** part-of relationship exists.
Ex: Database is a part of System





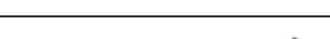
2) Class level relationships

1. **Generalisation:** It indicates that one of the two related classes (the subclass) is considered to be a specialized form of the other (the super type) and the superclass is considered a '**Generalization**' of the subclass.
 - i. A generalization is a relationship between a general thing (called the super class or parent) and a more specific kind of that thing (called the subclass or child).
 - ii. "is-a" kind-of relationship must exist.

2. Multiplicity

- a. The multiplicity of a property is an indication of how many objects may fill the property. The most common multiplicities you will see are
- i. 1 (An order must have exactly one customer .)
 - ii. 0..1 (A corporate customer may or may not have a single sales rep.)
 - iii. *(A customer need not place an Order and there is no upper limit to the number of Orders a Customer may place-zero or more orders .)

The relationships are denoted by :

| Relationship name | Symbol |
|-------------------|--|
| Association |  Association |
| Dependency |  Dependency |
| Aggregation |  Aggregation |
| Composition |  Composition |
| Generalisation |  Inheritance |

Purpose of Class Diagrams:

So the purpose of the class diagram can be summarized as:

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.
- Forward and reverse engineering.

