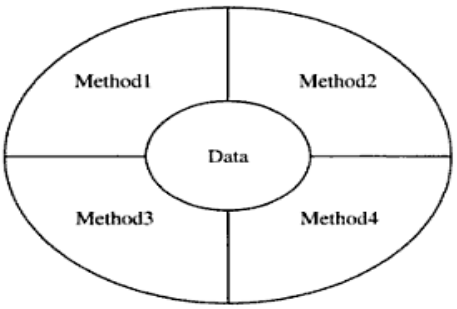
CS2353 OBJECT ORIENTED ANALYSIS AND DESIGN

UNIT 1

INTRODUCTION TO OBJECT ORIENTATION

Object Orientation is a term used to describe the object – oriented(OO) method of building software. In an OO approach, the data is treated as the most important element and it cannot flow freely around the system. Restrictions are placed on the number of units that can manipulate the data. This approach binds the data and the methods that will manipulate the data closely and prevents the data from being inadvertently modified. The following figure shows the method1, method2, method3, and method4.



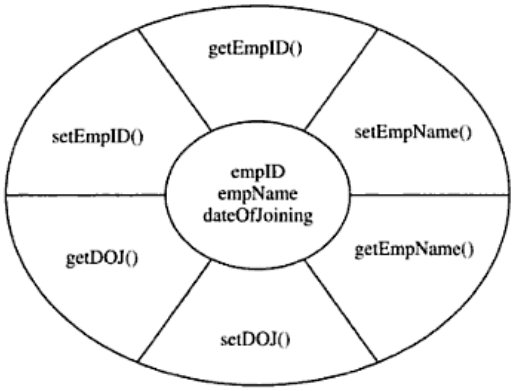
The 'object' forms the basis around which the following properties revolve:

1. Encapsulation
2. Abstraction, Implementation Hiding
3. Inheritance, dynamic binding, polymorphism
4. Overriding and overloading

Encapsulation:

In Object Orientation, a class is used as a unit to group related attributes and operations together. The outside world can interact with the data stored in the variables that represent the attributes of the class only through the operations of that class. Thus, the operations act as interfaces of the object of the class with the outside world.

For example, consider the class Employee with attributes empID, empName and dateOfJoining with is given below



Let the Employee class have the following operations:

setEmpID(employeeid: int)	assigns employeeid to empID
getEmpID():int	returns the value of empID
setEmpName(employeename: String)	assigns employeename to empName
getEmpName():String	returns the name of the employee
setDOJ(doj: Date)	assigns doj to date of joining
getDOJ():Date	returns the date of joining

For an object e1 of the type Employee, if it is necessary to set any value to attributes; then the methods setEmpID(employeeid:int), setEmpName(employeename:String) and setDOJ(doj:Date) must be used. Similarly, if it is necessary to know the value of any attribute, then the get operations have to be used. So, these operations act as the interface of the object e1 with the outside world. Thus, the attributes and operations are **encapsulated** within a **class**, and interaction with the attributes is done only through the interface provided by the **encapsulation.**

What is OOAD?

Object-oriented analysis and design (OOAD) is a software engineering approach that models a system as a group of interacting objects. Each object represents some entity of interest in the system being modeled, and is characterised by its class, its state (data elements), and its behavior. Various models can be created to show the static structure, dynamic behavior, and run-time deployment of these collaborating objects. There are a number of different notations for representing these models, such as the Unified Modeling Language (UML).

Object-oriented analysis (OOA) applies object-modelling techniques to analyze the functional requirements for a system. Object-oriented design (OOD) elaborates the analysis models to produce implementation specifications. OOA focuses on *what* the system does, OOD on *how* the system does it.

Object-oriented systems

An object-oriented system is composed of objects. The behavior of the system results from the collaboration of those objects. Collaboration between objects involves them sending messages to each other. Sending a message differs from calling a function in that when a target object receives a message, it itself decides what function to carry out to service that message. The same message may be implemented by many different functions, the one selected depending on the state of the target object.

The implementation of "message sending" varies depending on the architecture of the system being modeled, and the location of the objects being communicated with.

Object-oriented analysis

Object-oriented analysis (OOA) looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed. Analysis models do not consider any implementation constraints that might exist, such as concurrency, distribution, persistence, or how the system is to be built. Implementation constraints are dealt during object-oriented design (OOD). Analysis is done before the Design[*citation needed*].

The sources for the analysis can be a written requirements statement, a formal vision document, interviews with stakeholders or other interested parties. A system may be divided into multiple domains, representing different business, technological, or other areas of interest, each of which are analyzed separately.

The result of object-oriented analysis is a description of *what* the system is functionally required to do, in the form of a conceptual model. That will typically be presented as a set of use cases, one or more UML class diagrams, and a number of interaction diagrams. It may also include some kind of user interface mock-up. The purpose of object oriented analysis is to develop a

model that describes computer software as it works to satisfy a set of customer defined requirements.

Object-oriented design

Object-oriented design (OOD) transforms the conceptual model produced in object-oriented analysis to take account of the constraints imposed by the chosen architecture and any non-functional – technological or environmental – constraints, such as transaction throughput, response time, run-time platform, development environment, or programming language.
The concepts in the analysis model are mapped onto implementation classes and interfaces. The result is a model of the solution domain, a detailed description of *how* the system is to be built.

What is UML?

Unified Modelling Language (UML) is the set of notations,models and diagrams used when developing object-oriented (OO) systems.

UML is the industry standard OO visual modelling language. The latest version is UML 1.4 and was formed from the coming together of three leading software methodologists; Booch, Jacobson and Rumbaugh.

UML allows the analyst ways of describing structure, behaviour of significant parts of system and their relationships.

Unified Modeling Language (**UML**) is a standardized general-purpose modeling language in the field of software engineering. The standard is managed, and was created by, the Object Management Group. UML includes a set of graphic notation techniques to create visual models of software-intensive systems.

The Unified Modeling Language is commonly used to visualize and construct systems which are software intensive. Because software has become much more complex in recent years, developers are finding it more challenging to build complex applications within short time periods. Even when they do, these software applications are often filled with bugs, and it can take programmers weeks to find and fix them. This is time that has been wasted, since an approach could have been used which would have reduced the number of bugs before the application was completed.

However, it should be emphasized that UML is not limited simply modeling software. It can also be used to build models for system engineering, business processes, and organization structures. A special language called Systems Modeling Language was designed to handle systems which were defined within UML 2.0. The Unified Modeling Language is important for a number of reasons. First, it has been used as a catalyst for the advancement of technologies which are model driven, and some of these include Model Driven Development and Model Driven Architecture.

Because an emphasis has been placed on the importance of graphics notation, UML is proficient in meeting this demand, and it can be used to represent behaviors, classes, and aggregation. While software developers were forced to deal with more rudimentary issues in the past, languages like UML have now allowed them to focus on the structure and design of their software programs. It should also be noted that UML models can be transformed into various

other representations, often without a great deal of effort. One example of this is the ability to transform UML models into Java representations.

This transformation can be accomplished through a transformation language that is similar to QVT. Many of these languages may be supported by OMG. The Unified Modeling Language has a number of features and characteristics which separate it from other languages within the same category. Many of these attributes have allowed it to be useful for developers. In this article, I intend to show you many of these attributes, and you will then understand why the Unified Modeling Language is one of the most powerful languages in existence today.

Unified Process

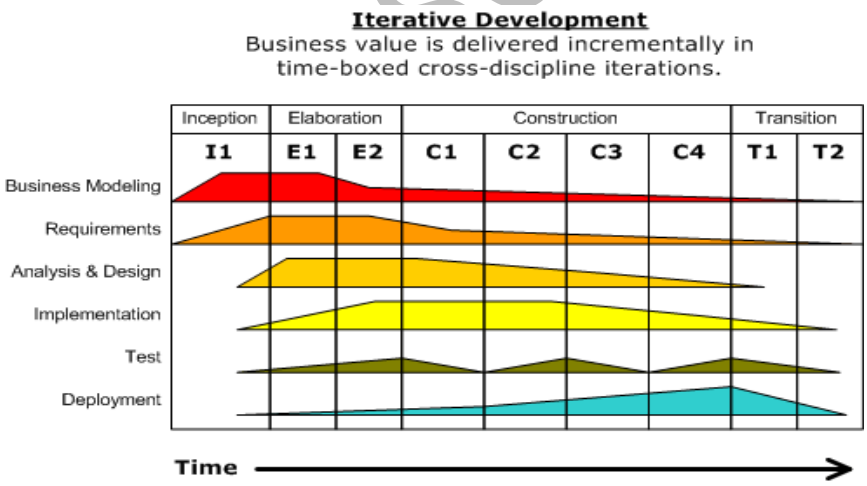
The **Unified Software Development Process** or *Unified Process* is a popular iterative and incremental software development process framework. The best-known and extensively documented refinement of the Unified Process is the Rational Unified Process (RUP).

Profile of a typical project showing the relative sizes of the four phases of the Unified Process.

Overview

The Unified Process is not simply a process, but rather an extensible framework which should be customized for specific organizations or projects. The *Rational Unified Process* is, similarly, a customizable framework. As a result it is often impossible to say whether a refinement of the process was derived from UP or from RUP, and so the names tend to be used interchangeably.

The name *Unified Process* as opposed to *Rational Unified Process* is generally used to describe the generic process, including those elements which are common to most refinements. The *Unified Process* name is also used to avoid potential issues of trademark infringement since *Rational Unified Process* and *RUP* are trademarks of IBM. The first book to describe the process was titled *The Unified Software Development Process* and published in 1999 by Ivar Jacobson, Grady Booch and James Rumbaugh. Since then various authors unaffiliated with Rational Software have published books and articles using the name *Unified Process*, whereas authors affiliated with Rational Software have favored the name *Rational Unified Process*.



Unified Process Characteristics

Iterative and Incremental

The Unified Process is an iterative and incremental development process. The Elaboration, Construction and Transition phases are divided into a series of timeboxed iterations. (The Inception phase may also be divided into iterations for a large project.) Each iteration results in

an *increment*, which is a release of the system that contains added or improved functionality compared with the previous release.

Although most iterations will include work in most of the process disciplines (*e.g.* Requirements, Design, Implementation, Testing) the relative effort and emphasis will change over the course of the project.

Use Case Driven

In the Unified Process, use cases are used to capture the functional requirements and to define the contents of the iterations. Each iteration takes a set of use cases or scenarios from requirements all the way through implementation, test and deployment.

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.

One of the most important deliverables of the process is the executable architecture baseline which is created during the Elaboration phase. This partial implementation of the system serves and validate the architecture and act as a foundation for remaining development.

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.

Project Lifecycle

The Unified Process divides the project into four phases:

- Inception
- Elaboration
- Construction
- Transition

Inception Phase

Inception is the smallest phase in the project, and ideally it should be quite short. If the Inception Phase is long then it may be an indication of excessive up-front specification, which is contrary to the spirit of the Unified Process.

The following are typical goals for the Inception phase.

- Establish a justification or business case for the project
- Establish the project scope and boundary conditions
- Outline the use cases and key requirements that will drive the design tradeoffs
- Outline one or more candidate architectures
- Identify risks
- Prepare a preliminary project schedule and cost estimate

The Lifecycle Objective Milestone marks the end of the Inception phase.

Elaboration Phase

During the Elaboration phase the project team is expected to capture a healthy majority of the system requirements. However, the primary goals of Elaboration are to address known risk factors and to establish and validate the system architecture. Common processes undertaken in this phase include the creation of use case diagrams, conceptual diagrams (class diagrams with only basic notation) and package diagrams (architectural diagrams).

The architecture is validated primarily through the implementation of an Executable Architecture Baseline. This is a partial implementation of the system which includes the core, most architecturally significant, components. It is built in a series of small, timeboxed iterations. By the end of the Elaboration phase the system architecture must have stabilized and the executable

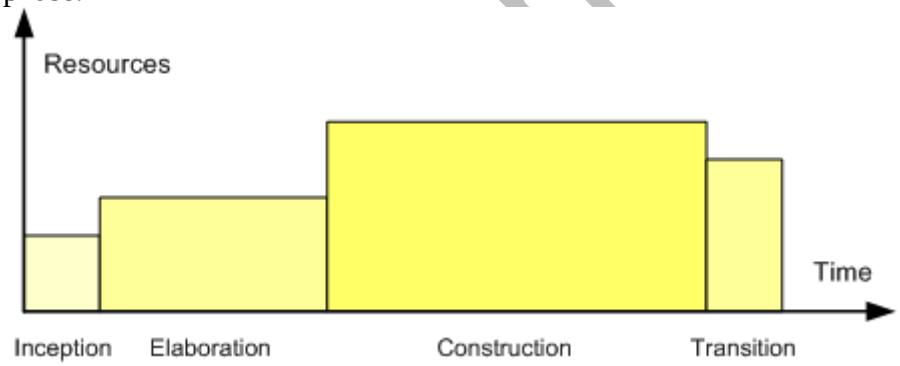
architecture baseline must demonstrate that the architecture will support the key system functionality and exhibit the right behavior in terms of performance, scalability and cost.
The final Elaboration phase deliverable is a plan (including cost and schedule estimates) for the Construction phase. At this point the plan should be accurate and credible, since it should be based on the Elaboration phase experience and since significant risk factors should have been addressed during the Elaboration phase.
The Lifecycle Architecture Milestone marks the end of the Elaboration phase.

Construction Phase

Construction is the largest phase in the project. In this phase the remainder of the system is built on the foundation laid in Elaboration. System features are implemented in a series of short, timeboxed iterations. Each iteration results in an executable release of the software. It is customary to write full text use cases during the construction phase and each one becomes the start of a new iteration. Common UML (Unified Modelling Language) diagrams used during this phase include Activity, Sequence, Collaboration, State (Transition) and Interaction Overview diagrams. The Initial Operational Capability Milestone marks the end of the Construction phase.

Transition Phase

The final project phase is Transition. In this phase the system is deployed to the target users. Feedback received from an initial release (or initial releases) may result in further refinements to be incorporated over the course of several Transition phase iterations. The Transition phase also includes system conversions and user training. The Product Release Milestone marks the end of the Transition phase.



Case Study: NextPOS System

The case study is the NextGen point-of-sale (POS) system. In this apparently straightforward problem domain, we shall see that there are very interesting requirement and design problems to solve. In addition, it is a realistic problem; organizations really do write POS systems using object technologies.

A POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and bar code scanner, and software to run the system. It interfaces to various service applications, such as a third-party tax calculator and inventory control. These systems must be relatively fault-tolerant; that is, even if
remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is
not crippled).

A POS system increasingly must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input, wireless PDAs, and so forth.

Furthermore, we are creating a commercial POS system that we will sell to different clients with disparate needs in terms of business rule processing. Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added. Therefore, we will need a mechanism to provide this flexibility and customization.

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

Architectural Layers and Case Study Emphasis

A typical object-oriented information system is designed in terms of several architectural layers or subsystems (see Figure 3.1). The following is not a complete list, but provides an example:

• **User Interface**—graphical interface; windows.

• **Application Logic and Domain Objects—**software objects representing domain concepts (for example, a software class named *Sale)* that fulfill application requirements.

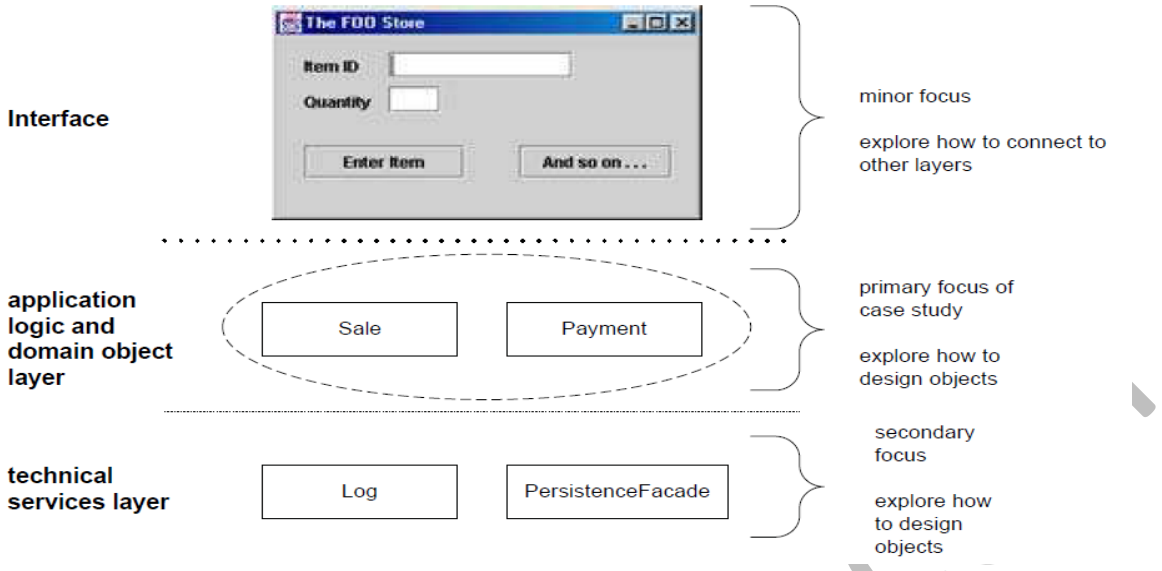
• **Technical Services**—general purpose objects and subsystems that provide supporting technical services, such as interfacing with a database or error logging. These services are usually application-independent and reusable across several systems.

OOA/D is generally most relevant for modeling the application logic and technical service layers.

The NextGen case study primarily emphasizes the problem domain objects, allocating responsibilities to them to fulfill the requirements of the application.

Object-oriented design is also applied to create a technical service subsystem for interfacing with a database.

In this design approach, the UI layer has very little responsibility; it is said to be *thin.* Windows do *not* contain code that performs application logic or processing. Rather, task requests are forwarded on to other layers.



Inception Phase

This is the part of the project where the original idea is developed. The amount of work done here is dependent on how formal project planning is done in your organization and the size of the project. During this part of the project some technical risk may be partially evaluated and/or eliminated. This may be done by using a few throw away prototypes to test for technical feasability of specific system functions. Normally this phase would take between two to six weeks for large projects and may be only a few days for smaller projects. The following should be done during this phase:

1. Project idea is developed.
2. Assess the capablilities of any current system that provides similar functionality to the new project even if the current system is a manual system. This will help determine cost savings that the new system can provide.
3. Utilize as many users and potential users as possible along with technical staff, customers, and management to determine desired system features, functional capabilities, and performance requirements. Analyze the scope of the proposed system.
4. Identify feature and functional priorities along with preliminary risk assessment of each system feature or function.
5. Identify systems and people the system will interact with.
6. For large systems, break the system down into subsystems if possible.
7. Identify all major use cases and describe significant use cases. No need to make expanded use cases at this time. This is just to help identify and present system functionality.
8. Develop a throw away prototype of the system with breadth and not depth. This prototype will address some of the greatest technical risks. The time to develop this prototype should be specifically limited. For a project that will take about one year, the prototype should take one month.
9. Present a business case for the project (white paper) identifying rough cost and value of the project. The white paper is optional for smaller projects. Define goals, estimate risks, and resources required to complete the project.

10. Set up some major project milestones (mainly for the elaboration phase). A rough estimate of the overall project size is made.
11. Preliminary determination of iterations and requirements for each iteration. This outlines system functions and features to be included in each iteration. Keep in mind that this plan will likely be changes as risks are further assessed and more requirements are determined.
12. Management Approval for a more serious evaluation of the project.

This phase is done once the business case is presented with major milestones determined (not cast in stone yet) and management approves the plan. At this point the following should be complete:

- Business case (if required) with risk assessment.
- Preliminary project plan with preliminary iterations planned.
- Core project requirements are defined on paper.
- Major use cases are defined.

The inception phase has only one iteration. All other phases may have multiple iterations. The overriding goal of the inception phase is to achieve concurrence among all stakeholders on the lifecycle objectives for the project. The inception phase is of significance primarily for new development efforts, in which there are significant business and requirements risks which must be addressed before the project can proceed. For projects focused on enhancements to an existing system, the inception phase is more brief, but is still focused on ensuring that the project is both worth doing and possible to do.

Objectives

The primary objectives of the Inception phase include:

Establishing the project's software scope and boundary conditions, including an operational vision, acceptance criteria and what is intended to be in the product and what is not.

Discriminating the critical use cases of the system, the primary scenarios of operation that will drive the major design tradeoffs.

Exhibiting, and maybe demonstrating, at least one candidate architecture against some of the primary scenarios

Estimating the overall cost and schedule for the entire project (and more detailed estimates for the elaboration phase that will immediately follow)

Estimating potential risks (the sources of unpredictability)

Preparing the supporting environment for the project.

Essential Activities

The essential activities of the Inception include:

Formulating the scope of the project. This involves capturing the context and the most important requirements and constraints to such an extent that you can derive acceptance criteria for the end product.

Planning and preparing a business case. Evaluating alternatives for risk management, staffing, project plan, and cost/schedule/profitability tradeoffs.

Synthesizing a candidate architecture, evaluating tradeoffs in design, and in make/buy/reuse, so that cost, schedule and resources can be estimated. The aim here is to demonstrate feasibility through some kind of proof of concept. This may take the form of a model which simulates what is required, or an initial prototype which explores what are considered to be the areas of high risk. The prototyping effort during inception should be

limited to gaining confidence that a solution is possible - the solution is realized during elaboration and construction.

Preparing the environment for the project, assessing the project and the organization, selecting tools, deciding which parts of the process to improve.

Milestone

The Lifecycle Objectives Milestone evaluates the basic viability of the project.

Tailoring Decisions

The example iteration workflow shown at the top of this page represents a typical Inception iteration in medium sized projects. The Sample Iteration Plan for Inception represents a different perspective of the breakdown of activities to undertake in an Inception iteration. This iteration plan is more complete in terms of workflow details and activities, and as such, more suitable for large projects. Small projects might decide to do only a subset of these workflow details, deviations should be challenged and documented as part of the project-specific process.

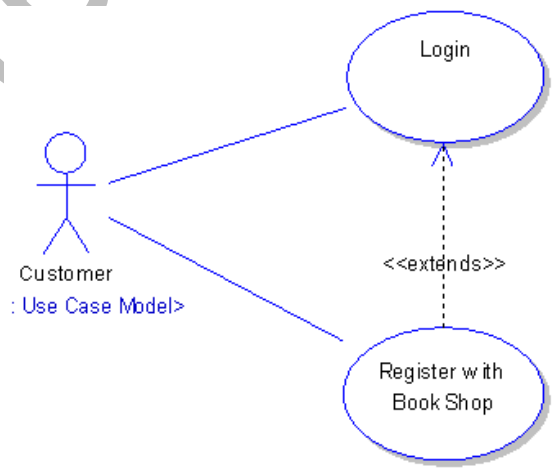
Inception Phase includes:

- Refining the scope of the project
- Project planning
- Risk identification and analysis
- Preparing the project environment
- Estimating the Budget

The Use Case Model

The Use Case Model describes the proposed functionality of the new system. A Use Case represents a discrete unit of interaction between a user (human or machine) and the system. A Use Case is a single unit of meaningful work; for example login to system, register with system and create order are all Use Cases. Each Use Case has a description which describes the functionality that will be built in the proposed system. A Use Case may 'include' another Use Case's functionality or 'extend' another Use Case with its own behaviour.

Use Cases are typically related to 'actors'. An actor is a human or machine entity that interacts with the system to perform meaningful work.

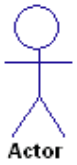


A Use Case description will generally include:

1. General comments and notes describing the use case;
2. Requirements - Things that the use case must allow the user to do, such as <ability to update order>, <ability to modify order> & etc.
3. Constraints- Rules about what can and can't be done. Includes i) pre-conditions that must be true before the use case is run -e.g. <create order> must precede <modify order>; ii) post-conditions that must be true once the use case is run e.g. <order is modified and consistent>; iii) invariants: these are always true - e.g. an order
4. Scenarios - Sequential descriptions of the steps taken to carry out the use case. May include multiple scenarios, to cater for exceptional circumstances and alternate processing paths;
5. Scenario diagrams -Sequence diagrams to depict the workflow - similar to (4) but graphically
6. Additional attributes such as implementation phase, version number, complexity rating,

Actors

An Actor is a user of the system. This includes both human users and other computer systems. An Actor uses a Use Case to perform some piece of work which is of value to the business. The set of Use Cases an actor has access to defines their overall role in the system and the scope of their action.



Constraints, Requirements and Scenarios

The formal specification of a Use Case includes:

1. Requirements. These are the formal functional requirements that a Use Case must provide to the end user. They correspond to the functional specifications found in structured methodologies. A requirement is a contract that the Use Case will perform some action or provide some value to the system.
2. Constraints. These are the formal rules and limitations that a Use Case operates under, and includes pre- post- and invariant conditions. A pre-condition specifies what must have already occurred or be in place before the Use Case may start. A post-condition documents what will be true once the Use Case is complete. An invariant specifies what will be true throughout the time the Use Case operates.
3. Scenarios. Scenarios are formal descriptions of the flow of events that occurs during a Use Case instance. These are usually described in text and correspond to a textual representation of the Sequence Diagram.

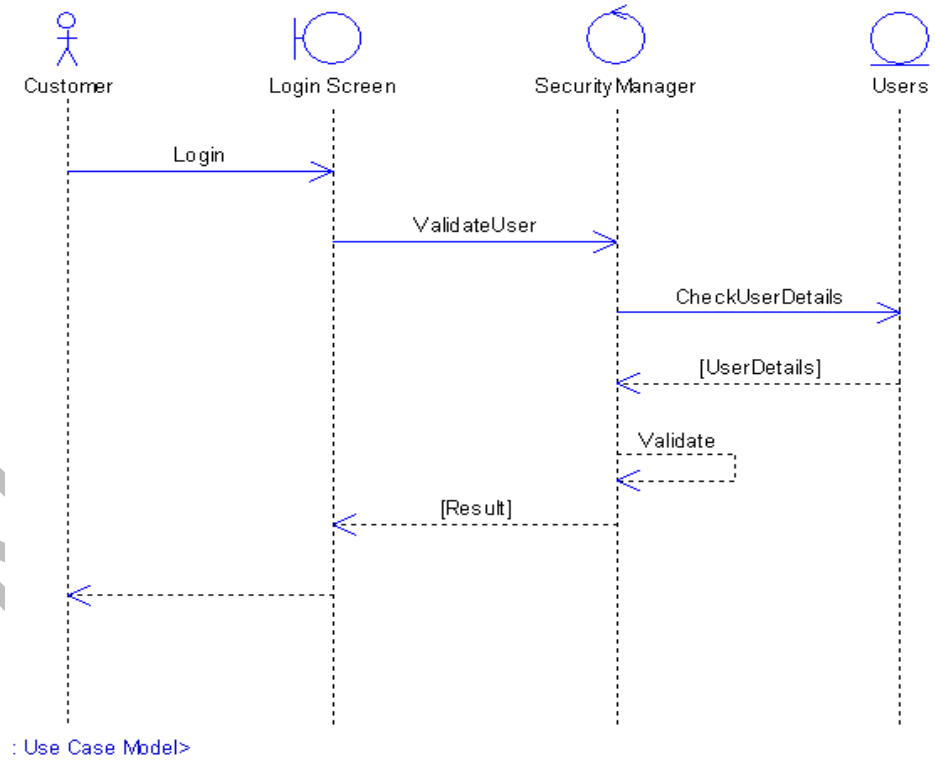
Includes and Extends relationships between Use Cases

One Use Case may include the functionality of another as part of its normal processing. Generally, it is assumed that the included Use Case will be called every time the basic path is run. An example may be to list a set of customer orders to choose from before modifying a selected order - in this case the <list orders> Use Case may be included every time the <modify order> Use Case is run.

A Use Case may be included by one or more Use Cases, so it helps to reduce duplication of functionality by factoring out common behaviour into Use Cases that are re-used many times.
One Use Case may extend the behaviour of another - typically when exceptional circumstances are encountered. For example, if before modifying a particular type of customer order, a user must get approval from some higher authority, then the <get approval> Use Case may optionally extend the regular <modify order> Use Case.

Sequence Diagrams

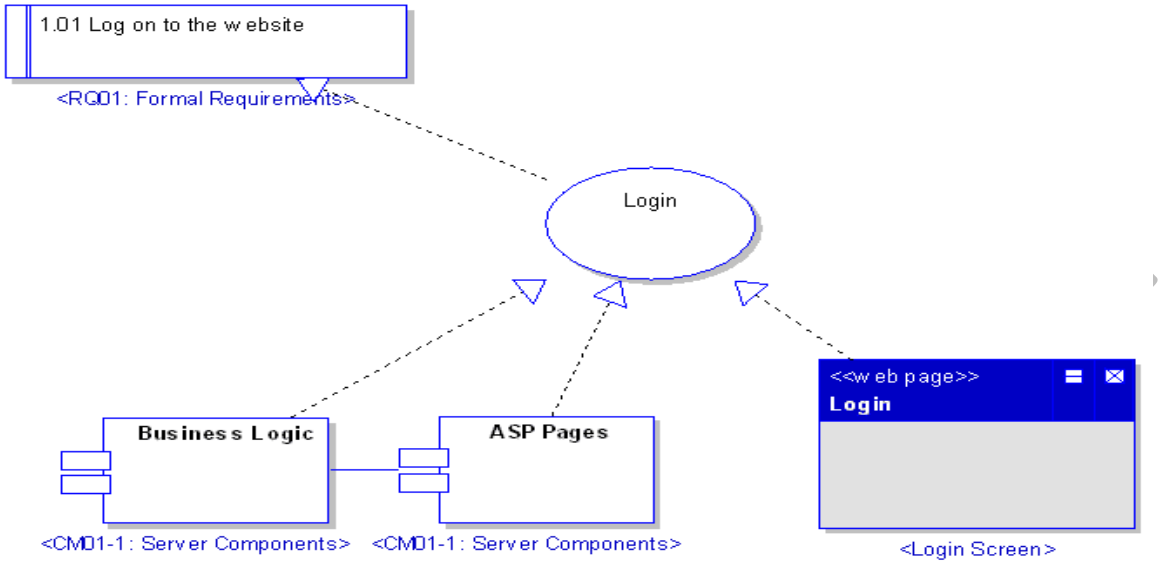
UML provides a graphical means of depicting object interactions over time in Sequence Diagrams. These typically show a user or actor, and the objects and components they interact with in the execution of a use case. One sequence diagram typically represents a single Use Case 'scenario' or flow of events.
Sequence diagrams are an excellent way to document usage scenarios and to both capture required objects early in analysis and to verify object usage later in design. Sequence diagrams show the flow of messages from one object to another, and as such correspond to the methods and events supported by a class/object.
The diagram illustrated below shows an example of a sequence diagram, with the user or actor on the left initiating a flow of events and messages that correspond to the Use Case scenario. The messages that pass between objects will become class operations in the final model.



Implementation Diagram

A Use Case is a formal description of functionality the system will have when constructed. An implementation diagram is typically associated with a Use Case to document what design elements (eg. components and classes) will implement the Use Case functionality in the new system. This provides a high level of traceability for the system designer, the

customer and the team that will actually build the system. The list of Use Cases that a component or class is linked to documents the minimum functionality that must be implemented by the component.



The example above shows that the Use Case "Login" implements the formal requirement "1.01 Log on to the website". It also states that the Business Logic component and ASP Pages component implement some or all of the Login functionality. A further refinement is to show the Login screen (a web page) as implementing the Login interface. These implementation or realisation links define the traceability from the formal requirements, through Use Cases on to Components and Screens.

Relationships Between Use Cases

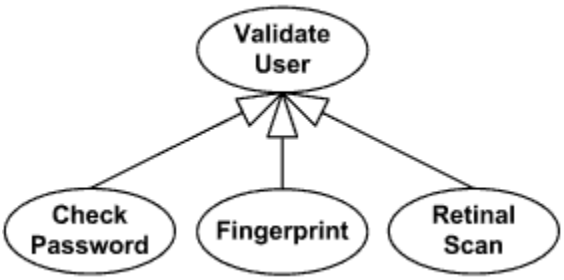
Use cases could be organized using following relationships:

- Generalization
- Association
- Extend
- Include

Generalization Between Use Cases

Generalization between use cases is similar to generalization between classes – child use case inherits properties and behavior of the parent use case and may override the behavior of the parent.

Notation: Generalization is rendered as a solid directed line with a large open arrowhead (same as generalization between classes).



Generalization between use cases

Association Between Use Cases

Use cases can only be involved in **binary** Associations. Two use cases specifying the same subject cannot be associated since each of them individually describes a complete usage of the system.

Extend Relationship

Extend is a **directed relationship** from an **extending use case** to an **extended use case** that specifies how and when the behavior defined in usually supplementary (optional) extending use case can be inserted into the behavior defined in the use case to be extended.

Note: **Extended use case is meaningful** on its own, independently of the extending use case, while the **extending use case** typically defines behavior that is **not necessarily meaningful by itself**.

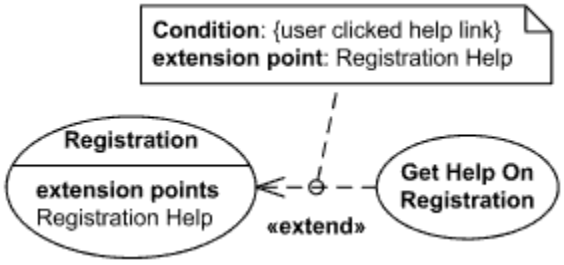
The extension takes place at one or more extension points defined in the **extended use case**.

The extend relationship is **owned** by the extending use case. The same extending use case can extend more than one use case, and extending use case may itself be extended.

Extend relationship between use cases is shown by a dashed arrow with an open arrowhead from the **extending use case** to the **extended (base) use case**. The arrow is labeled with the keyword **«extend»**.

Registration use case is meaningful on its own, and it could be extended with optional **Get Help On Registration** use case

The **condition** of the extend relationship as well as the references to the extension points are optionally shown in a **Note** attached to the corresponding extend relationship.



Registration use case is conditionally extended by Get Help On Registration use case in extension point Registration Help

Extension Point

An **extension point** is a **feature** of a use case which identifies (references) a point in the behavior of the use case where that behavior can be extended by some other (extending) use case, as specified by an extend relationship.

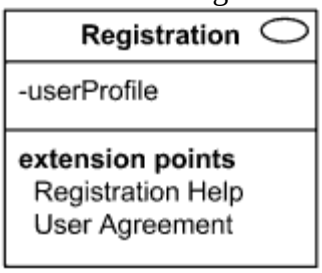
Extension points may be shown in a compartment of the use case oval symbol under the heading **extension points**. Each extension point must have a **name**, unique within a use case. Extension points are shown as text string according to the syntax:

<extension point> ::= <name> [: <explanation>]

The optional description is given usually as informal text, but can also be given in other forms, such as the name of a state in a state machine, an activity in an activity diagram, a precondition, or a postcondition.

Registration **use case** with **extension points** Registration Help and User Agreement

Extension points may be shown in a compartment of the use case rectangle with **ellipse icon** under the heading **extension points**.



Extension points of the Registration **use case** shown using the rectangle notation

Include Relationship

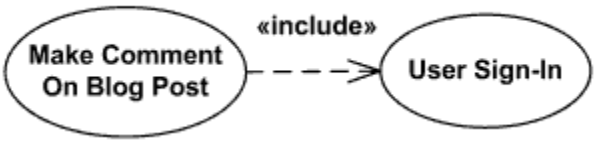
An **include** relationship is a **directed relationship** between two use cases, implying that the behavior of the required (not optional) **included** use case is inserted into the behavior of the **including** (base) use case. Including use case **depends on** the addition of the included use case.

The include relationship is intended to be used when there are **common parts** of the behavior of two or more use cases. This common part is extracted into a separate use case to be included by all the base use cases having this part in common.

Execution of the included use case is analogous to a subroutine call or macro command in programming. All of the behavior of the included use case is executed at a single location in the including use case before execution of the including use case is resumed.

As the primary use of the include relationship is to **reuse common parts**, including use cases are usually **not complete** by themselves but dependent on the included use cases.

Include relationship between use cases is shown by a dashed arrow with an open arrowhead from the including (base) use case to the included (common part) use case. The arrow is labeled with the keyword **«include»**.



Four Phases of Unified Process

The Unified Process consists of cycles that may repeat over the long-term life of a system. A cycle consists of four phases: Inception, Elaboration, Construction and Transition. Each cycle is concluded with a release, there are also releases within a cycle. Let's briefly review the four phases in a cycle:

* **Inception Phase** - During the inception phase the core idea is developed into a product vision. In this phase, we review and confirm our understanding of the core business drivers. We want to understand the business case for why the project should be attempted. The inception phase establishes the product feasibility and delimits the project scope.

* **Elaboration Phase** - During the elaboration phase the majority of the Use Cases are specified in detail and the system architecture is designed. This phase focuses on the "Do-Ability" of the project. We identify significant risks and prepare a schedule, staff and cost profile for the entire project.

* **Construction Phase** - During the construction phase the product is moved from the architectural baseline to a system complete enough to transition to the user community. The architectural baseline grows to become the completed system as the design is refined into code.

* **Transition Phase** - In the transition phase the goal is to ensure that the requirements have been met to the satisfaction of the stakeholders. This phase is often initiated with a beta release of the application. Other activities include site preparation, manual completion, and defect identification and correction. The transition phase ends with a postmortem devoted to learning and recording lessons for future cycles.

UNIT 2 - Elaboration

Elaboration Phase

The primary purpose of this phase is to complete the most essential parts of the project that are high risk and plan the construction phase. This is the part of the project where technical risk is fully evaluated and/or eliminated by building the highest risk parts of the project. During this phase personnel requirements should be more accurately determined along with estimated man hours to complete the project. The complete cost and time frame of the project is more firmly determined. During this phase how the system will work must be considered. Use cases will help identify risks. Steps to take during this phase:

1. Complete project plan with construction iterations planned with requirements for each iteration.
2. 80% of use cases are completed. Significant use cases are described in detail.
3. The project domain model is defined. (Don't get bogged down)
4. Rank use cases by priority and risk. Do the highest priority and highest risk use cases first. Items that may be high risk:
    - Overall system architecture especially when dealing with communication between subsystems.
    - Team structure.
    - Anything not done before or used before such as a new programming language, or using the unified/iterative process for the first time.
5. Begin design and development of the riskiest and highest priority use cases. There will be an iteration for each high risk and priority use case.
6. Plan the iterations for the construction phase. This involves choosing the length of the iterations and deciding which use cases or parts of use cases will be implemented during each iteration. Develop the higher priority and risk use cases during the first iterations in the construction phase.

As was done on a preliminary level in the previous phase, the value (priority) of use cases and their respective risks must be more fully assessed in this phase. This may be done be either assigning an number to each use case for both value and risk. or categorize them by high, medium, or low value and risk. Time required for each use case should be estimated to the man week. Do the highest priority and highest risk use cases first.

Requirements to be completed for this phase include:

- Description of the software architecture. Therefore most use cases should be done, activity diagrams, state charts, system sequence diagrams, and the domain model should be mostly complete.
- A prototype that overcomes the greatest project technical risk and has minimal high priority functionality.
- Complete project plan.
- Development plan.

There may be an elaboration phase for each high risk use case.

Considering the various diagrams and charts to be created, when they are created, and the best order to create them in, there seems to be a variety of opinions. This is because in the real world there may be more than one correct solution and there are no hard and fast rules that work everytime. In a way, this flexibility is a strength of UML. Some documentation indicates that

most use cases should be done before creating a domain model and others indicate that the domain model can be built on a use case by use case basis. A good compromise is to spend a short time on a brief domain model during the elaboration phase, then enhance the domain model as each use case is developed during the elaboration and construction phase iterations. Some documentation indicates that activity diagrams and class diagrams should be complete before the domain model is done. It is possible to create some of the diagrams and charts (artifacts) in parallel with each otherCompletion of 80% of use case diagrams.

1. Completion of 80% of high level use case diagrams.
2. Completion of expanded use case diagrams for major use cases only.
3. System sequence diagrams for major use cases.
4. Domain model (Don't get bogged down here with details). Just get a good idea of concepts involved. Use use cases to create the domain model. Any use case that strongly impacts the domain model should be considered and concepts from that use case should be incorporated in the domain model. The initial domain model may be drawn without lines and attributes to avoid too much detail and determine important use cases. The domain model may be refined later as the project analysis continues. If the system is large, domain models should be done on a per use case basis.
5. Optionally create a glossary of terms for concepts to improve team communication.

After this point the design part of the project begins (although more analysis is done for each use case) and the following will be done in each iteration of the elaboration and construction phases.

1. Operation contracts based on domain model and use cases.
2. Collaboration diagrams.
3. Class diagrams.
4. Map class and collaboration diagrams to code.
5. Update the domain model but do not force it to the class diagrams.

Considerations during this project should be the following:

- Consider possible significant changes (down the road) to the system during analysis.
- Regarding system functional ability what do you expect to be able to change?

I. Elaboration Phase:

Elaboration phase plans the necessary activities and required resources and specifies the features and designing the architecture.

- Things to do: With the input of the use case model generated from the previous phase, we transform it into a design model via an analysis model. In brief, both an analysis model and a design model are structures made up of *classifiers* and a set of use-case realizations that describe how this structure realizes the use cases. Classifiers are, in general, "class-like" things.

  The analysis model is a detailed specification of the requirements and works as a first cut at a design model, although it is a model of its own. It is used by developers to understand precisely the use cases as described in the requirements. The analysis model is different from the design model in that it is a conceptual model rather than a blueprint of the implementation.
  - Class Diagrams
  - Sequence Diagrams
  - Collaboration Diagrams
- Exit Criteria:

- A detailed software development plan, containing:
  - An updated risk assessment,
  - A management plan,
  - A staffing plan,
  - A phase plan showing the number and contents of the iteration
  - An iterative plan, detailing the next iteration
  - The development environment and other tools required
  - A test plan
- A baseline vision, in the form of a set of evalution criteria for the final product
- Objective, measurable evalution criteria for assessing the results of the initial iterations of the construction phase
- A domain analysis model (80% complete), sufficient to be able to call the corresponding architecture 'complete'.
- A software architecture description (stating constraints and limitations)
- An executable architectural baseline.

- **Glossary:**
  - *baseline:* a release that is subject to change management and configuration control.
  - *stereotype:* an extension of the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your particular problem.

Elaboration Phase

During the Elaboration phase the project team is expected to capture a healthy majority of the system requirements. However, the primary goals of Elaboration are to address known risk factors and to establish and validate the system architecture. Common processes undertaken in this phase include the creation of use case diagrams, conceptual diagrams (class diagrams with only basic notation) and package diagrams (architectural diagrams).

The architecture is validated primarily through the implementation of an Executable Architecture Baseline. This is a partial implementation of the system which includes the core, most architecturally significant, components. It is built in a series of small, timeboxed iterations. By the end of the Elaboration phase the system architecture must have stabilized and the executable architecture baseline must demonstrate that the architecture will support the key system functionality and exhibit the right behavior in terms of performance, scalability and cost.

The final Elaboration phase deliverable is a plan (including cost and schedule estimates) for the Construction phase. At this point the plan should be accurate and credible, since it should be based on the Elaboration phase experience and since significant risk factors should have been addressed during the Elaboration phase.

The Lifecycle Architecture Milestone marks the end of the Elaboration phase.

Domain Model

A **domain model**, or **Domain Object Model** (DOM) in problem solving and software engineering can be thought of as a conceptual model of a domain of interest (often referred to as a problem domain) which describes the various entities, their attributes and relationships, plus the constraints that govern the integrity of the model elements comprising that problem domain.

Overview

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 Case models.

An important benefit 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 is especially helpful as a communication tool and a focusing point both amongst the different members of the business team as well as between the technical and business teams.

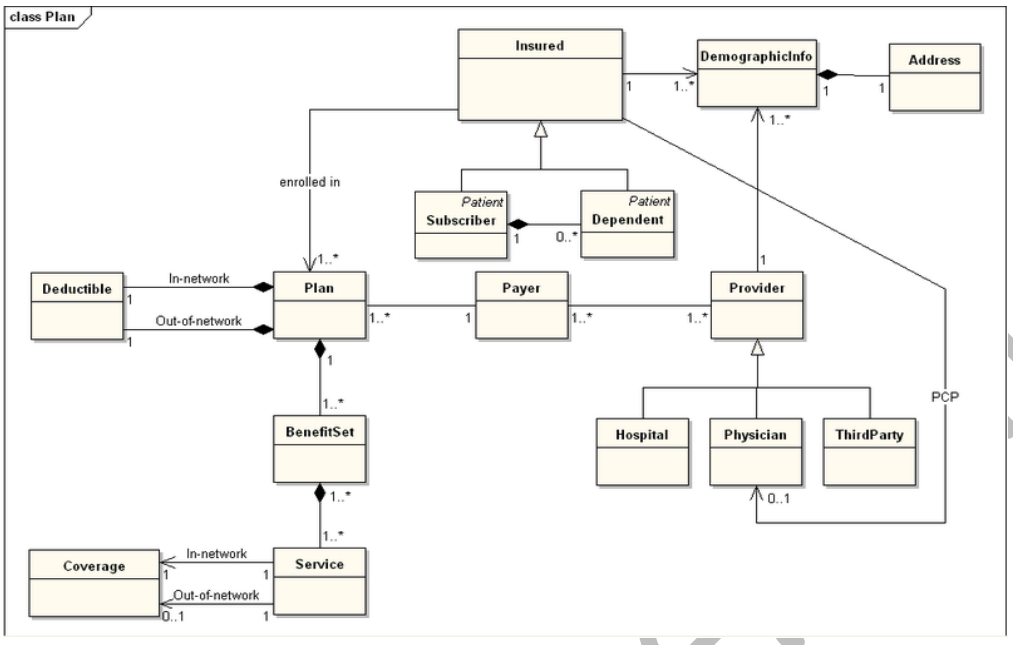
Usage

A well-thought domain model serves as a clear depiction of the conceptual fabric of the problem domain and therefore is invaluable to ensure all stakeholders are aligned in the scope and meaning of the concepts indigenous to the problem domain. A high fidelity 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 UML, a class diagram is used to represent the domain model. In Domain-driven design, the domain model (Entities and Value objects) is a part of the Domain layer which often also includes other concepts such as Services.

Sample domain model for a health insurance plan



Concepts: Conceptual Data Modeling

Introduction

Conceptual data modeling represents the initial stage in the development of the design of the persistent data and persistent data storage for the system. In many cases, the persistent data for the system are managed by a relational database management system (RDBMS). The business and system entities identified at a conceptual level from the business models and system requirements will be evolved through the use-case analysis, use-case design, and database design activities into detailed physical table designs that will be implemented in the RDBMS. Note that the Conceptual Data Model discussed in this concept document is not a separate artifact. Instead it consists of a composite view of information contained in existing Business Modeling, Requirements, and Analysis and Design Disciplines artifacts that is relevant to the development of the Data Model.

The Data Model typically evolves through the following three general stages:

Conceptual—This stage involves the identification of the high level key business and system entities and their relationships that define the scope of the problem to be addressed by the system. These key business and system entities are defined using the modeling elements of the UML profile for business modeling included in the Business Analysis Model and the Analysis Class model elements of the Analysis Model.

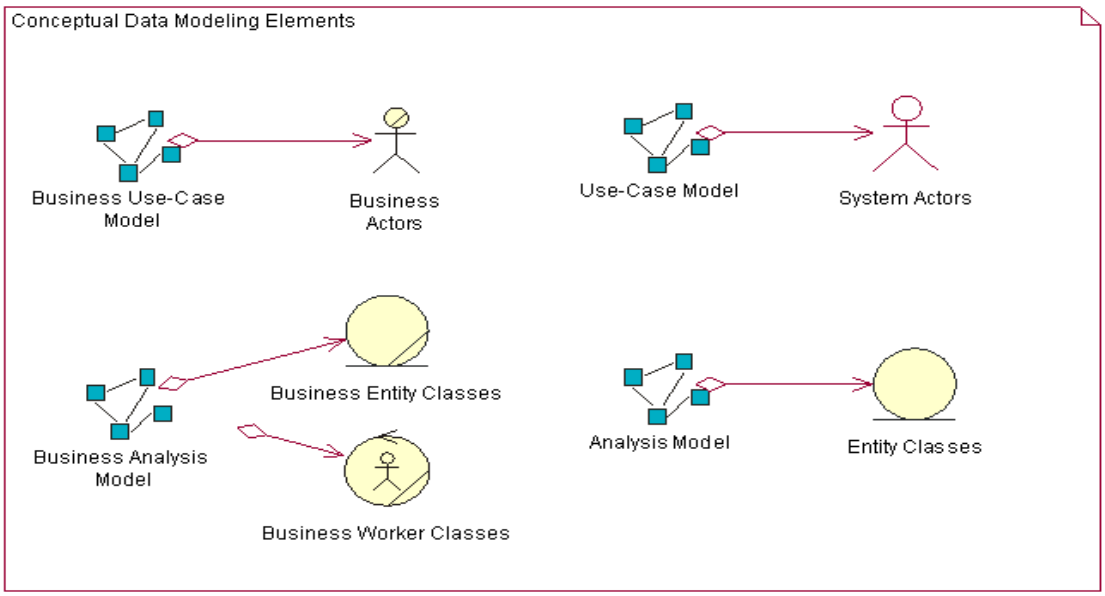
Logical—This stage involves the refinement of the conceptual high level business and system entities into more detailed logical entities. These logical entities and their relationships can be optionally defined in a Logical Data Model using the modeling elements of the UML profile for database design as described in Guidelines: Data Model. This optional Logical Data Model is part of the Artifact: Data Model and not a separate RUP artifact.

Physical—This stage involves the transformation of the logical class designs into detailed and optimized physical database table designs. The physical stage also includes the

mapping of the database table designs to tablespaces and to the database component in the database storage design.

The activities related to database design span the entire software development lifecycle, and the initial database design activities might start during the inception phase. For projects that use business modeling to describe the business context of the application, database design may start at a conceptual level with the identification of Business Actors and Business Use Cases in the Business Use-Case Model, and the Business Workers and Business Entities in the Business Analysis Model. For projects that do not use business modeling, the database design might start at the conceptual level with the identification of System Actors and System Use Cases in the Use-Case Model, and the identification of Analysis Classes in the Analysis Model from the Use-Case Realizations.

The figure below shows the set of Conceptual Data Model elements that reside in the Business Models, Requirements Models, and the Analysis Model.



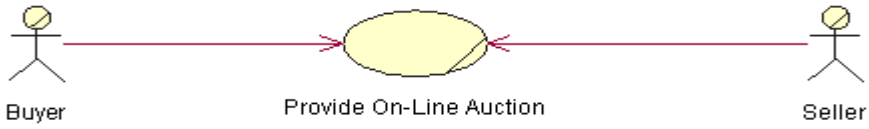
The following sections describe the elements of the Business Models, Use-Case Model, and Analysis Model that can be used to define the initial Conceptual Data Model for persistent data in the system.

Conceptual Data Modeling Elements

Business Models

Business Use-Case Model

The Business Use-Case Model consists of Business Actors and Business Use Cases. The Business Use Cases represent key business processes that are used to define the context for the system to be developed. Business Actors represent key external entities that interact with the business through the Business Use Cases. The figure below shows a very simple example Business Use-Case Model for an online auction application.

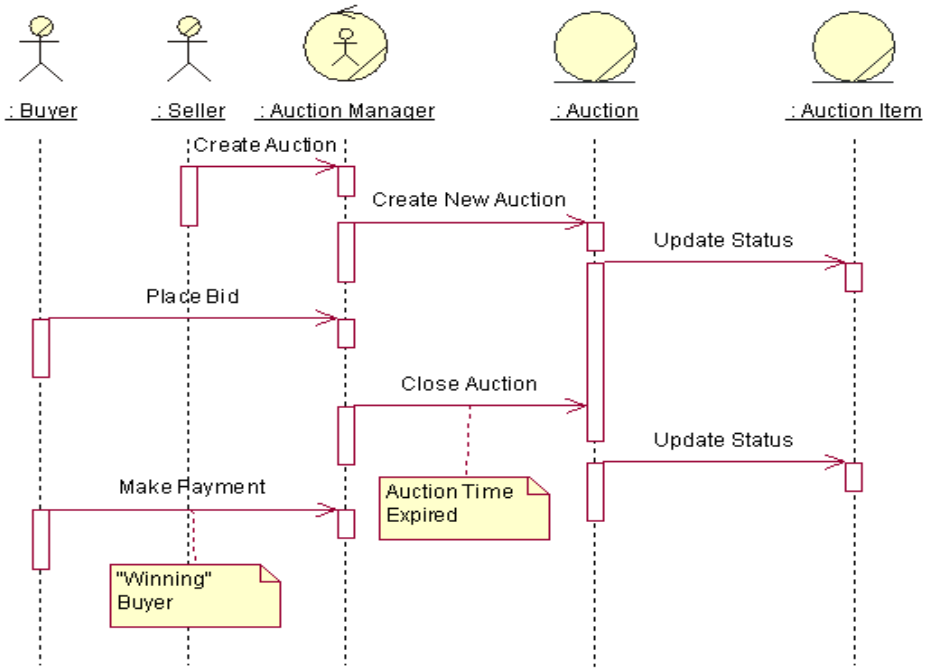


As entities of significance to the problem of space for the system, Business Actors are candidate entities for the Conceptual Data Model. In the example above, the Buyer and Seller Business Actors are candidate entities for which the online auction application must store information.

Business Analysis Model

The Business Analysis Model contains classes that model the Business Workers and Business Entities identified from analysis of the workflow in the Business Use Case. Business Workers represent the participating workers that perform the actions needed to carry out that workflow. Business Entities are "things" that the Business Workers use or produce during that workflow. In many cases, the Business Entities represent types of information that the system must store persistently.

The figure below shows an example sequence diagram that depicts Business Workers and Business Entities from one scenario of the Business Use Case titled "Provide Online Auction" for managing an auction.



In this simplified example, the Auction Manager object represents a Business Worker role that will likely be performed by the online auction management system itself. The Auction and Auction Item objects are Business Entities that are used or produced by the Auction Manager worker acting as an agent for the Seller and Buyer Business Actors. From a database design perspective, the Auction and Auction Item Business Entities are candidate entities for the Conceptual Data Model.

Requirements and Analysis Models

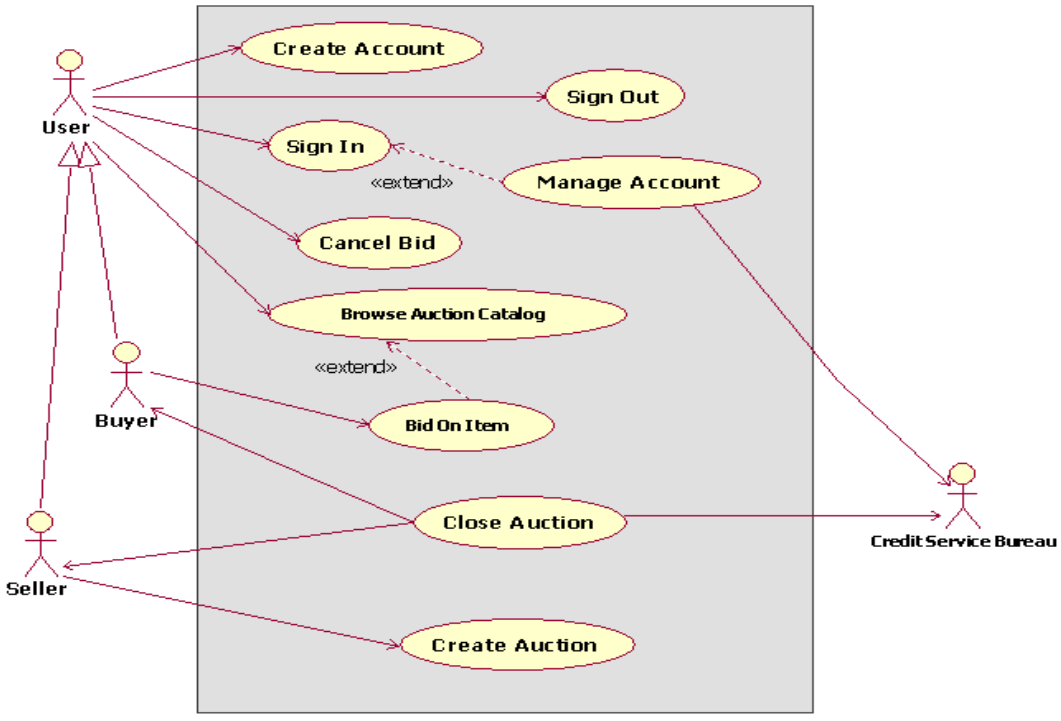
For projects that do not perform business modeling, the Requirements (System Use Case) and Analysis Models contain model elements that can be used to develop an initial Conceptual Data Model. For projects that use business modeling, the business entities and relationships identified in the Business Analysis Models are refined and detailed in the Analysis Model as Entity Classes.

System Use-Case Model

The System Use-Case Model contains System Actors and System Use Cases that define the primary interactions of the users with the system. The System Use Cases define the functional requirements for the system.

From a conceptual data modeling perspective, the System Actors represent entities external to the system for which the system might need to store persistent information. This is important in cases where the System Actor is an external system that provides data to and/or receives data from the system under development. System Actors can be derived from the Business Actors in the Business Use-Case Model and the Business Workers in the Business Analysis Model.

The figure below depicts the Business Use-Case Model for the online auction system. In this model, the Buyer and Seller Business Actors are now derived from a generic User Business Actor. A new System Actor named Credit Service Bureau has been added to reflect the need to process payments through an external entity. This new System Actor is another candidate entity for the Conceptual Data Model.



Analysis Model

The Analysis Model contains the Analysis Classes identified in the Use-Case Realizations for the System Use Cases. The types of Analysis Classes that are of primary interest from a conceptual

data modeling perspective are the Entity Analysis Classes. As defined in Guidelines: Analysis Class, Entity Analysis Classes represent information managed by the system that must be stored in a persistent manner. The Entity Analysis Classes and their relationships form the basis of the initial Data Model for the application.
The conceptual Entity Analysis Classes in the Analysis Model might be refined and detailed into logical Persistent Design Classes in the Design Model. These design classes represent candidate tables in the Data Model. The attributes of the classes are candidate columns for the tables and also represent candidate keys for them. See Guidelines: Forward-Engineering Relational Databases for a description of how elements in the Design Model can be mapped to Data Model elements.

Conceptual Class Category List
A *conceptual class* is a real-world concept or thing; a conceptual or essential perspective. At the noun filtering stage we are looking for conceptual classes. As we move through the design process we will start to design *software classes* that represent an implementation perspective of a software component but we will not get into language specific classes in 466. A conceptual class is *not* an *implementation* class, such as a class that can be implemented in an OO language such as Java or C++.

Conceptual Class Category	**Examples**
Physical or tangible objects	Register, Airplane
Specifications, deigns or descriptions of things	ProductSpecification FlightDescription
Places	Store Airport
Transactions	Sale Payment Reservation
Transaction line items	SalesLineItem
Roles of people	Cashier Pilot
Containers of other things	Store Bin Airplane
Things in a container	Item Passenger
Other computer or electro-mechanical systems external to the system	CreditPaymentAuthorizationSystem AirTrafficControl
Organizations	SalesDepartment ObjectAirline
Events	Sale Payment Meeting

	Flight Crash Landing
Rules and policies	RefundPolicy CancellationPolicy
Catalogs	ProductCatalog PartsCatalog
Records of finance, work, contracts, legal matters	Receipt Ledger EmploymentContract MaintenanceLog
Financial instruments and services	LineOfCredit Stock
Manuals, documents, reference papers, books	DailyPriceChangeList RepairManual

Types of Classes

During use case realization, we identify mainly four "types" of classes, boundary classes, data store classes and control classes.

The entity classes represent the information that the system uses. Examples of entity classes are: Customer, Product, and Supplier. Entity classes are essential to the system as the expected functionality of the system includes maintaining information about them or retrieving information from them.

The boundary classes represent the interaction between the system and its actors. A GUI form is an example of a boundary class.

Data store classes encapsulate the design decisions about data storage and retrieval strategies. This provides us flexibility to move a n application from database platform to another.

The control classes represent the control logic of the system. They implement the flow of events as given in a use case.

Entity Classes

Entity classes are the abstractions of the keys concepts of the system being modelled. If the steps of the Architectural Analysis have been carried out, many of the entity classes may have already been identified during those steps.

The Core functionality and logic of the system are encapsulated in the various entity classes. For example, if interest is to be calculated and paid to savings account holders, a savings Account entity class may be responsible for computing and returning the interest.

You can normally look for the following types of things as potential entity classes:

- Roles played by people or organizations about which information is required to be maintained by the system. For Example, Student in a Library Management System, Vendor in a Purchase Ordering System.
- Other physical, tangible things. For example, Book in a Library Management System.
- Events that requires remembrance. For example, Reservation and Issue in a Library Management System.

The logical data structures (attributes and relationships) of the entity classes would be designed to hold and manipulate the data according to the system's requirements. Values of the attributes and their relationships of the entity class objects are often given by actors. The entity classes are responsible for storing and managing information in the system.
Entity class objects are usually persistent, having attributes and relationships that need to be retained for a long time, sometimes even before the life of system. An entity class is usually not specific to one use cause realization. Objects of most entity classes would required in multiple use cases. Sometimes, an entity object may not be specific to the system itself.

Boundary Classes
Boundary classes represent the interaction between the system and its actors. They insulate the system from changes in the surroundings of the system, such as user interfaces, and interfaces to other systems.
There may be various types of boundary classes in a system:

- User Interfaces classes: Classes for encapsulating the human user interface of the system, such as GUI forms.
- System Interface Classes: Classes that encapsulate the interaction of the system with other systems.
- Device Interface Classes: Classes that provide the interface to devices that detect external events.

An important objective of identifying boundary classes is to ensure that the entity classes and the control classes are not affected by any changes to the boundary classes.
Actors interact with the system only through the boundary classes.

User Interface Classes
A user interface class represents the interaction between a use case and its initiating actor. This class has the responsibility of coordinating the interaction with the actor. A boundary class may have various subsidiary classes to which some of its responsibilities are delegated. For example, in a GUI application, there may be multiple forms within a use case.
During use case analysis, you should use the boundary classes as just place-holders for the GUI forms. Detailed GUI design is an activity of Class Design. During Analysis, the emphasis should be only on isolating all environment-dependent behaviour as boundary classes. These classes will get refined or replaced in the later stages.

System Interface Classes
A system interface class is responsible for interfacing with an external system. The interface offered by the external system would have been identified by the developers of that system. Thus, the behaviour of a system interface class should be derived directly from the interface specifications of the external system.
System interface classes achieve the purpose of isolating the internal details of the external systems, which may change over a period of time. Our system should not get affected by such changes in the internal details of the external systems.

Device Interface Classes

Device interface classes are responsible for interacting with the external devices that the system may depend upon for receiving inputs or handling outputs. Examples of such external devices would be: bar code reader, system clock, printer, etc.
A device may already have a well-defined interface, which could be used by you later during design. Therefore, a note of such interface should be made in the model documentation.

Data Store Classes
Data Store classes encapsulate our design decisions about the database structures which are used to store entity class objects, and to retrieve them later. For each entity class that required persistence, we create a corresponding data store class. A data store class typically receives an object of an entity class, and make it persistence (for example, by inserting a row in a table). At a later point of time, we may ask the data store class to return the entity class object.
Encapsulating the database design designs in data store classes, makes the entity classes independent of the database structure, and thus provides us greater flexibility to move an application from one database platform to another.

Controller Classes
Controller classes provide co-ordinating behaviour in the system. A typical example would be a controller class implementing the logic and flow of events of a use case.
Controller Classes isolates the entity classes and boundary classes from each other, making the system independent of the changes to the system boundary. They also isolate the use case specific behaviour from the entity class objects, thus making them re-usable across use cases and even across systems.
Simple use cases may be performed without using controller classes, with direct flow of data between boundary objects and entity objects. However, more complex use cases usually require and benefit from such controller classes. The characteristics of controller classes are:

- They define the order of events and transactions within a use case. In other words, they encapsulate the use case-specific behaviour of the system.
- They are relatively independent of the changes to the internal structure or behaviour of the entity classes.
- They are nearly independent of changes to the boundary classes.
- They may use or set several entity classes, thus coordinating the behaviour of these entity classes. However, this coordination can be invisible to the participating entity classes.

Though most of the times a control class correspond to a single use case, some times a single controller class may be use to control several use cases. Some tines there may even be multiple controller classes with in a single use case. As mentioned earlier, there may be use cases that do not require controller classes.

Association:

Association defines the relationship between two or more classes in the System. These generally relates to the one object having instance or reference of another object inside it. This article discusses on how we can implement Association in UML.

Associations in UML can be implemented using following ways:
1) Multiplicity
2) Aggregation
3) Composition

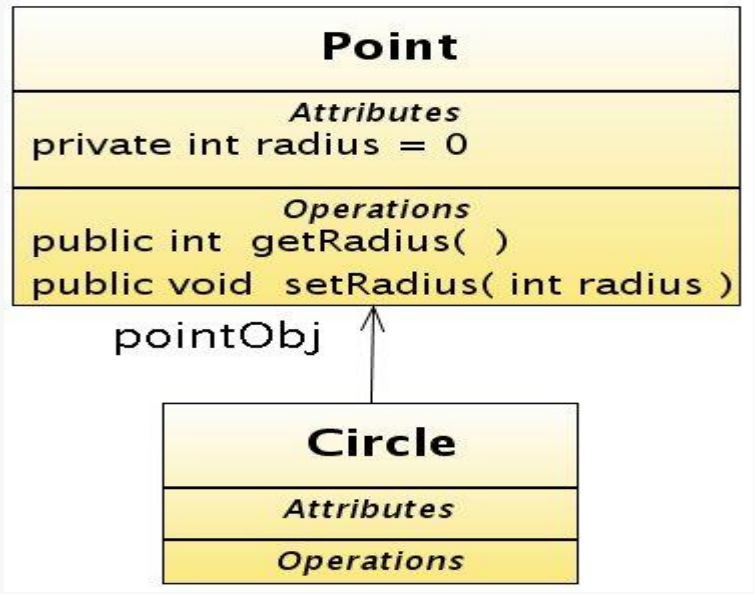
Multiplicity in UML:
Multiplicity indicates the no of instance of one class is linked to one instance of another class.
The various multiplicity values are listed below:

Notation	Description
1	Only One Instance
0..1	Zero or One Instance
*	Many Instance
0..*	Zero or Many Instance
1..*	One or Many Instance

Multiplicity Example:

```
public class Circle {
   private Point pointObj;

}

public class Point {
            private int X_POS = 0;
            private int Y_POS = 0;

      public int getXpos() {
         return this.X_POS;
      }

      public void setXpos(int xpos) {
         this.X_POS = xpos;
      }

      public int getYpos() {
         return this.Y_POS;
      }

      public void setYpos(int ypos) {
         this.Y_POS = ypos;
      }
}
```

The above code can be represented in UML as shown below:

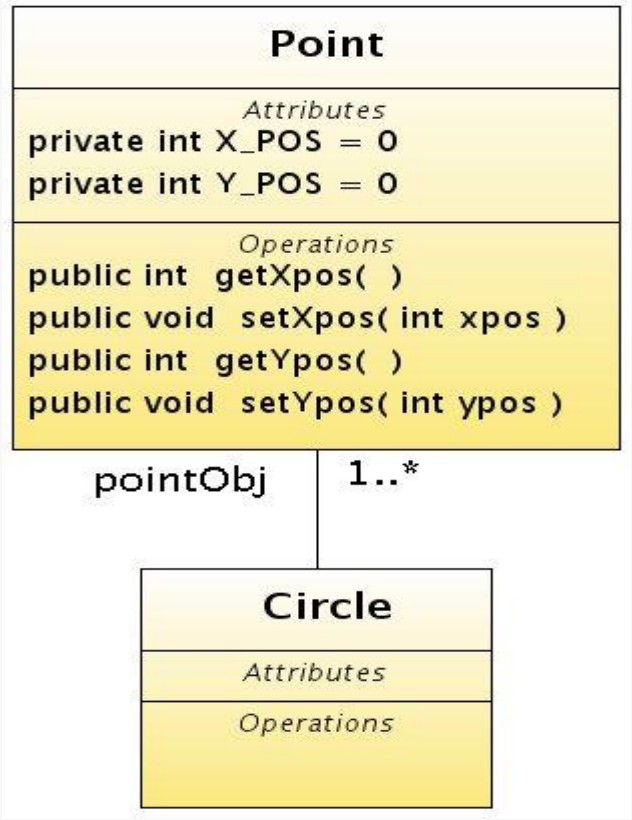


Association in UML

Now we will modify the above example to handle multiple objects as shown below:

```
public class Circle {
   private Point[] pointObj;

}

public class Point {
            private int X_POS = 0;
            private int Y_POS = 0;

      public int getXpos() {
         return this.X_POS;
      }

      public void setXpos(int xpos) {
         this.X_POS = xpos;
      }

      public int getYpos() {
         return this.Y_POS;
      }

      public void setYpos(int ypos) {
         this.Y_POS = ypos;
      }
}
```

The above code can be represented in UML as shown below:



Multiplicity Association in UML

As we can see that the Circle Class holds an Array of Point Object so we have added 1..* to the Point Object indicating that Circle Object holds More than one Point Object.

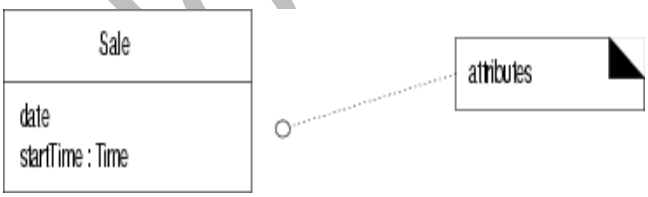
Attributes

attribute

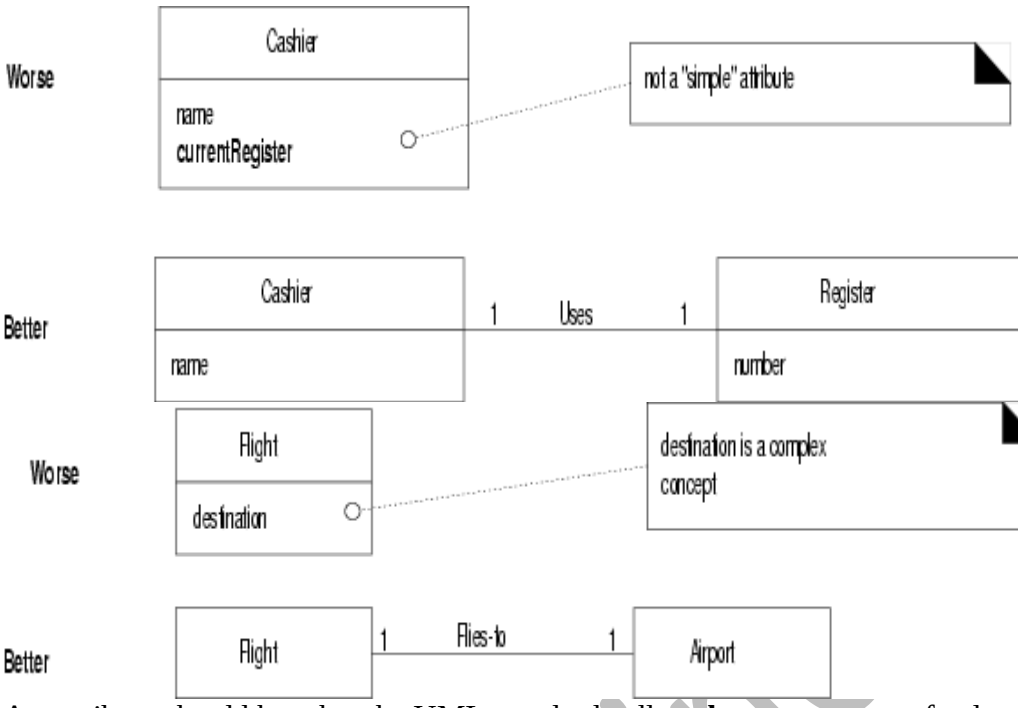
a logical data value of an object

In UML:

- Attributes are shown in the second compartment of the class box.
- The type of an attribute may optionally be shown.



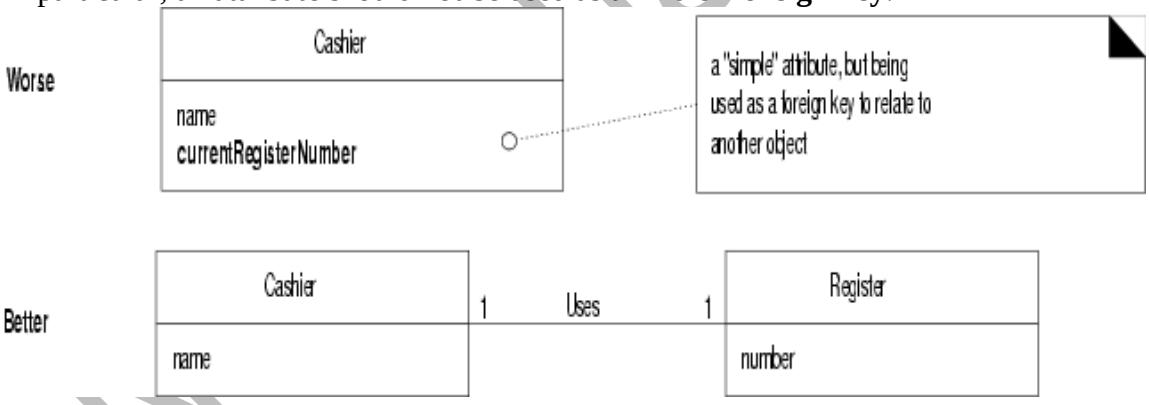
In a domain model, attributes and data types should be simple. Complex concepts should be represented by an association to another conceptual class.



An attribute should be what the UML standard calls a **data type**: a set of values for which unique identity is not meaningful. Numbers, strings, Booleans, dates, times, phone numbers, and addresses are examples of data types. Values of these types are called **value objects**.

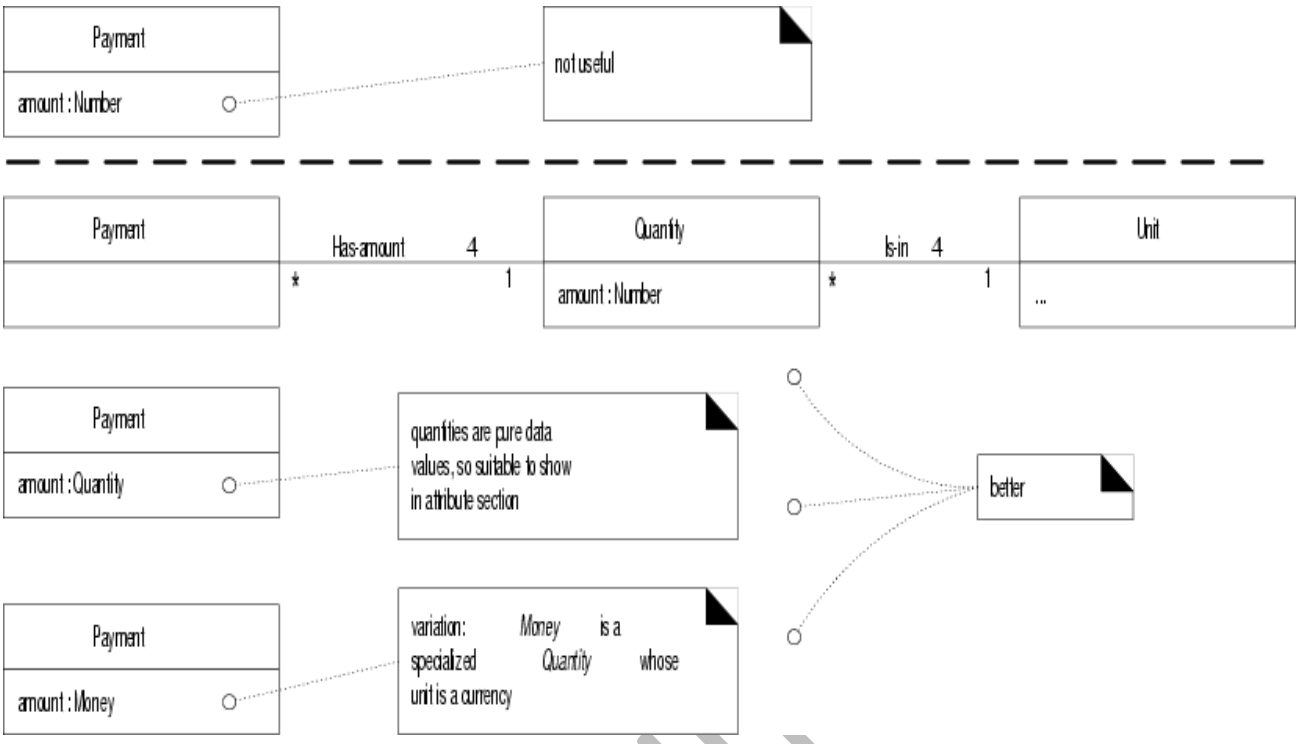
Relating Types

Conceptual classes in a domain model should be related by associations, not attributes. In particular, an attribute should not be used as a kind of **foreign key**.



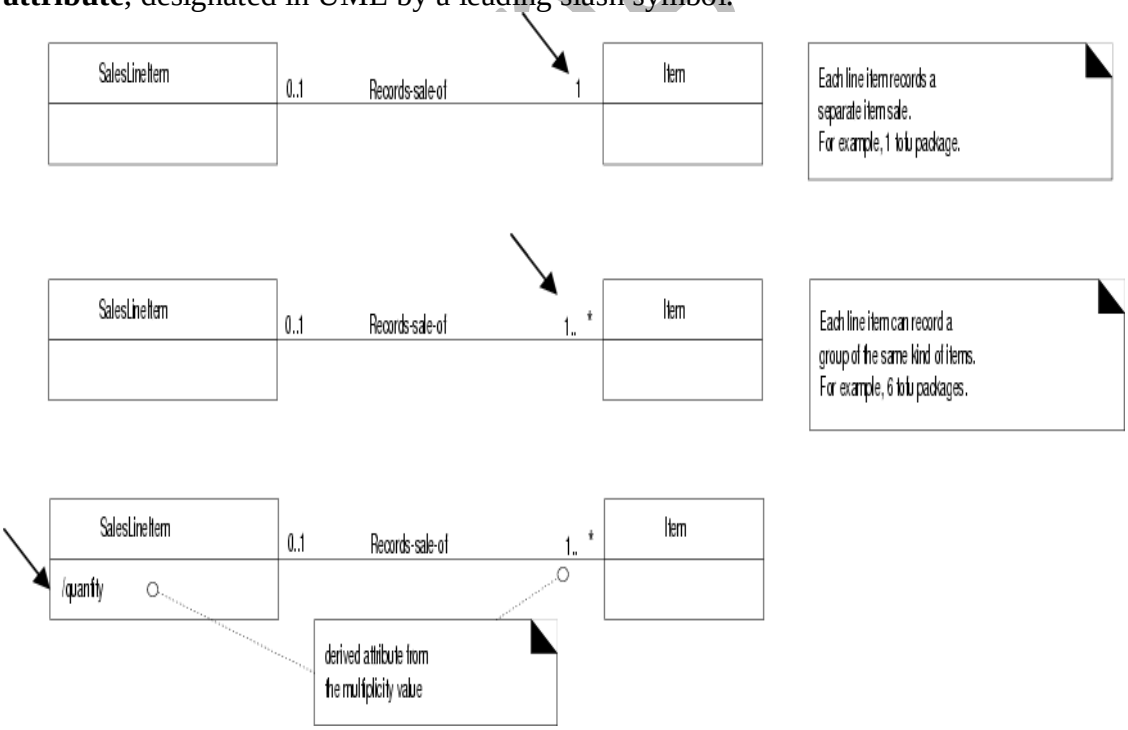
Quantities and Units

Quantities with associated units should be represented either as conceptual classes or as attributes of specialized types that imply units (e.g., *Money* or *Weight*).

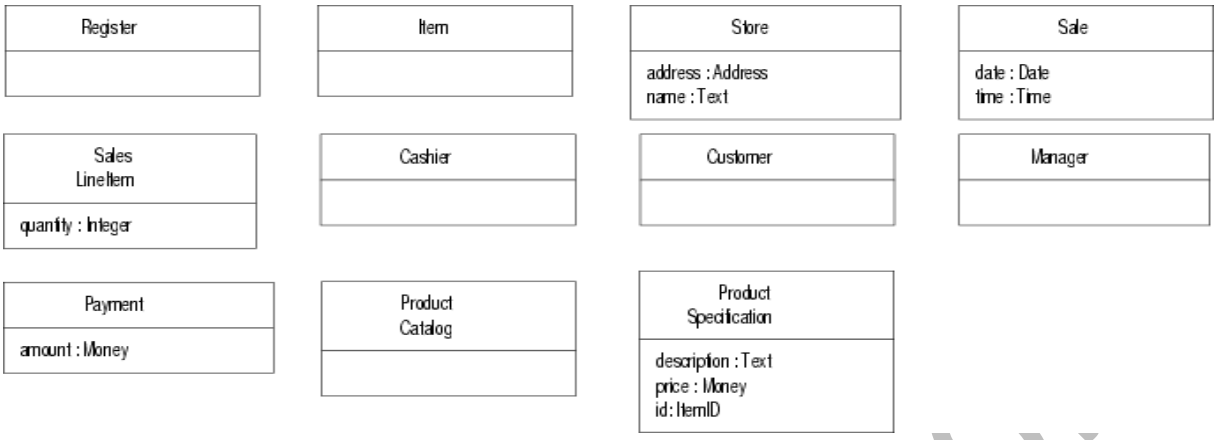


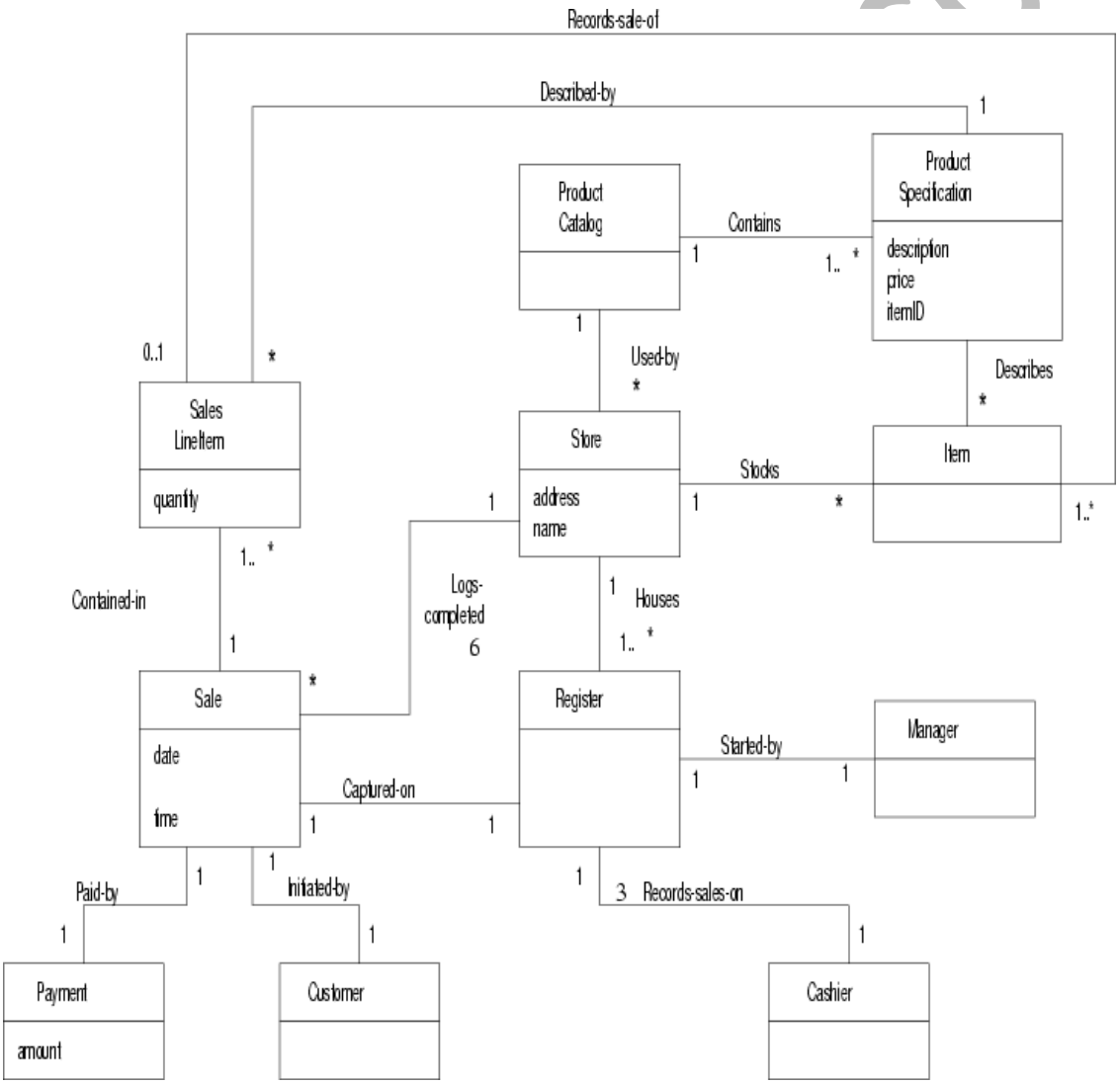
Derived Attributes

A quantity that can be calculated from other values, such as role multiplicities, is a **derived attribute**, designated in UML by a leading slash symbol.



NextGen POS Domain Model Attributes





Strategies to Identify Conceptual Classes
Two techniques are presented in the following sections:
1. Use a conceptual class category list.
2. Identify noun phrases.
Another excellent technique for domain modeling is the use of **analysis patterns,** which are existing partial domain models created by experts

Finding Conceptual Classes with Noun Phrase Identification
Another useful technique (because of its simplicity) suggested in [Abbot83] is linguistic analysis: identify the nouns and noun phrases in textual descriptions of a domain, and consider them as candidate conceptual classes or attributes.

Care must be applied with this method; a mechanical noun-to-class mapping isn't possible, and words in natural languages are ambiguous. Nevertheless, it is another source of inspiration. The fully dressed use cases are an excellent description to draw from for this analysis. For example, the current scenario of the *Process Sale* use case can be used.

Main Success Scenario (or Basic Flow):
1. Customer arrives at a **POS checkout** with **goods** and/or **services** to purchase.
2. Cashier starts a new **sale.**
3. Cashier enters **item identifier.**
4. System records **sale line item** and presents **item description, price,** and running **total.** Price calculated from a set of price rules. Cashier repeats steps 2-3 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 the completed **sale** and sends sale and payment information to the external **Accounting** (for accounting and **commissions)** and **Inventory** systems (to update inventory).
9. System presents **receipt.**
10.Customer leaves with receipt and goods (if any).
Extensions (or Alternative Flows):
7a. Paying by cash:
1. Cashier enters the cash **amount tendered.**
2. System presents the **balance due,** and releases the **cash drawer.**
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. System records the cash payment.
The domain model is a visualization of noteworthy domain concepts and vocabulary. Where are those terms found? In the use cases. Thus, they are a rich source to mine via noun phrase identification.

Some of these noun phrases are candidate conceptual classes, some may refer to conceptual classes that are ignored in this iteration (for example, "Accounting" and "commissions"), and some may be attributes of conceptual classes.

A weakness of this approach is the imprecision of natural language; different noun phrases may represent the same conceptual class or attribute, among other ambiguities. Nevertheless, it is recommended in combination with the *Conceptual Class Category List* technique.

Specification or Description Conceptual Classes

The following discussion may at first seem related to a rare, highly specialized issue. However, it turns out that the need for specification conceptual classes (as will be defined) is common in any domain models. Thus, it is emphasized.

Note that in earlier times a *register* was just one possible implementation of how to record sales. The term has acquired a generalized meaning over time.

Assume the following:
• An *Item* instance represents a physical item in a store; as such, it may even have a serial number.
• An *Item* has a description, price, and itemID, which are not recorded anywhere else.
• Everyone working in the store has amnesia.
• Every time a real physical item is sold, a corresponding software instance of *Item* is deleted from "software land."
With these assumptions, what happens in the following scenario?
There is strong demand for the popular new vegetarian burger—ObjectBurger. The store sells out, implying that all *Item* instances of ObjectBurgers are deleted from computer memory. Now, here is the heart of the problem: If someone asks, "How much do Object Burgers cost?", no one can answer, because the memory of their price was attached to inventoried instances, which were deleted as they were sold.

Notice also that the current model, if implemented in software *as* described, has duplicate data and is space-inefficient because the description, price, and itemID are duplicated for every *Item* instance of the same product.

The Need for Specification or Description Conceptual Classes

The preceding problem illustrates the need for a concept of objects that are specifications or descriptions of other things. To solve the *Item* problem, what *is* needed is a P*roductSpecification* (or *ItemSpecification, ProductDescription, ...*) conceptual class that records information about items. A *ProductSpecification* does not represent an *Item*, it represents a description of information *about* items. Note that even if all inventoried items are sold and their corresponding *Item* software instances are deleted, the *ProductSpecifications* still remain.

Description or specification objects are strongly related to the things they describe. In a domain model, it is common to state that an *XSpecification Describes an X*.

The need for specification conceptual classes is common in sales and product domains. It is also common in manufacturing, where a *description* of a manufactured thing is required that is distinct from the thing itself. Time and space have been taken in motivating specification conceptual classes because they are very common; it is not a rare modeling concept.

When Are Specification Conceptual Classes Required?
The following guideline suggests when to use specifications:

Add a specification or description conceptual class (for example, *ProductSpecification)* 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 (for example, *Item)* results in a loss of information that needs to be maintained, due to the incorrect association of information with the deleted thing.
3. It reduces redundant or duplicated information.

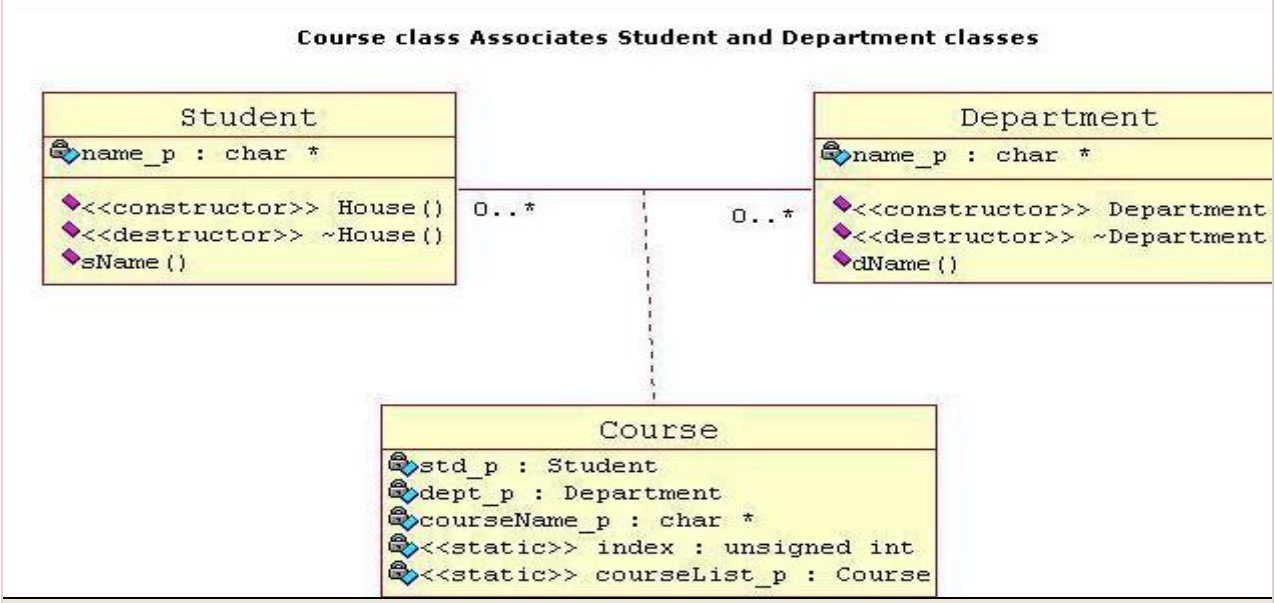
Association, Aggregation and Composition Relationships with Examples

Association is a simple structural connection or channel between classes and is a relationship where all objects have their own lifecycle and there is no owner.

Lets take an example of Department and Student.

Multiple students can associate with a single Department and single student can associate with multiple Departments, but there is no ownership between the objects and both have their own lifecycle. Both can create and delete independently.

Here is respective Model and Code for the above example.



Aggregation is a specialize form of Association where all object have their own lifecycle but there is a ownership like parent and child. Child object can not belong to another parent object at the same time. We can think of it as "has-a" relationship.

Implementation details:

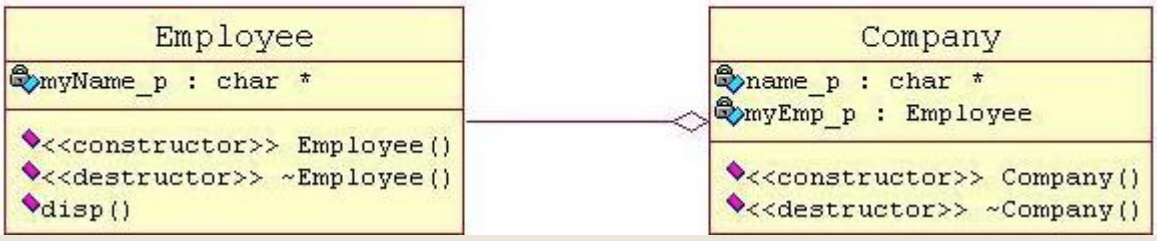
1. Typically we use pointer variables that point to an object that lives outside the scope of the aggregate class
2. Can use reference values that point to an object that lives outside the scope of the aggregate class
3. Not responsible for creating/destroying subclasses

Lets take an example of Employee and Company.

A single Employee can not belong to multiple Companies (legally!!), but if we delete the Company, Employee object will not destroy.

Here is respective Model and Code for the above example.

Employee class has Agregation Relatioship with Company class



Composition is again specialize form of Aggregation. It is a strong type of Aggregation. Here the Parent and Child objects have coincident lifetimes. Child object dose not have it's own lifecycle and if parent object gets deleted, then all of it's child objects will also be deleted.

Implentation details:

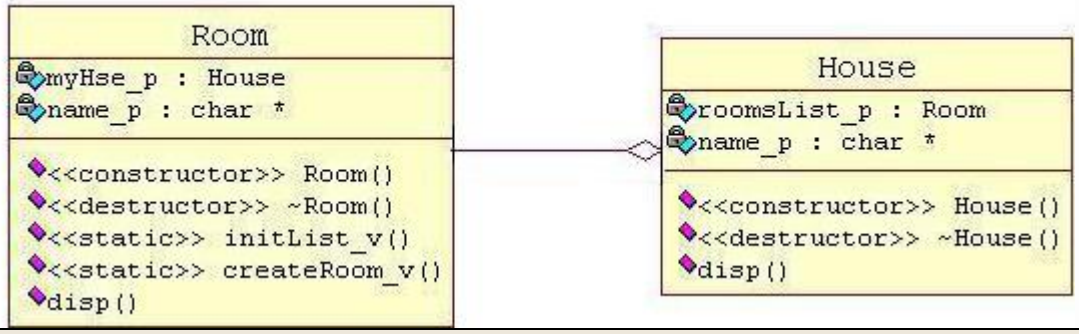
1. Typically we use normal member variables
2. Can use pointer values if the composition class automatically handles allocation/deallocation
3. Responsible for creation/destruction of subclasses

Lets take an example of a relationship between House and it's Rooms.

House can contain multiple rooms there is no independent life for room and any room can not belong to two different house. If we delete the house room will also be automatically deleted.

Here is respective Model and Code for the above example.

Room class has Composition Relationship with House class



Association:

1. Create a folder called "Links"
2. Create a shortcut/link inside this folder and link it to www.go4expert.com
3. Create another shortcut/link instide this folder and link it to www.google.com
4. Ask your friend to do the same on another machine using same links

(www.go4expert.com and www.google.com)

5. Delete the "Links" folder, and open your browser to check if www.go4expert.com and www.google.com still exist or not

Briefly, Association is a relationship where all the objects have different lifecycles. there is no owner.

Aggregation:

1. Create a file called file.txt
2. Make a simple Application to open the File.txt (rw mode), but don't program it close the connection.
3. Run an instance of this application (it should work ok and can open the file for rw)
4. Keep the first instance, and run another instance of this application (In theory it should complain that it can't open the file in rw mode because it is already used by other application).
5. Close the 2 instances (make sure you close the connection).

From the above application, we understand that the Application and the File has a separate lifecycles, however this file can be opened only by one application simuletanously (there is only one parent at the same time, however, this parent can move the child to another parent or can make it orphan).

Composition:

1. Open a new Document name it as test.txt
2. Write this sentence inside this document "This is a composition".
3. Save the document.
4. Now, delete this document.

This is what is called composition, you can't move the sentence "This is a omposition" from the document because its lifecycle is linked to the parent (i.e. the document here !!)

UML Activity Diagram

Overview:

Activity diagram is another important diagram in UML to describe dynamic aspects of the system.

Activity diagram is basically a flow chart to represent the flow form one activity to another activity. The activity can be described as an operation of the system.

So the control flow is drawn from one operation to another. This flow can be sequential, branched or concurrent. Activity diagrams deals with all type of flow control by using different elements like fork, join etc.

Purpose:

The basic purposes of activity diagrams are similar to other four diagrams. It captures the dynamic behaviour of the system. Other four diagrams are used to show the message flow from one object to another but activity diagram is used to show message flow from one activity to another.

Activity is a particular operation of the system. Activity diagrams are not only used for visualizing dynamic nature of a system but they are also used to construct the executable system by using forward and reverse engineering techniques. The only missing thing in activity diagram is the message part.

It does not show any message flow from one activity to another. Activity diagram is some time considered as the flow chart. Although the diagrams looks like a flow chart but it is not. It shows different flow like parallel, branched, concurrent and single.
So the purposes can be described as:

- Draw the activity flow of a system.
- Describe the sequence from one activity to another.
- Describe the parallel, branched and concurrent flow of the system.

How to draw Component Diagram?

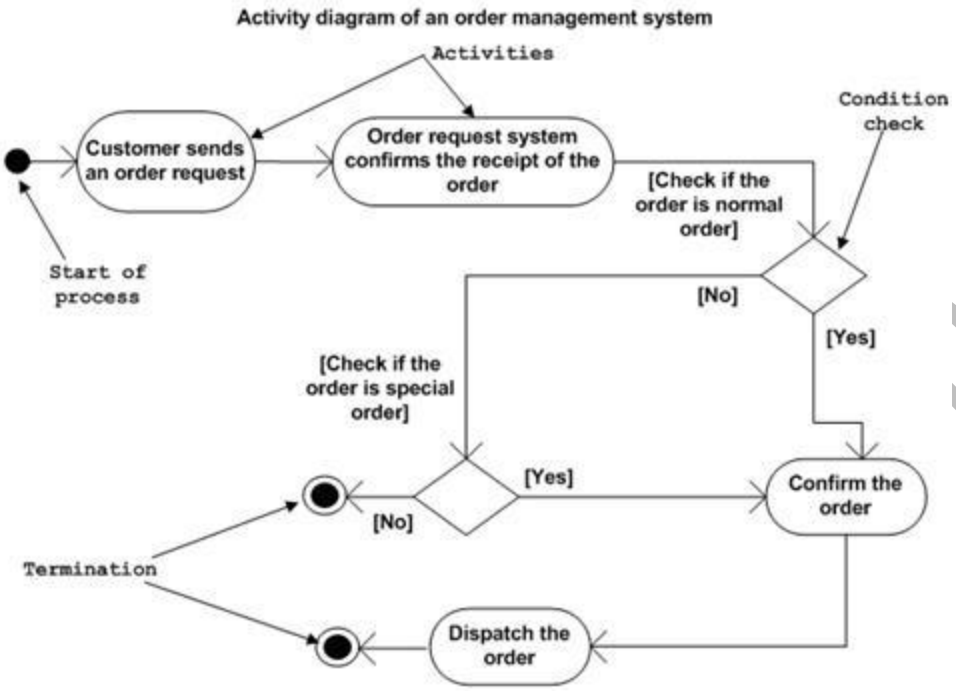
Activity diagrams are mainly used as a flow chart consists of activities performed by the system. But activity diagram are not exactly a flow chart as they have some additional capabilities. These additional capabilities include branching, parallel flow, swimlane etc.
Before drawing an activity diagram we must have a clear understanding about the elements used in activity diagram. The main element of an activity diagram is the activity itself. An activity is a function performed by the system. After identifying the activities we need to understand how they are associated with constraints and conditions.
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 Interaction 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.

UNIT 3 - Sequence Diagram

UML sequence diagrams are used to represent or model the flow of messages, events and actions between the objects or components of a system. Time is represented in the vertical direction showing the sequence of interactions of the header elements, which are displayed horizontally at the top of the diagram.

Sequence Diagrams are used primarily to design, document and validate the architecture, interfaces and logic of the system by describing the sequence of actions that need to be performed to complete a task or scenario. UML sequence diagrams are useful design tools because they provide a dynamic view of the system behavior which can be difficult to extract from static diagrams or specifications.

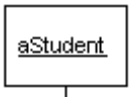
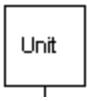
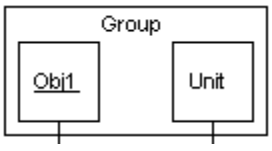
Although UML sequence diagrams are typically used to describe object-oriented software systems, they are also extremely useful as system engineering tools to design system architectures, in business process engineering as process flow diagrams, as message sequence charts and call flows for telecom/wireless system design, and for protocol stack design and analysis.

Sequence Diagram Drawing Elements

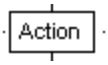
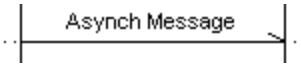
This tutorial describes the basic drawing elements used in sequence diagrams and when they are used. These are the diagram elements that are supported by the **Sequence Diagram Editor** tool. Some are not part of the UML specification and may not be supported by other UML tools.

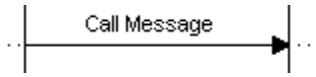
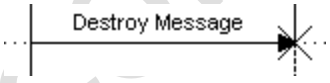
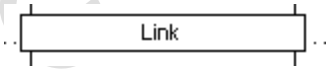
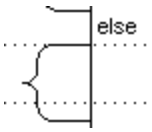
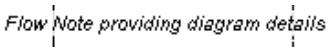
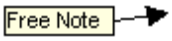
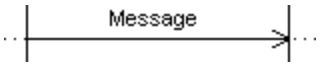
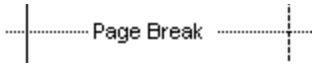
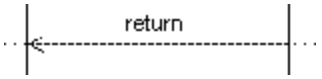
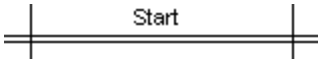
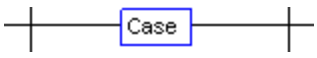
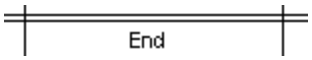
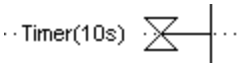
Sequence Diagram Header Elements

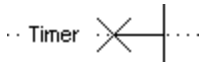
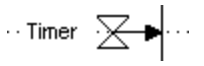
The header portion of the sequence diagram represents the components or objects of the system being modeled and are laid out horizontally at the top of the diagram. See an example sequence diagram here.

Actor	Represents an external person or entity that interacts with the system	
Object	Represents an object in the system or one of its components	
Unit	Represents a subsystem, component, unit, or other logical entity in the system (may or may not be implemented by objects)	
Separator	Represents an interface or boundary between subsystems, components or units (e.g., air interface, Internet, network)	
Group	Groups related header elements into subsystems or components	

Sequence Diagram Body Elements

Action	Represents an action taken by an actor, object or unit	
Asynchronous Message	An asynchronous message between header elements	

Block	A block representing a loop or conditional for a particular header element 
Call Message	A call (procedure) message between header elements 
Create Message	A "create" message that creates a header element (represented by lifeline going from dashed to solid pattern) 
Destroy Element	Represents the destruction of a header element 
Destroy Message	Represents the destruction of a header element as a result of a call from another element 
Diagram Link	Represents a portion of a diagram being treated as a functional block. Similar to a procedure or function call that abstracts functionality or details not shown at this level. Can optionally be linked to another diagram for elaboration. 
Else Block	Represents an "else" block portion of a diagram block 
Flow Note	Documentation note that is automatically formatted to flow after previous elements 
Free Note	Documentation note that is free-flowing and can be placed anywhere in the diagram (can also be anchored relative to a flow element) 
Message	A simple message between header elements 
Page Break	A page break in the diagram 
Return Message	A return message between header elements 
Scenario Start	Start of a scenario (set of alternatives) 
Scenario Case	Start of an alternative or case in a scenario 
Scenario End	End of a scenario 
State	A state change for a header element 
Steady State	A steady state in the system 
Timer Start	Start of a timer for a particular header element 

Timer Stop	Stop of a timer for a particular header element	
Timer Expiration	Expiration of a timer for a particular header element	

What can be modeled using sequence diagrams?

Sequence diagrams are particularly useful for modeling:

- **Complex interactions between components.** Sequence diagrams are often used to design the interactions between components of a system that need to work together to accomplish a task. They are particularly useful when the components are being developed in parallel by different teams (typical in wireless and telephony systems) because they support the design of robust interfaces that cover multiple scenarios and special cases.
- **Use case elaboration.** Usage scenarios describe a way the system may be used by its actors. The UML sequence diagram can be used to flesh out the details of one or more use cases by illustrating visually how the system will behave in a particular scenario. The use cases along with their corresponding sequence diagrams describe the expected behavior of the system and form a strong foundation for the development of system architectures with robust interfaces.
- **Distributed & web-based systems.** When a system consists of distributed components (such as a client communicating with one or more servers over the Internet), sequence diagrams can be used to document and validate the architecture, interfaces and logic of each of these components for a set of usage scenarios.
- **Complex logic.** UML sequence diagrams are often used to model the logic of a complex feature by showing the interactions between the various objects that collaborate to implement each scenario. Modeling multiple scenarios showing different aspects of the feature helps developers take into account special cases during implementation.
- **State machines.** Telecom, wireless and embedded systems make extensive use of state machine based designs where one or more state machines communicate with each other and with external entities to perform their work. For example, each task in the protocol stack of a cellular phone goes through a series of states to perform actions such as setup a call or register with a new base station. Similarly the call processing components of a Mobile Switching Center use state machines to control the registration and transfer of calls to roaming subscribers. Sequence diagrams (or call flows as they are commonly referred to in the telecom and wireless industry) are useful for these types of applications because they can visually depict the messages being exchanged between the components and their associated state transitions.

Benefits of using UML sequence diagrams

These are some of the main benefits of using UML sequence diagrams.

1. Help you discover architectural, interface and logic problems early. Because they allow you to flesh out details before having to implement anything, sequence diagrams are useful tools to find architectural, interface and logic problems early on in the design process. You can validate your architecture, interfaces, state machine and logic by seeing how the system architecture would handle different basic scenarios and special cases.

This is particularly true for systems involving the interaction of components that are being implemented in parallel by different teams. In the cell phone example, each task would typically be implemented by a separate team. Having a set of sequence diagrams describing how the

interfaces are actually used and what messages/actions are expected at different times gives each team a consistent and robust implementation plan. You can also document how special cases should be handled across the entire system.

The very act of creating the sequence diagrams and making them work with your architecture is valuable because it forces you to think about details such as interfaces, states, message order, assignment of responsibilities, timers/timeouts and special/error cases ahead of time.

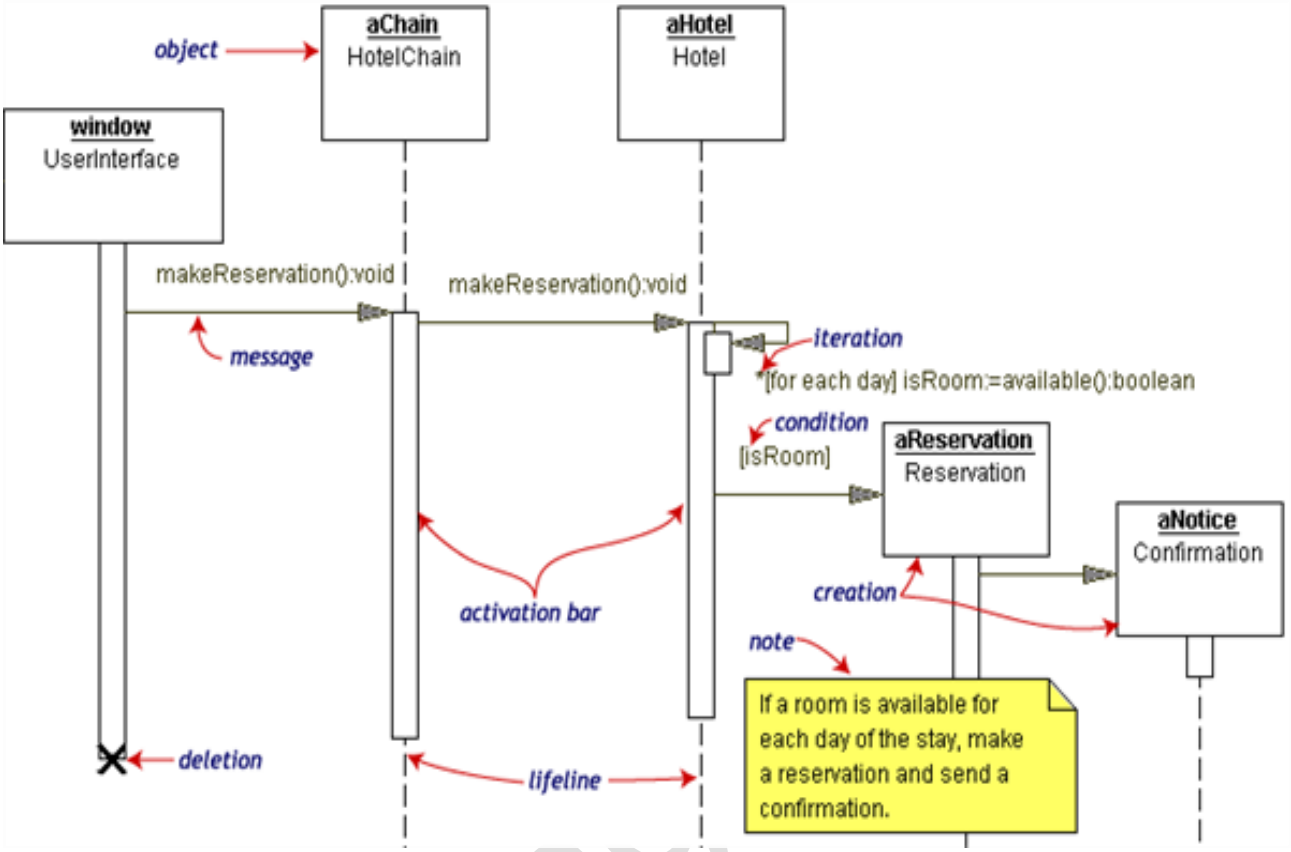
2. Collaboration tool. Sequence diagrams are valuable collaboration tools during design meetings because they allow you to discuss the design in concrete terms. You can see the interactions between entities, various proposed state transitions and alternate courses/special cases on paper as you discuss the design.

In our experience, having a concrete design proposal during design meetings greatly enhances the productivity of these meetings even if the proposed design has problems. You can narrow down the problems and then make corrections to solve them. The proposal serves as a concrete starting point for the discussion and as a place to capture proposed changes.

Sequence diagram editor makes it so easy to edit your sequence diagrams that you could even make the corrections in real time during the meeting and instantly see the result of the changes as you make them.

3. Documentation. Sequence diagrams can be used to document the dynamic view of the system design at various levels of abstraction, which is often difficult to extract from static diagrams or even the complete source code. The diagrams can abstract much of the implementation detail and provide a high level view of system behavior.

Below is a sequence diagram for making a hotel reservation. The object initiating the sequence of messages is **a Reservation window**.

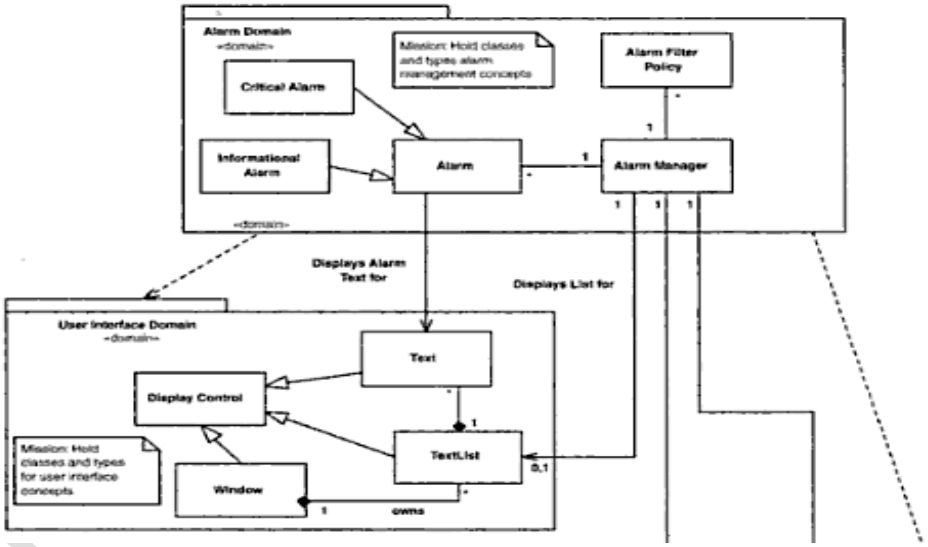


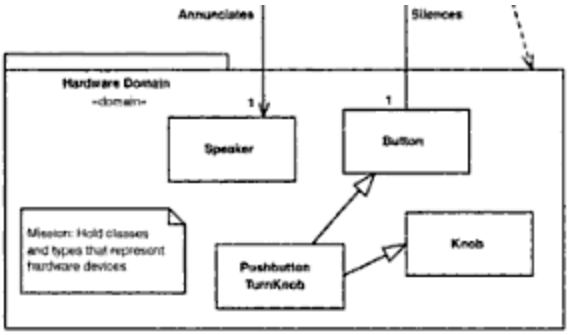
The **Reservation window** sends a makeReservation() message to a **HotelChain**. The **HotelChain** then sends a makeReservation() message to a **Hotel**. If the **Hotel** has available rooms, then it makes a **Reservation** and a **Confirmation**.
Each vertical dotted line is a **lifeline**, representing the time that an object exists. Each arrow is a message call. An arrow goes from the sender to the top of the **activation bar** of the message on the receiver's lifeline. The activation bar represents the duration of execution of the message.
In our diagram, the **Hotel** issues a **self call** to determine if a room is available. If so, then the **Hotel** creates a **Reservation** and a **Confirmation**. The asterisk on the self call means **iteration** (to make sure there is available room for each day of the stay in the hotel). The expression in square brackets, [ ], is a **condition**.

The diagram has a clarifying **note**, which is text inside a dog-eared rectangle. Notes can be put into any kind of UML diagram.

There are many possible logical architectures—that is, ways to organize your design model. The ROPES process recommends a logical architecture based on the concept of *domains*. A domain is an independent subject area that generally has its own vocabulary. The use of the term *domain* in this way is similar to its use in [1]. Domains provide a means by which your model can be organized—partitioned into its various subjects, such as user interface, hardware, alarm management, communications, operating system, data management, medical diagnostics, guidance and navigation, avionics, image reconstruction, task planning, and so on.

Used in this way, a domain is just a UML package. UML packages contain model elements, but other than providing a namespace, packages have no semantics and are not instantiable.[2] The UML does not provide a criterion for what should go in one package versus another, but domains do. For this reason, we represent domain as a «domain» stereotype package that includes a mission, specifically "hold classes and types around the common subject matter." The use of domains does not dictate how objects will be organized and deployed at run-time but what the physical architecture is all about.





Logical Architecture

A package diagram is a UML diagram composed only of packages and the dependencies between them. A package is a UML construct that enables you to organize model elements, such as use cases or classes, into groups. Packages are depicted as file folders and can be applied on any UML diagram. Create a package diagram to:

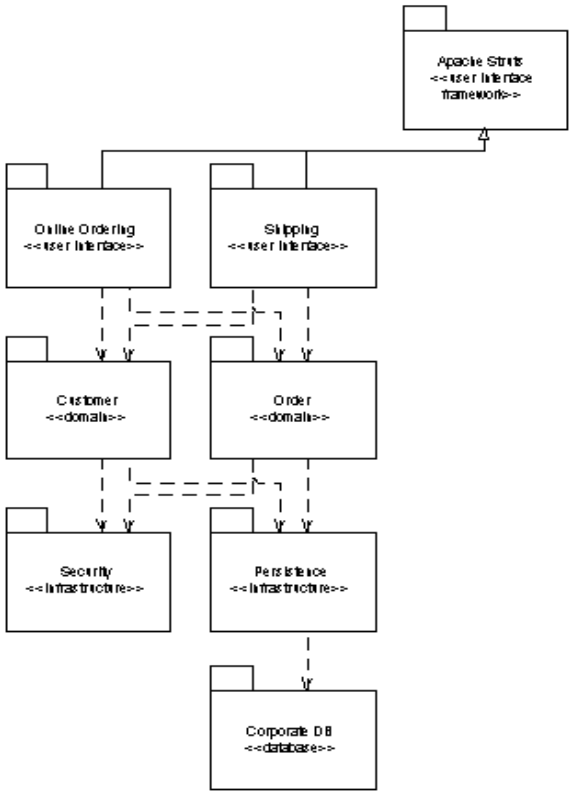
- Depict a high-level overview of your requirements (overviewing a collection of UML Use Case diagrams)
- Depict a high-level overview of your architecture/design (overviewing a collection of UML Class diagrams).
- To logically modularize a complex diagram.
- To organize Java source code.

There are guidelines for:

1. Class Package Diagrams
2. Use Case Package Diagrams
3. Packages

1. Class Package Diagrams

A class package diagram.



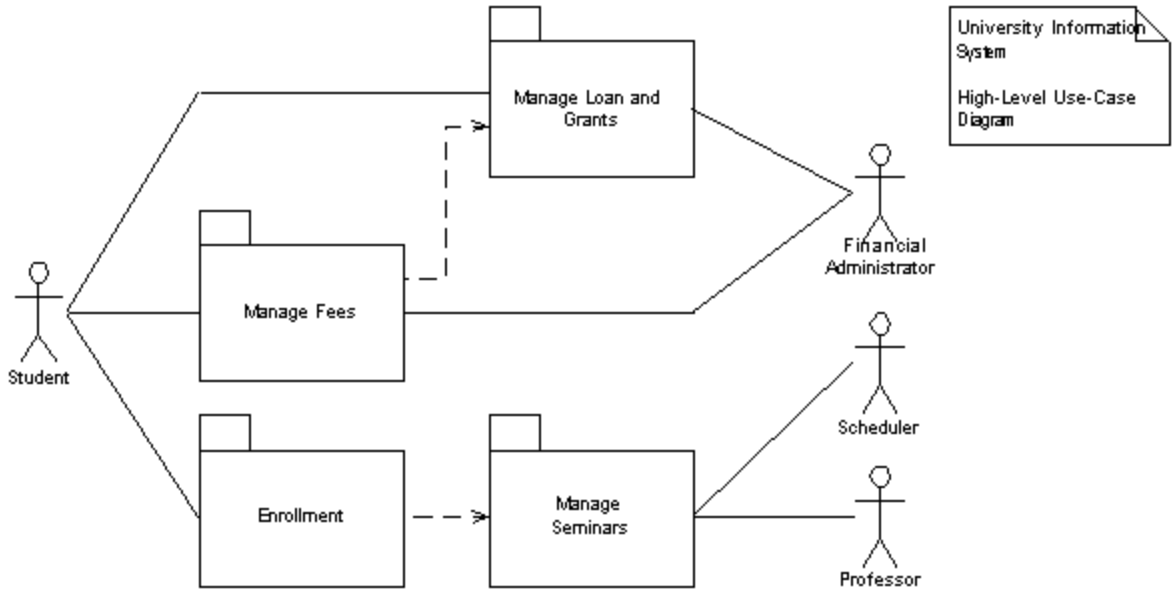
1. Create UML Component Diagrams to Physically Organize Your Design.
2. Place Subpackages Below Parent Packages.
3. Vertically Layer Class Package Diagrams.
4. Create Class Package Diagrams to Logically Organize Your Design. **Error! Reference source not found.**depicts a UML Class diagram organized into packages. In addition to the package guidelines presented below, apply the following heuristics to organize UML Class diagrams into package diagrams:

- Place the classes of a framework in the same package.
- Classes in the same inheritance hierarchy typically belong in the same package.
- Classes related to one another via aggregation or composition often belong in the same package.
- Classes that collaborate with each other a lot, information that is reflected by your UML Sequence diagrams and UML Collaboration diagrams, often belong in the same package.

2. Use Case Package Diagrams

Use cases are often a primary requirement artifact in object-oriented development methodologies, this is particularly true of instantiations of the Unified Process, and for larger projects package diagrams are often created to organize these usage requirements.

A UML Use Case diagram comprised mostly of packages.



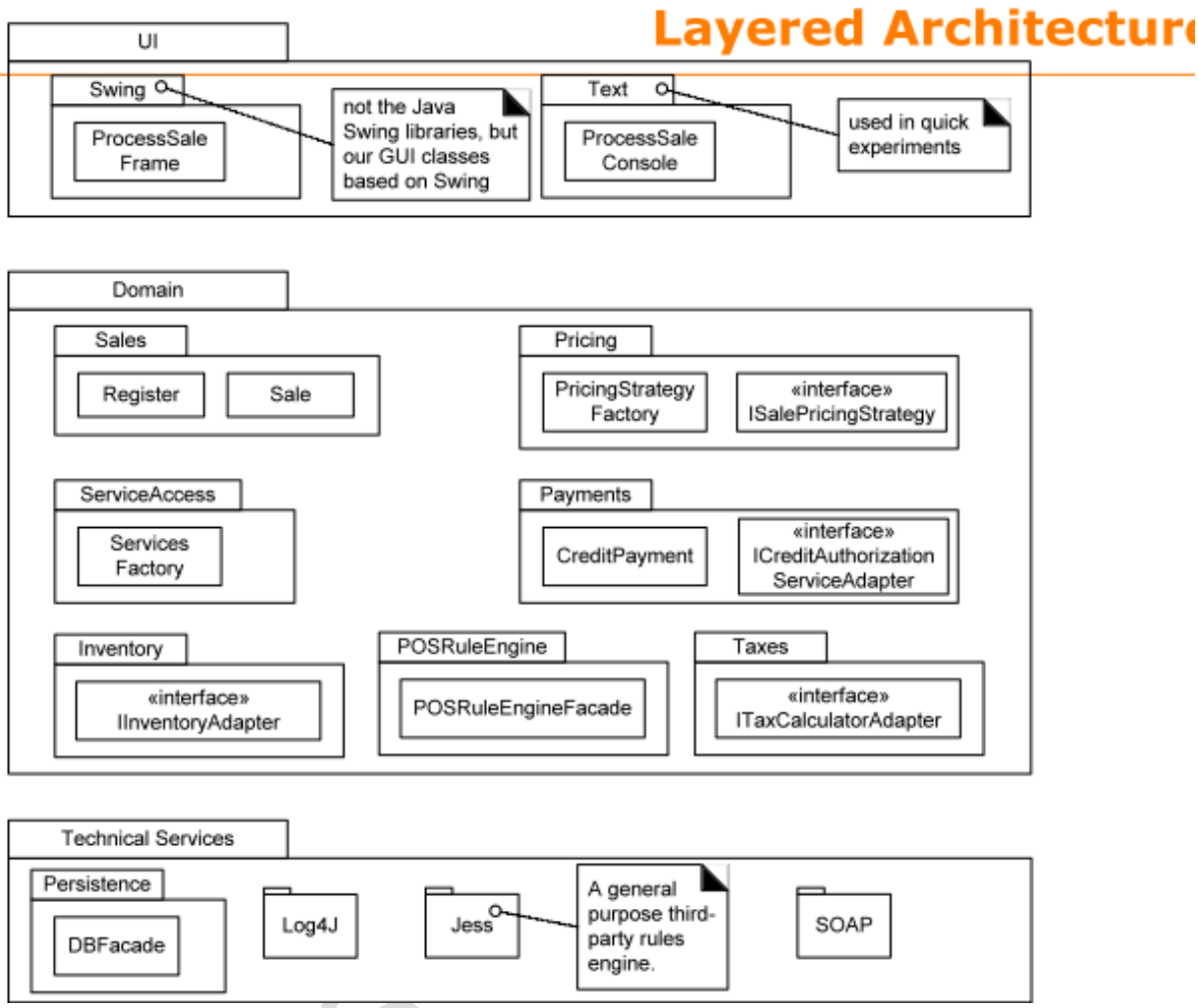
1. Create Use Case Package Diagrams to Organize Your Requirements
2. Include Actors on Use Case Package Diagrams
3. Horizontally Arrange Use Case Package Diagrams

3. Packages

The advice presented in this section is applicable to the application of packages on any UML diagram, not just package diagrams.

1. Give Packages Simple, Descriptive Names
2. Apply Packages to Simplify Diagrams
3. Packages Should be Cohesive
4. Indicate Architectural Layers With Stereotypes on Packages
5. Avoid Cyclic Dependencies Between Packages
6. Package Dependencies Should Reflect Internal Relationships

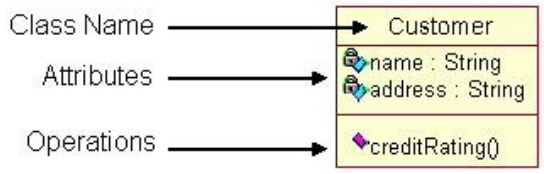
Logical Architecture Refinement:



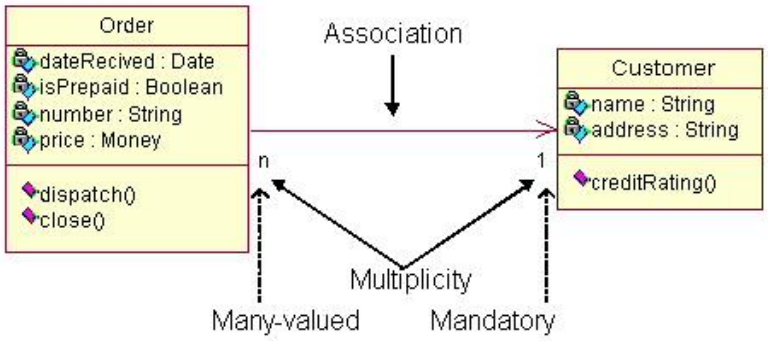
UML Class diagrams:

Class diagrams are widely used to describe the types of objects in a system and their relationships. Class diagrams model class structure and contents using design elements such as classes, packages and objects. Class diagrams describe three different perspectives when designing a system, conceptual, specification, and implementation.[1] These perspectives become evident as the diagram is created and help solidify the design. This example is only meant as an introduction to the UML and class diagrams. If you would like to learn more see the Resources page for more detailed resources on UML.

Classes are composed of three things: a name, attributes, and operations. Below is an example of a class.

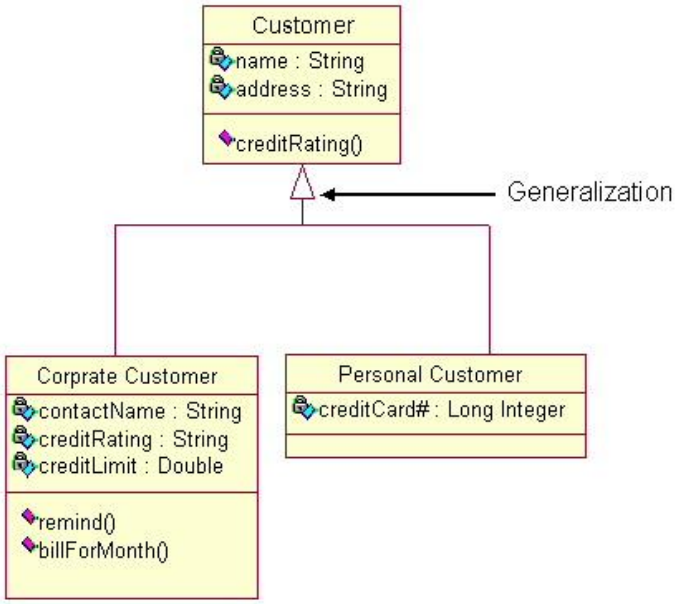


Class diagrams also display relationships such as containment, inheritance, associations and others.[2] Below is an example of an associative relationship:



The association relationship is the most common relationship in a class diagram. The association shows the relationship between instances of classes. For example, the class Order is associated with the class Customer. The multiplicity of the association denotes the number of objects that can participate in then relationship.[1] For example, an Order object can be associated to only one customer, but a customer can be associated to many orders.

Another common relationship in class diagrams is a generalization. A generalization is used when two classes are similar, but have some differences. Look at the generalization below:



In this example the classes Corporate Customer and Personal Customer have some similarities such as name and address, but each class has some of its own attributes and operations. The

class Customer is a general form of both the Corporate Customer and Personal Customer classes.[1] This allows the designers to just use the Customer class for modules and do not require in-depth representation of each type of customer.

When to Use: Class Diagrams

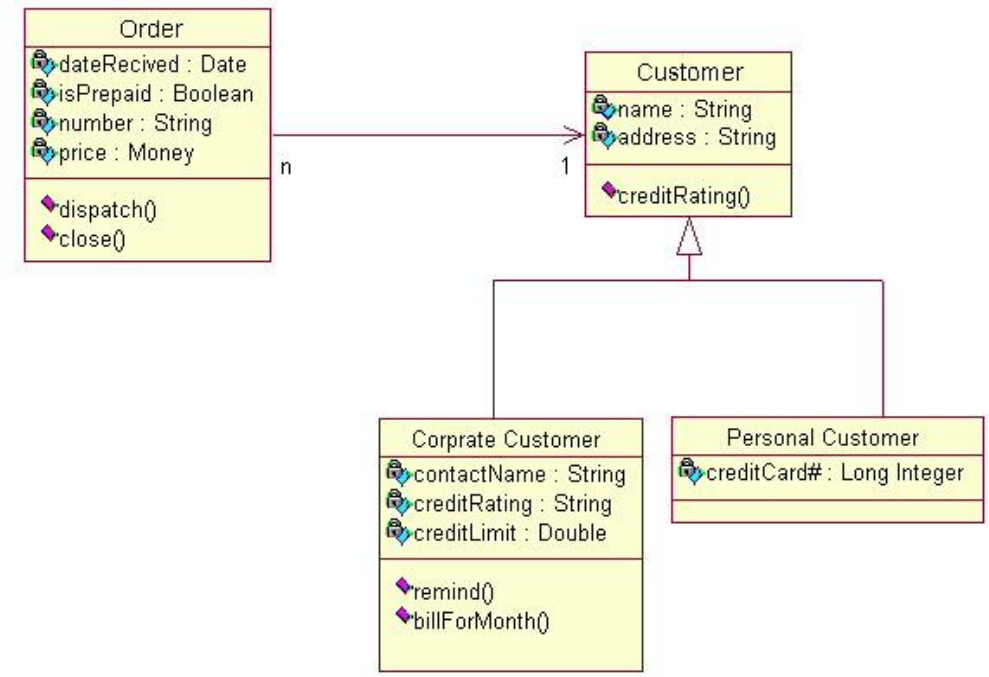
Class diagrams are used in nearly all Object Oriented software designs. Use them to describe the Classes of the system and their relationships to each other.

How to Draw: Class Diagrams

Class diagrams are some of the most difficult UML diagrams to draw. To draw detailed and useful diagrams a person would have to study UML and Object Oriented principles for a long time. Therefore, this page will give a very high level overview of the process. To find list of where to find more information see the Resources page.

Before drawing a class diagram consider the three different perspectives of the system the diagram will present; conceptual, specification, and implementation. Try not to focus on one perspective and try see how they all work together.

When designing classes consider what attributes and operations it will have. Then try to determine how instances of the classes will interact with each other. These are the very first steps of many in developing a class diagram. However, using just these basic techniques one can develop a complete view of the software system.



UML Interaction Diagram:

Overview:

From the name *Interaction* it is clear that the diagram is used to describe some type of interactions among the different elements in the model. So this interaction is a part of dynamic behaviour of the system.

This interactive behaviour is represented in UML by two diagrams known as *Sequence diagram* and *Collaboration diagram*. The basic purposes of both the diagrams are similar.
Sequence diagram emphasizes on time sequence of messages and collaboration diagram emphasizes on the structural organization of the objects that send and receive messages.

Purpose:

The purposes of interaction diagrams are to visualize the interactive behaviour of the system. Now visualizing interaction is a difficult task. So the solution is to use different types of models to capture the different aspects of the interaction.
That is why sequence and collaboration diagrams are used to capture dynamic nature but from a different angle.
So the purposes of interaction diagram can be describes as:

- To capture dynamic behaviour of a system.
- To describe the message flow in the system.
- To describe structural organization of the objects.
- To describe interaction among objects.

How to draw Component Diagram?

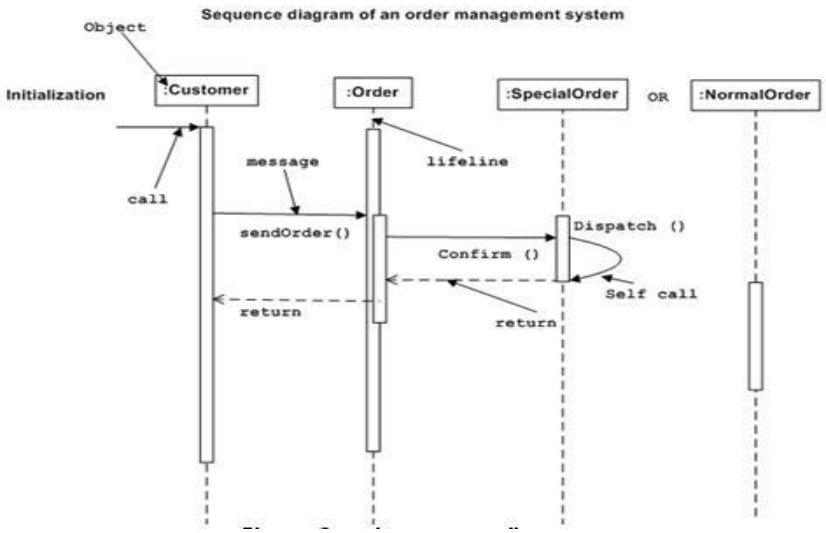
As we have already discussed that the purpose of interaction diagrams are to capture the dynamic aspect of a system. So to capture the dynamic aspect we need to understand what a dynamic aspect is and how it is visualized. Dynamic aspect can be defined as the snap shot of the running system at a particular moment.
We have two types of interaction diagrams in UML. One is sequence diagram and the other is a collaboration diagram. The sequence diagram captures the time sequence of message flow from one object to another and the collaboration diagram describes the organization of objects in a system taking part in the message flow.
So the following things are to identified clearly before drawing the interaction diagram:

Objects taking part in the interaction.
Message flows among the objects.
The sequence in which the messages are flowing.
Object organization.

Following are two interaction diagrams modeling order management system. The first diagram is a sequence diagram and the second is a collaboration diagram.

The Sequence Diagram:

The sequence diagram is having four objects (Customer, Order, SpecialOrder and NormalOrder).
The following diagram has shown the message sequence for *SpecialOrder* object and the same can be used in case of *NormalOrder* object. Now it is important to understand the time sequence of message flows. The message flow is nothing but a method call of an object.
The first call is *sendOrder ()* which is a method of *Order* object. The next call is *confirm ()* which is a method of *SpecialOrder* object and the last call is *Dispatch ()* which is a method of *SpecialOrder* object. So here the diagram is mainly describing the method calls from one object to another and this is also the actual scenario when the system is running.

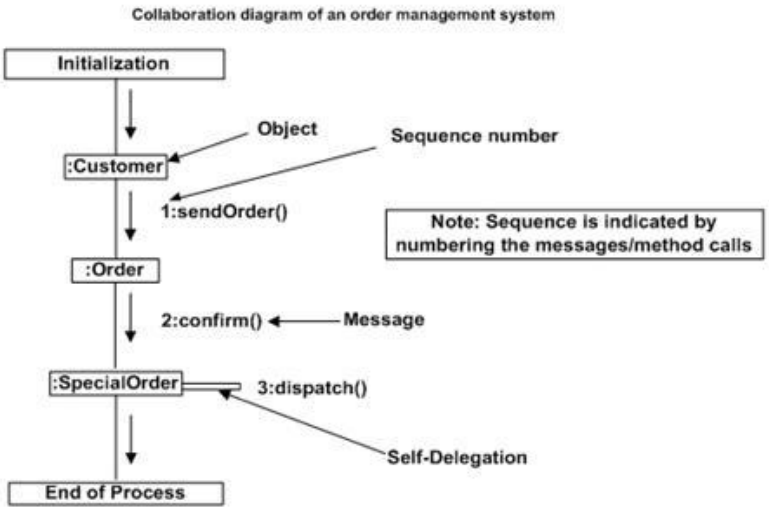


The Collaboration Diagram:

The second interaction diagram is collaboration diagram. It shows the object organization as shown below. Here in collaboration diagram the method call sequence is indicated by some numbering technique as shown below. The number indicates how the methods are called one after another. We have taken the same order management system to describe the collaboration diagram.

The method calls are similar to that of a sequence diagram. But the difference is that the sequence diagram does not describe the object organization where as the collaboration diagram shows the object organization.

Now to choose between these two diagrams the main emphasis is given on the type of requirement. If the time sequence is important then sequence diagram is used and if organization is required then collaboration diagram is used.



Where to use Interaction Diagrams?

We have already discussed that interaction diagrams are used to describe dynamic nature of a system. Now we will look into the practical scenarios where these diagrams are used. To understand the practical application we need to understand the basic nature of sequence and collaboration diagram.

The main purposes of both the diagrams are similar as they are used to capture the dynamic behaviour of a system. But the specific purposes are more important to clarify and understood.
Sequence diagrams are used to capture the order of messages flowing from one object to another. And the collaboration diagrams are used to describe the structural organizations of the objects taking part in the interaction. A single diagram is not sufficient to describe the dynamic aspect of an entire system so a set of diagrams are used to capture is as a whole.
The interaction diagrams are used when we want to understand the message flow and the structural organization. Now message flow means the sequence of control flow from one object to another and structural organization means the visual organization of the elements in a system.
In a brief the following are the usages of interaction diagrams:

- To model flow of control by time sequence.
- To model flow of control by structural organizations.
- For forward engineering.
- For reverse engineering.

UNIT 4 – GRASP

General Responsibility Assignment Software Patterns (or **Principles**), abbreviated **GRASP**, consists of guidelines for assigning responsibility to classes and objects in object-oriented design.
The different patterns and principles used in GRASP are: Information Expert, Creator, Controller, Low Coupling, High Cohesion, Polymorphism, Pure Fabrication, Indirection, Protected Variations. All these patterns answer some software problem, and in almost every case these problems are common to almost every software development project. These techniques have not been invented to create new ways of working but to better document and standardize old, tried-and-tested programming principles in object oriented design.
It has been said that "the critical design tool for software development is a mind well educated in design principles. It is not the UML or any other technology". Thus, GRASP is really a mental toolset, a learning aid to help in the design of object oriented software.

Creator

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 contains or compositely aggregates 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.

Information Expert

Information Expert is a principle used to determine where to delegate responsibilities. These responsibilities include methods, computed fields and so on.
Using the principle of Information Expert a general approach to assigning responsibilities is to look at a given responsibility, determine the information needed to fulfill it, and then determine where that information is stored.
Information Expert will lead to placing the responsibility on the class with the most information required to fulfill it.

Controller

The **Controller** pattern assigns the responsibility of dealing with system events to a non-UI class that represent the overall system or a use case scenario. A Controller object is a non-user interface object responsible for receiving or handling a system event.

A use case controller should be used to deal with *all* system events of a use case, and may be used for more than one use case (for instance, for use cases *Create User* and *Delete User*, one can have one *UserController*, instead of two separate use case controllers).

It is defined as the first object beyond the UI layer that receives and coordinates ("controls") a system operation. The controller should delegate to other objects the work that needs to be done; it coordinates or controls the activity. It should not do much work itself. The GRASP Controller can be thought of as being a part of the Application/Service layer (assuming that the application has made an explicit distinction between the App/Service layer and the Domain layer) in an object-oriented system with common layers.

Low Coupling

Low Coupling is an evaluative pattern, which dictates how to assign responsibilities to support:

- low dependency between classes;
- low impact in a class of changes in other classes;
- high reuse potential

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 adverse to change.

Polymorphism

According to **Polymorphism**, responsibility of defining the variation of behaviors based on type is assigned to the types for which this variation happens. This is achieved using polymorphic operations.

Pure Fabrication

A **pure fabrication** is a class that does not represent a concept in the problem domain, specially made up to achieve low coupling, high cohesion, and the reuse potential thereof derived (when a solution presented by the *Information Expert* pattern does not). This kind of class is called "Service" in Domain-driven design.

Indirection

The **Indirection** pattern supports low coupling (and reuse potential) between two elements by assigning the responsibility of mediation between them to an intermediate object. An example of this is the introduction of a controller component for mediation between data (model) and its representation (view) in the Model-view-controller pattern.

Protected Variations

The **Protected Variations** pattern protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability with an interface and using polymorphism to create various implementations of this interface.

Visibility

In object-oriented design, there is a notation of visibility for attributes and operations. UML identifies four types of visibility: public, protected, private, and package.

The UML specification does not require attributes and operations visibility to be displayed on the class diagram, but it does require that it be defined for each attribute or operation. To display visibility on the class diagram, you place the visibility mark in front of the attribute's or operation's name. Though UML specifies four visibility types, an actual programming language may add additional visibilities, or it may not support the UML-defined visibilities. Table 4 displays the different marks for the UML-supported visibility types.

MarkVisibility type

+ Public
# Protected
- Private
~ Package

Now, let's look at a class that shows the visibility types indicated for its attributes and operations. All the attributes and operations are public, with the exception of the updateBalance operation. The updateBalance operation is protected.

A BankAccount class that shows the visibility of its attributes and operations

Visibility

• Visibility — the ability of one object to "see" or have a reference to another object.

Visibility is required for one object to message another

Visibility

In common usage, **visibility** is the ability of an object to "see" or have a reference to another object. More generally, it is related to the issue of scope: Is one resource (such as an instance) within the scope of another? There are four common ways that visibility can be achieved from object *A* to object *B*:

- **Attribute visibility**—B is an attribute of A.
- **Parameter visibility**—B is a parameter of a method of A.
- **Local visibility**—B is a (non-parameter) local object in a method of A.
- **Global visibility**—B is in some way globally visible.

The motivation to consider visibility is this:

> For an object A to send a message to an object B, B must be visible to A.

For example, to create an interaction diagram in which a message is sent from a *Register* instance to a *ProductCatalog* instance, the *Register* must have visibility to the *ProductCatalog*. A typical visibility solution is that a reference to the *ProductCatalog* instance is maintained as an attribute of the *Register*.

Attribute Visibility

Attribute visibility from A to B exists when B is an attribute of A. It is a relatively permanent visibility because it persists as long as A and B exist. This is a very common form of visibility in object-oriented systems.

To illustrate, in a Java class definition for *Register*, a *Register* instance may have attribute visibility to a *ProductCatalog*, since it is an attribute (Java instance variable) of the *Register*.

```
public class Register
{
...
private ProductCatalog catalog;
...
}
```

This visibility is required because in the *enterItem* diagram shown in Figure 18.2, a *Register* needs to send the *getSpecification* message to a *ProductCatalog*:

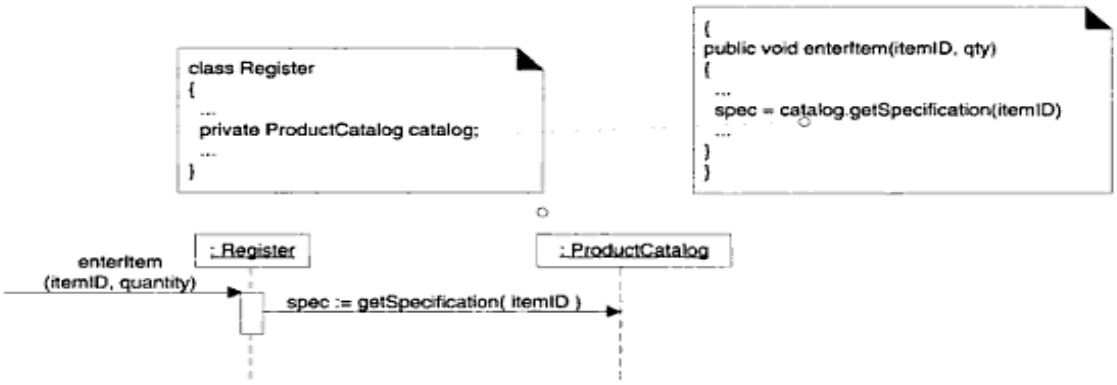


Figure 18.2 Attribute visibility.

Parameter Visibility

Parameter visibility from A to B exists when B is passed as a parameter to a method of A. It is a relatively temporary visibility because it persists only within the scope of the method. After attribute visibility, it is the second most common form of visibility in object-oriented systems.

To illustrate, when the *makeLineItem* message is sent to a *Sale* instance, a *ProductSpecification* instance is passed as a parameter. Within the scope of the *makeLineItem* method, the *Sale* has parameter visibility to a *ProductSpecification* (see Figure 18.3).

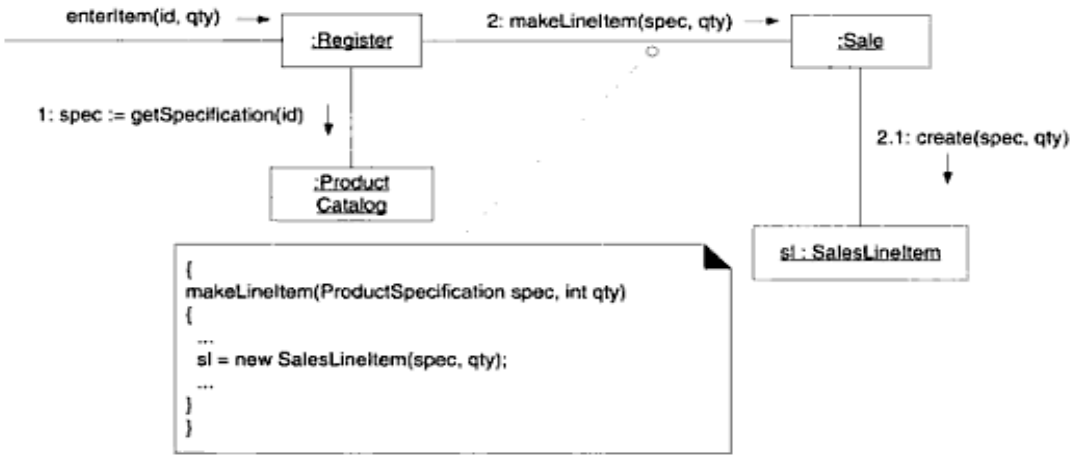


Figure 18.3 Parameter visibility.

It is common to transform parameter visibility into attribute visibility. For example, when the *Sale* creates a new *SalesLineItem*, it passes a *ProductSpecification* in to its initializing method (in C++ or Java, this would be its **constructor**). Within the initializing method, the parameter is assigned to an attribute, thus establishing attribute visibility (Figure 18.4).

Local Visibility

Local visibility from A to B exists when B is declared as a local object within a method of A. It is a relatively temporary visibility because it persists only within the scope of the method. After parameter visibility, it is the third most common form of visibility in object-oriented systems.

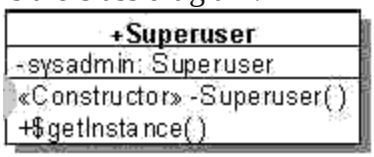
Applying GOF Design Patterns:

Design patterns represent common software problems and the solutions to those problems in a formal manner. They were inspired by a book written by architect Christopher Alexander. Patterns were introduced in the software world by another book: "Design Patterns: Elements of Reusable Object-Oriented Software", by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. These people were nicknamed the "Gang of Four" for some mysterious reason. The Gang of Four describes 23 design patterns. With patterns you don't have to reinvent the wheel and get proven solutions for frequently encountered problems. Many books and articles have been written on this subject. This means that design patterns are becoming common knowledge,

which leads to better communication. To summarize design patterns save time, energy while making your life easier.

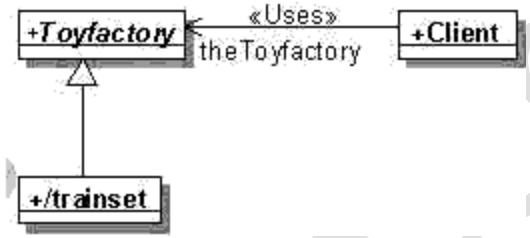
Singleton

The singleton pattern deals with situations where only one instance of a class must be created. Take the case of a system administrator or superuser. This person has the right to do everything in a computer system. In addition we will also have classes representing normal users. Therefore we must ensure that these classes have no access to the super user constructor. The solution to this problem in C++ and Java is to declare the superuser constructor private. The superuser class itself has a private static attribute sysadmin, which is initialised using the class constructor. Now we get an instance of the super user class with a public static method that returns sysadmin. Here is the class diagram:



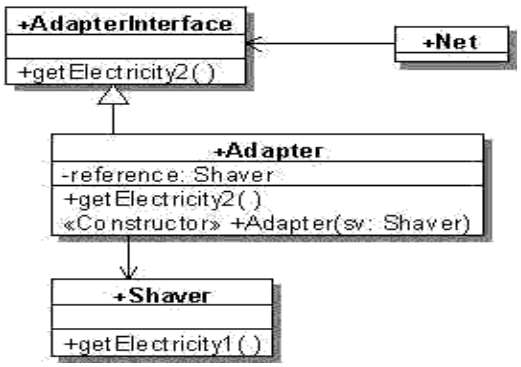
Factory Method

The Factory Method pattern deals with situations where at runtime one of several similar classes must be created. Visualise this as a factory that produces objects. In a toy factory for instance we have the abstract concept of toy. Every time we get a request for a new toy a decision must be made - what kind of a toy to manufacture. Similarly to the Singleton pattern the Factory Method pattern utilises a public static accessor method. In our example the abstract Toyfactory class will have a getInstance() method, which is inherited by its non abstract subclasses.



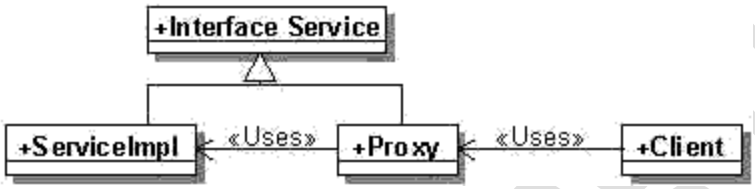
Adapter

Sometimes you will have two classes that can in principle work well together, but they can't interface with each other for some reason. This kind of problem occurs when travelling abroad and you carry an electric shaver with you. Although it will work perfectly when you are at home. There can be problems in a foreign country, because of a different standard voltage. The solution is to use an adapter. Let's turn our attention back to the software domain. Here we will have an interface which defines new methods for example getElectricity2. An adapter class will wrap around the Shaver class. The adapter class will implement the interface with the new method.



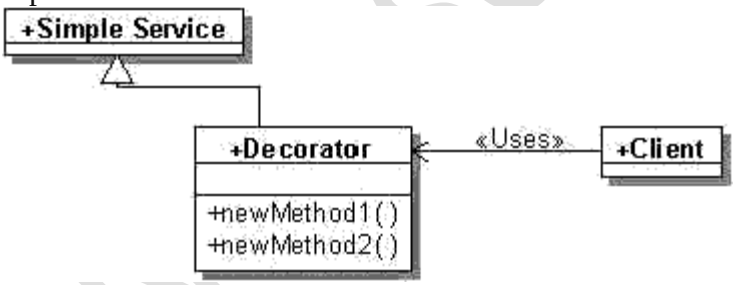
Proxy

The Proxy pattern deals with situations where you have a complex object or it takes a long time to create the object. The solution to this problem is to replace the complex object with a simple 'stub' object that has the same interface. The stub acts as a body double. This is the strategy used by the Java Remote Method Invocation API. The reason to use the proxy pattern in this case is that the object is on a remote server causing network overhead. Other reasons to use the proxy can be restricted access (Java applets for example) or time consuming calculations.



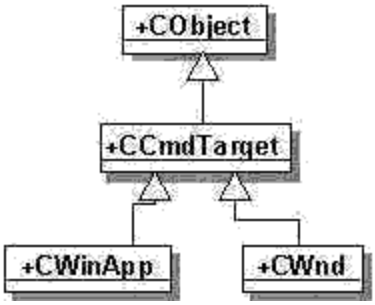
Decorator

The Decorator is usually a subclass, that is a body double for its superclass or another class with identical interface. The goal of the Decorator pattern is to add or improve the capabilities of the super class.



Composite

The composite is often encountered in GUI packages like for instance the Java Abstract Windwing Toolkit (AWT) or Microsoft Foundation (MFC) classes. All objects in this pattern have a common abstract superclass that descibes basic object conduct. The base class in the MFC hierarchy is CObject. It provides functions for debugging and serialization. All the MFC classes even the most basic ones inherit these facilities.



Observer and MVC

An application with Model - View - Controller setup usually uses the Observer Pattern. In a Java webserver environment the model will be represented by Java classes encompassing the business logic, the view is represented by Java Server Pages which display HTML in the client's browser and we will have a Servlets as Controllers. The observer pattern strategy is to have view components take subscriptions for their model. This ensures that they will get notified if the model changes.

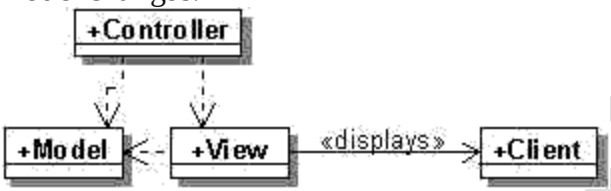
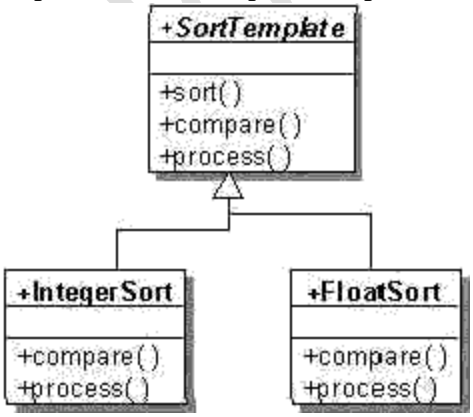


Figure - Observer and MVC Class Diagram

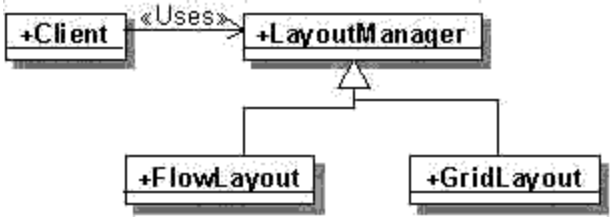
Template

In the good old days before OOP writing functions was the recommended thing to do. A sort algorithm would be implement by half dozen of functions, one sort function for integers, one sort function for floating points, one sort function for doubles etc. These functions are so similar that nobody in their right mind will type them letter by letter. Instead a programmer will write a template and copy the template several times. After that it's just a matter of writing down datatypes as appropriate. Thanks to OOP and the Template Design Pattern less code is required for this task. First we need to define an abstract Template class let's call it SortTemplate and it will have methods sort, compare and process (performs one cycle of the algorithm). Then we define concrete classes for each datatype. These classes are subclasses of SortTemplate and implement the compare and process methods.



Strategy

The Strategy Design Pattern makes it possible choose an implementation of an algorithm at run time. The implementation classes implement the same interface. In the case of the Java AWT Layout classes, the common interface is LayoutManager.



Summary

Design patterns are a hot research item. New patterns are emerging every day. In the future design patterns will be integrated in development tools. The main advantages of design patterns:

- Provide proven solutions
- Simplify complex problems
- Improve communication

Classification and list

Design patterns were originally grouped into the categories: creational patterns, structural patterns, and behavioral patterns, and described using the concepts of delegation, aggregation, and consultation. For further background on object-oriented design, see coupling and cohesion, inheritance, interface, and polymorphism. Another classification has also introduced the notion of architectural design pattern that may be applied at the architecture level of the software such as the Model-View-Controller pattern.

Creational Design Patterns

In software engineering, **creational design patterns** are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

Some examples of creational design patterns include:

- Abstract factory pattern: centralize decision of what factory to instantiate
- Factory method pattern: centralize creation of an object of a specific type choosing one of several implementations
- Builder pattern: separate the construction of a complex object from its representation so that the same construction process can create different representations
- Lazy initialization pattern: tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed
- Object pool pattern: avoid expensive acquisition and release of resources by recycling objects that are no longer in use

- Prototype pattern: used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects
- Singleton pattern: restrict instantiation of a class to one object

Behavioral pattern

In software engineering, **behavioral design patterns** are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

Examples of this type of design pattern include:

- Chain of responsibility pattern: Command objects are handled or passed on to other objects by logic-containing processing objects
- Command pattern: Command objects encapsulate an action and its parameters
- "Externalize the Stack": Turn a recursive function into an iterative one that uses a stack.
- Interpreter pattern: Implement a specialized computer language to rapidly solve a specific set of problems
- Iterator pattern: Iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation
- Mediator pattern: Provides a unified interface to a set of interfaces in a subsystem
- Memento pattern: Provides the ability to restore an object to its previous state (rollback)
- Null Object pattern: designed to act as a default value of an object
- Observer pattern: aka Publish/Subscribe or Event Listener. Objects register to observe an event which may be raised by another object
    - Weak reference pattern: De-couple an observer from an observable.
- Protocol stack: Communications are handled by multiple layers, which form an encapsulation hierarchy.
- State pattern: A clean way for an object to partially change its type at runtime
- Strategy pattern: Algorithms can be selected on the fly
- Specification pattern: Recombinable Business logic in a boolean fashion
- Template method pattern: Describes the program skeleton of a program
- Visitor pattern: A way to separate an algorithm from an object
- Single-serving visitor pattern: Optimise the implementation of a visitor that is allocated, used only once, and then deleted
- Hierarchical visitor pattern: Provide a way to visit every node in a hierarchical data structure such as a tree.
- Scheduled-task pattern: A task is scheduled to be performed at a particular interval or clock time (used in real-time computing)

Concurrency pattern

In software engineering, **concurrency patterns** are those types of design patterns that deal with multi-threaded programming paradigm. Examples of this class of patterns include:

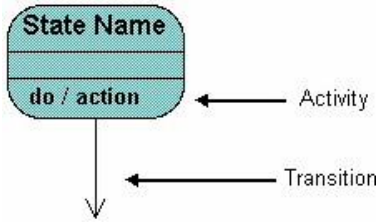
- Active Object
- Balking pattern
- Double checked locking pattern
- Guarded suspension
- Leaders/followers pattern
- Monitor Object
- Read write lock pattern
- Scheduler pattern
- Thread pool pattern
- Thread-Specific Storage
- Reactor pattern

UNIT 5 - UML state diagrams and modeling

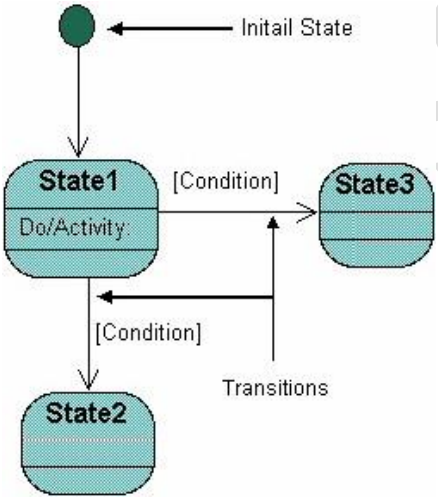
State diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur. Each diagram usually represents objects of a single class and track the different states of its objects through the system.

When to Use: State Diagrams
Use state diagrams to demonstrate the behavior of an object through many use cases of the system. Only use state diagrams for classes where it is necessary to understand the behavior of the object through the entire system. Not all classes will require a state diagram and state diagrams are not useful for describing the collaboration of all objects in a use case. State diagrams are other combined with other diagrams such as interaction diagrams and activity diagrams.

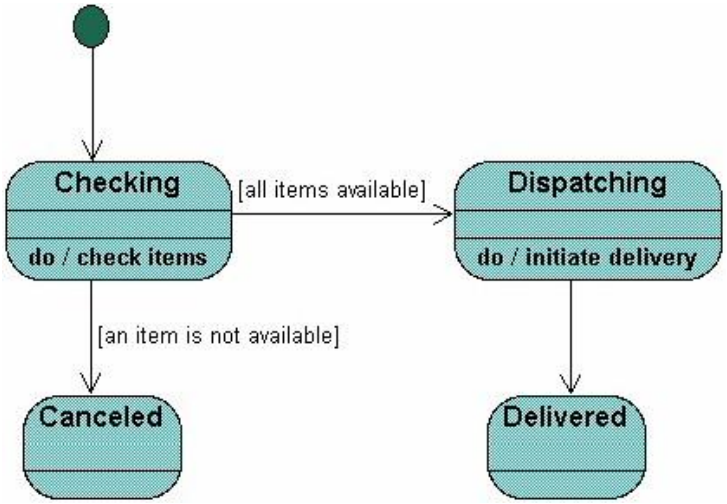
How to Draw: State Diagrams
State diagrams have very few elements. The basic elements are rounded boxes representing the state of the object and arrows indicting the transition to the next state. The activity section of the state symbol depicts what activities the object will be doing while it is in that state.



All state diagrams being with an initial state of the object. This is the state of the object when it is created. After the initial state the object begins changing states. Conditions based on the activities can determine what the next state the object transitions to.

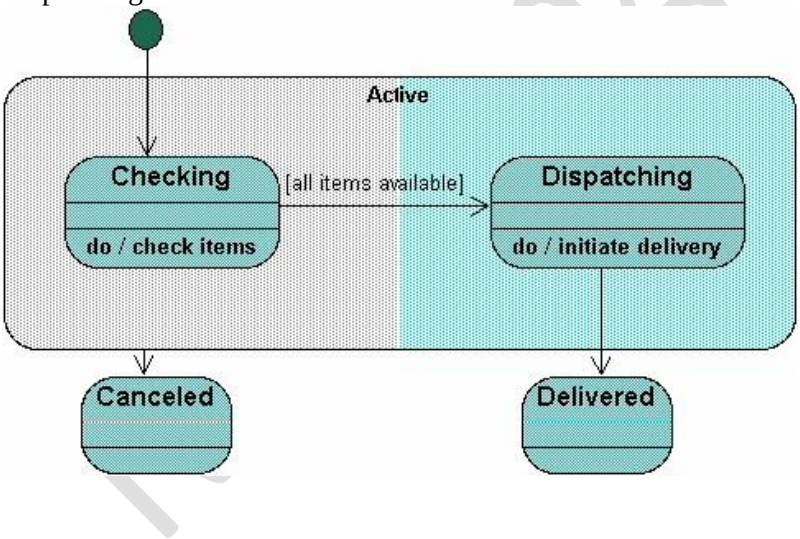


Below is an example of a state diagram might look like for an Order object. When the object enters the Checking state it performs the activity "check items." After the activity is completed the object transitions to the next state based on the conditions [all items available] or [an item is not available]. If an item is not available the order is canceled. If all items are available then the order is dispatched. When the object transitions to the Dispatching state the activity "initiate delivery" is performed. After this activity is complete the object transitions again to the Delivered state.



State diagrams can also show a super-state for the object. A super-state is used when many transitions lead to the a certain state. Instead of showing all of the transitions from each state to the redundant state a super-state can be used to show that all of the states inside of the super-state can transition to the redundant state. This helps make the state diagram easier to read.

The diagram below shows a super-state. Both the Checking and Dispatching states can transition into the Canceled state, so a transition is shown from a super-state named Active to the state Cancel. By contrast, the state Dispatching can only transition to the Delivered state, so we show an arrow only from the Dispatching state to the Delivered state.



Activity Diagrams

Activity diagrams describe the workflow behavior of a system. Activity diagrams are similar to state diagrams because activities are the state of doing something. The diagrams describe the state of activities by showing the sequence of activities performed. Activity diagrams can show activities that are conditional or parallel.

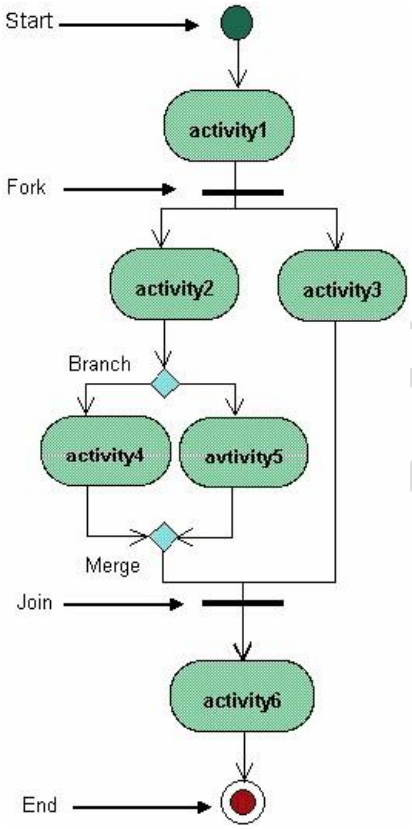
When to Use: Activity Diagrams

Activity diagrams should be used in conjunction with other modeling techniques such as interaction diagrams and state diagrams. The main reason to use activity diagrams is to model the workflow behind the system being designed. Activity Diagrams are also useful for: analyzing a use case by describing what actions need to take place and when they should occur; describing a complicated sequential algorithm; and modeling applications with parallel processes.

However, activity diagrams should not take the place of interaction diagrams and state diagrams. Activity diagrams do not give detail about how objects behave or how objects collaborate.

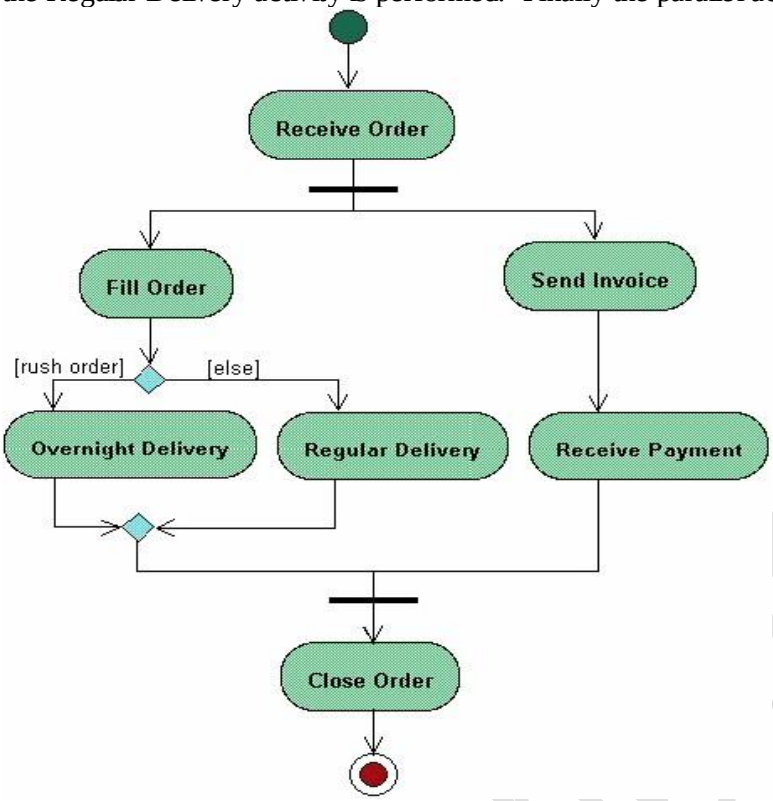
How to Draw: Activity Diagrams

Activity diagrams show the flow of activities through the system. Diagrams are read from top to bottom and have branches and forks to describe conditions and parallel activities. A fork is used when multiple activities are occurring at the same time. The diagram below shows a fork after activity1. This indicates that both activity2 and activity3 are occurring at the same time. After activity2 there is a branch. The branch describes what activities will take place based on a set of conditions. All branches at some point are followed by a merge to indicate the end of the conditional behavior started by that branch. After the merge all of the parallel activities must be combined by a join before transitioning into the final activity state.



Below is a possible activity diagram for processing an order. The diagram shows the flow of actions in the system's workflow. Once the order is received the activities split into two parallel sets of activities. One side fills and sends the order while the other handles the billing. On the Fill Order side, the method

of delivery is decided conditionally. Depending on the condition either the Overnight Delivery activity or the Regular Delivery activity is performed. Finally the parallel activities combine to close the order.



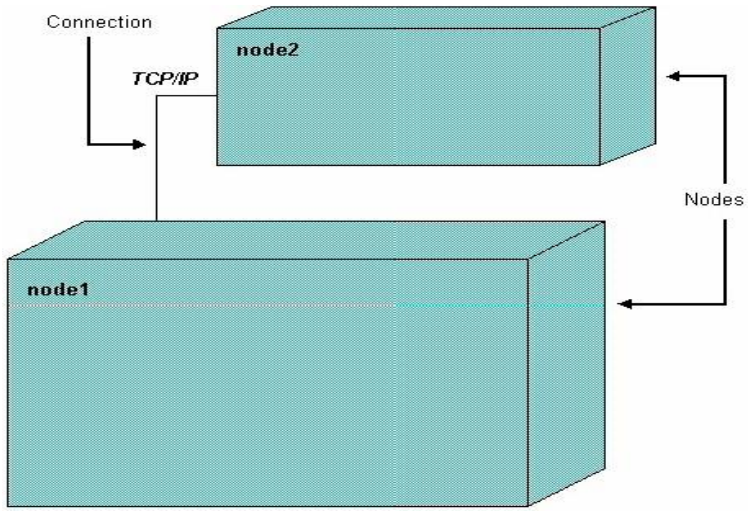
1

Physical Diagrams:
There are two types of physical diagrams: **deployment diagrams** and **component diagrams.** Deployment diagrams show the physical relationship between hardware and software in a system. Component diagrams show the software components of a system and how they are related to each other. These relationships are called dependencies.

When to Use: Physical Diagrams
Physical diagrams are used when development of the system is complete. Physical diagrams are used to give descriptions of the physical information about a system.

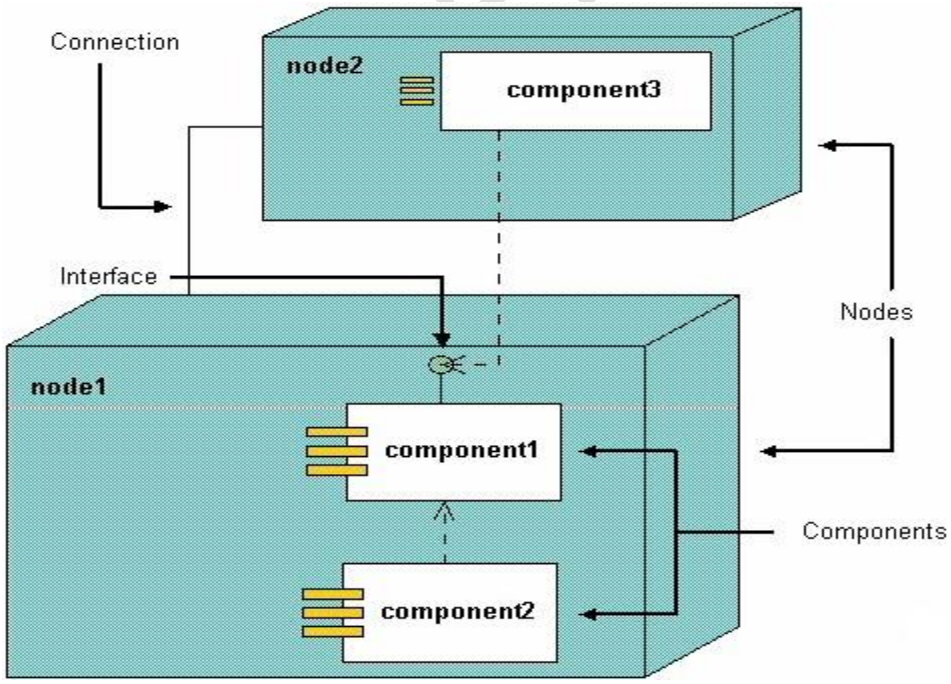
How to Draw: Physical Diagrams
Many times the deployment and component diagrams are combined into one physical diagram. A combined deployment and component diagram combines the features of both diagrams into one diagram.

The deployment diagram contains nodes and connections. A node usually represents a piece of hardware in the system. A connection depicts the communication path used by the hardware to communicate and usually indicates a method such as TCP/IP.



The component diagram contains components and dependencies. Components represent the physical packaging of a module of code. The dependencies between the components show how changes made to one component may affect the other components in the system. Dependencies in a component diagram are represented by a dashed line between two or more components. Component diagrams can also show the interfaces used by the components to communicate to each other.

The combined deployment and component diagram below gives a high level physical description of the completed system. The diagram shows two nodes which represent two machines communicating through TCP/IP. Component2 is dependent on component1, so changes to component 2 could affect component1. The diagram also depicts component3 interfacing with component1. This diagram gives the reader a quick overall view of the entire system.



UML MODELING

It is very important to distinguish between the UML model. Different diagrams are used for different type of UML modeling. There are three important type of UML modelings:

Structural modeling:
Structural modeling captures the static features of a system. They consist of the followings:

- Classes diagrams
- Objects diagrams
- Deployment diagrams
- Package diagrams
- Composite structure diagram
- Component diagram

Structural model represents the framework for the system and this framework is the place where all other components exist. So the class diagram, component diagram and deployment diagrams are the part of structural modeling. They all represent the elements and the mechanism to assemble them. But the structural model never describes the dynamic behavior of the system. Class diagram is the most widely used structural diagram.

Behavioral Modeling:
Behavioral model describes the interaction in the system. It represents the interaction among the structural diagrams. Behavioral modeling shows the dynamic nature of the system. They consist of the following:

- Activity diagrams
- Interaction diagrams
- Use case diagrams

All the above show the dynamic sequence of flow in a system.

Architectural Modeling:
Architectural model represents the overall framework of the system. It contains both structural and behavioral elements of the system. Architectural model can be defined as the blue print of the entire system. Package diagram comes under architectural modeling.

UML Operation Contract

A UML Operation contract identifies system state changes when an operation happens. Effectively, it will define what each system operation does. An operation is taken from a system sequence diagram. It is a single event from that diagram. A domain model can be used to help generate an operation contract. The domain model can be marked as follows to help with the operation contract:

- Green - Pre existing concepts and associations.
- Blue - Created associations and concepts.
- Red - Destroyed concepts and associations.

Operation Contract Syntax

Name: appropriateName

Responsibilities: Perform a function

Cross References: System functions and Use Cases

Exceptions: none

Preconditions: Something or some relationship exists

Postconditions: An association was formed

When making an operation contract, think of the state of the system before the action (snapshot) and the state of the system after the action (a second snapshot). The conditions both before and after the action should be described in the operation contract. Do not describe how the action or state changes were done. The pre and post conditions describe state, not actions.

Typical postcondion changes:

- Object attributes were changed.
- An instance of an object was created.
- An association was formed or broken.

Postconditions are described in the past tense. They declare state changes to the system. Fill in the name, then responsibilities, then postconditions.

UML Deployment Diagram

Overview:

Deployment diagrams are used to visualize the topology of the physical components of a system where the software components are deployed.
So deployment diagrams are used to describe the static deployment view of a system. Deployment diagrams consist of nodes and their relationships.

Purpose:

The name Deployment itself describes the purpose of the diagram. Deployment diagrams are used for describing the hardware components where software components are deployed. Component diagrams and deployment diagrams are closely related. Component diagrams are used to describe the components and deployment diagrams shows how they are deployed in hardware.

UML is mainly designed to focus on software artifacts of a system. But these two diagrams are special diagrams used to focus on software components and hardware components.
So most of the UML diagrams are used to handle logical components but deployment diagrams are made to focus on hardware topology of a system. Deployment diagrams are used by the system engineers.

The purpose of deployment diagrams can be described as:

- Visualize hardware topology of a system.
- Describe the hardware components used to deploy software components.
- Describe runtime processing nodes.

How to draw Component Diagram?

Deployment diagram represents the deployment view of a system. It is related to the component diagram. Because the components are deployed using the deployment diagrams. A deployment diagram consists of nodes. Nodes are nothing but physical hardwares used to deploy the application.

Deployment diagrams are useful for system engineers. An efficient deployment diagram is very important because it controls the following parameters

- Performance
- Scalability
- Maintainability
- Portability

So before drawing a deployment diagram the following artifacts should be identified:

- Nodes
- Relationships among node

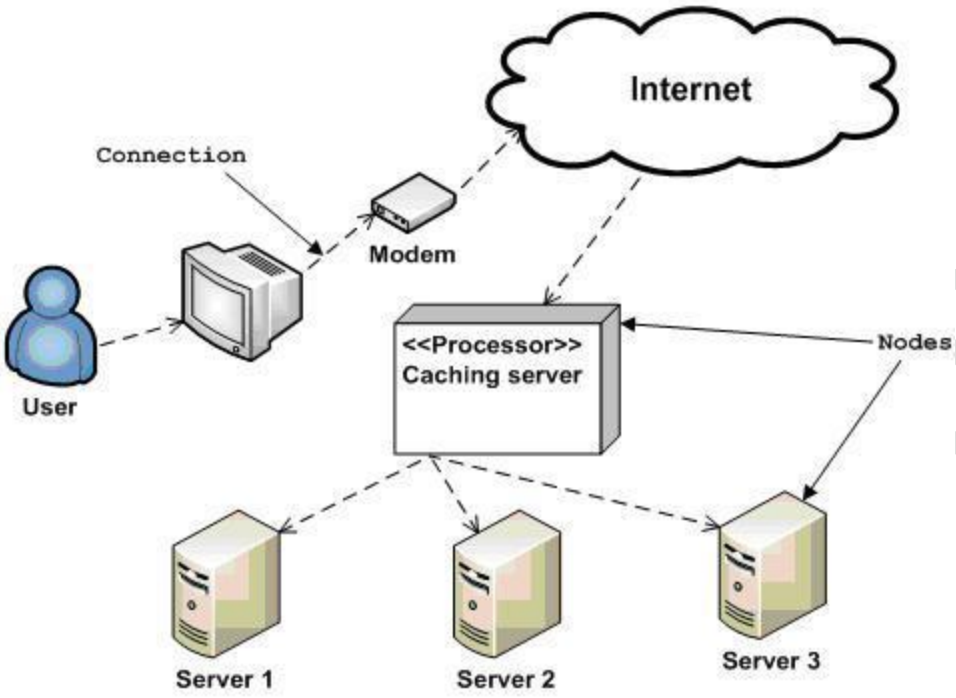
The following deployment diagram is a sample to give an idea of the deployment view of order management system. Here we have shown nodes as:

- Monitor
- Modem
- Caching server
- Server

The application is assumed to be a web based application which is deployed in a clustered environment using server 1, server 2 and server 3. The user is connecting to the application using internet. The control is flowing from the caching server to the clustered environment.

So the following deployment diagram has been drawn considering all the points mentioned above:

Deployment diagram of an order management system



Where to use Deployment Diagrams?

Deployment diagrams are mainly used by system engineers. These diagrams are used to describe the physical components (hardwares), their distribution and association. To clarify it in details we can visualize deployment diagrams as the hardware components/nodes on which software components reside.

Software applications are developed to model complex business processes. Only efficient software applications are not sufficient to meet business requirements. Business requirements can be described as to support increasing number of users, quick response time etc.
To meet these types of requirements hardware components should be designed efficiently and in a cost effective way.

Now a day's software applications are very complex in nature. Software applications can be stand alone, web based, distributed, mainframe based and many more. So it is very important to design the hardware components efficiently.

So the usage of deployment diagrams can be described as follows:

- To model the hardware topology of a system.
- To model embedded system.
- To model hardware details for a client/server system.
- To model hardware details of a distributed application.
- Forward and reverse engineering.

Deployment Diagrams

Deployment diagram shows execution architecture of systems that represent the assignment (deployment) of software artifacts to deployment targets (usually nodes).

Nodes represent either hardware devices or software execution environments. They could be connected through communication paths to create network systems of arbitrary complexity. Artifacts represent concrete elements in the physical world that are the result of a development process and are deployed on nodes.

Note, that components were directly deployed to nodes in UML 1.x deployment diagrams. In UML 2.x artifacts are deployed to nodes, and artifacts could manifest components. So components are now deployed to nodes indirectly through artifacts.

The following nodes and edges are typically drawn in a UML deployment diagram: artifact, **association** between artifacts, **dependency** between artifacts, component, manifestation, node, device, execution environment, **composition** of nodes, **communication path**, **deployment specification**, deployment specification **dependency**, deployment specification **association**, deployment.

You can find some deployment diagrams examples here:

- Web Application Deployment
- Clustered Deployment of J2EE Web Application
- Apple iTunes Deployment

Artifact

An **artifact** is a classifier that represents some **physical entity**, piece of information that is used or is produced by a software development process, or by deployment and operation of a system. Artifact is source of a deployment to a node. A particular instance (or "copy") of an artifact is deployed to a node instance.

Artifacts may have **properties** that represent features of the artifact, and operations that can be performed on its instances. Artifacts have **fileName** attribute - a concrete name that is used to refer to the artifact in a physical context - e.g. **file name** or **URI**.

Some real life examples of artifacts are:

- model file
- source file
- script
- binary executable file
- text document
- mail message
- table in a database

The **UML Standard Profile** defines several **standard stereotypes** that apply to artifacts:

«file» A physical file in the context of the system developed.

Standard stereotypes - subclasses of **«file»**:

«document» A generic file that is not a «source» file or «executable».

«source» A source file that can be compiled into an executable file.

«library» A static or dynamic library file.

«executable» A program file that can be executed on a computer system.

«script» A script file that can be interpreted by a computer system.

Standard UML 1.x stereotype that is now **obsolete**:

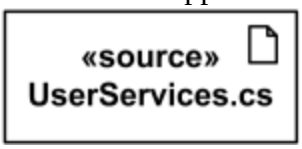
«table» Table in database.

Standard stereotypes can be further specialized into implementation and platform specific stereotypes in profiles. For example, an EJB profile might define **«jar»** as a subclass of «executable» for executable Java archives. Specific profiles are expected to stereotype artifact to model sets of files (e.g., as characterized by a "file extension" on a file system).

Artifacts are deployed to a deployment target. **Instance specification** was extended in UML to allow instances of **artifacts** to be **deployed artifacts** in a deployment relationship.

An artifact is presented using an ordinary class rectangle with the keyword **«artifact»**. Examples in **UML** specification also show **document icon** in upper right corner.

Artifact web-app.war



Source file artifact UserServices.cs

Library commons.dll

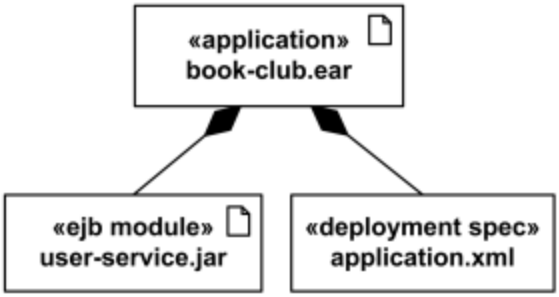
Alternatively, artifact may be depicted by an icon.

Artifact web-tools-lib.jar

Optionally, the underlining of the name of an artifact instance may be omitted, as the context is assumed to be known to users.

Associations Between Artifacts

Artifacts can be involved in associations to other artifacts, e.g. composition associations. For instance, a deployment descriptor artifact for a component may be contained within the artifact that manifests that component. In that way, the component and its descriptor are deployed to a node instance as one artifact instance.

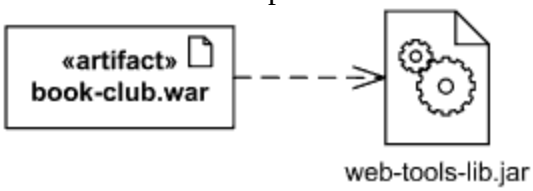


Application book-club.ear artifact contains EJB user-service.jar artifact and deployment descriptor.

Dependency Between Artifacts

Artifacts can be involved in dependency relationship with other artifacts.

Dependency between artifacts is notated in the same way as general dependency, i.e. as a general dashed line with an open arrow head directed from client artifact to supplier artifact.



The book-club.war artifact depends on web-tools-lib.jar artifact.

Artifact Manifestation

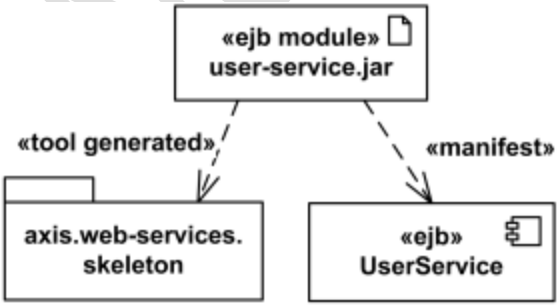
Manifestation is an abstraction relationship which represents the concrete physical rendering of one or more model elements by an **artifact** or utilization of the model elements in the construction or generation of the artifact. An artifact **manifests** one or more model elements.

Note, that since UML 2.0 artifacts can manifest any packageable element, not just component as it was in previous versions of UML.

The artifact **owns** the manifestations, each representing the utilization of a packageable element.

Specific **profiles** are expected to stereotype the manifestation relationship to indicate particular forms of manifestation. For example, «**tool generated**» and «**custom code**» might be two manifestations for different classes embodied in an artifact.

A manifestation is notated in the same way as abstraction dependency, i.e. as a dashed line with an open arrow head directed from **artifact** to **packageable element**, (e.g. to **component** or **package**) and is labeled with the keyword «**manifest**».



EJB component UserService and skeleton of web services are manifested by EJB module user-service.jar artifact

In **UML 1.x**, the concept of manifestation was referred to as **implementation** and annotated as «**implement**». Since this was one of the many uses of the word "implementation" this has been replaced in UML 2.x by «**manifest**».

Deployment Target

Artifacts are deployed to deployment targets. **Deployment target** is the location for a deployed artifact.

Deployment target owns the set of deployment that target it.

Deployment target is specialized by:

- node
- **property**

Instance specification was extended in **UML 2.0** to allow instance of a **node** to be **deployment target** in a deployment relationship.

Property was also extended in **UML 2.0** with the capability of being a **deployment target** in a deployment relationship. This enables modeling the deployment to hierarchical nodes that have properties functioning as internal parts.

Deployment target has no specific notation by itself, see notations for subclasses.

Node

A **Node** is a deployment target which represents computational resource upon which **artifacts** may be deployed for execution.

A Node is shown as a perspective, 3-dimensional view of a cube.

Application Server Node

Node is **associated** with a **Deployment** of an Artifact. It is also **associated** with a set of Elements that are deployed on it. This is a derived association in that these **Packageable Elements** are involved in a **Manifestation** of an Artifact that is deployed on the Node.

Nodes may have an internal structure defined in terms of **parts** and **connectors** associated with them for advanced modeling applications. Parts of node could be solely of type Node. **Hierarchical nodes** (i.e., nodes within nodes) can be modeled using **composition associations**, or by defining an internal structure for advanced modeling applications.

Nodes can be interconnected through **communication paths** to define network structures. Communication paths can be defined between nodes such as "**application server**" and "**client workstation**" to define the possible communication paths between nodes. Specific network topologies can then be defined through **links** between node instances.

Node is specialized by:

- device
- execution environment

Device

A **device** is a subclass of node which represents a physical computational resource with processing capability upon which artifacts may be deployed for execution.

A **device** is rendered as a node (perspective, 3-dimensional view of a cube) annotated with keyword «**device**».

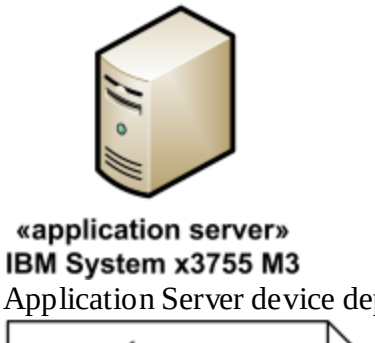
Application Server device

Device may be depicted using custom icon.

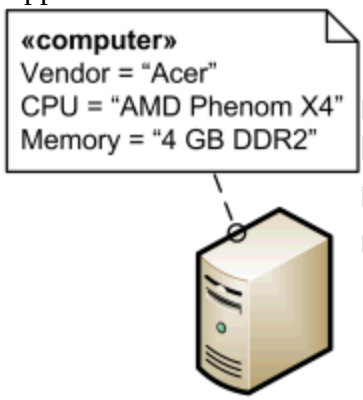
UML provides no standard stereotypes for devices. Examples of non-normative stereotypes for devices are:

- «application server»
- «client workstation»
- «mobile device»
- «embedded device»

Profiles, stereotypes, and tagged values could be used to provide custom icons and properties for the devices.



Application Server device depicted using custom icon



Computer stereotype with tags applied to Device class.

Database Server device depicted using custom icon

Mobile smartphone device depicted using custom icon

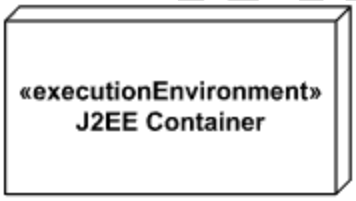
Devices may be complex (i.e., they may consist of other devices) where a physical machine is decomposed into its elements, either through namespace ownership or through attributes that are typed by Devices.

Execution Environment

An **execution environment** is a node that offers an execution environment for specific types of component that are deployed on it in the form of executable artifacts. Components of the appropriate type are deployed to specific execution environment nodes.

Execution environment implements a standard set of **services** that components require at execution time (at the modeling level these services are usually implicit). For each deployment of component, aspects of these services may be determined by properties in a **deployment specification** for a particular kind of execution environment.

Execution environment is notated as a node (perspective, 3-dimensional view of a cube) annotated with the standard stereotype «**executionEnvironment**».

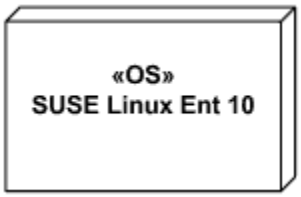


Execution environment - J2EE Container

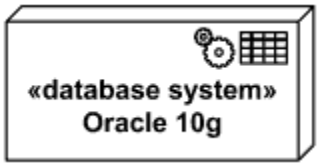
This «**executionEnvironment**» is pesky sesquipedalian to use. UML provides no other standard stereotypes for execution environments. Examples of reasonable non-normative stereotypes are:

- «OS»
- «workflow engine»
- «database system»
- «J2EE container»

- «web server»
- «web browser»



Linux Operating System Execution Environment

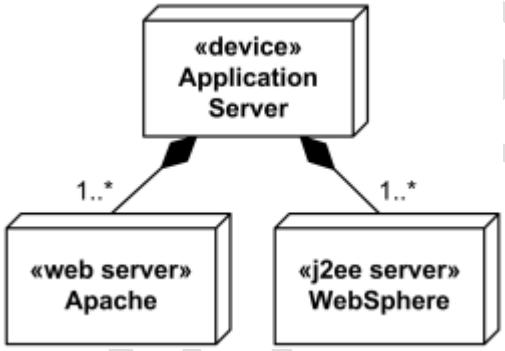


Oracle 10g DBMS Execution Environment

An execution environment can optionally have an explicit interface of **system level services** that can be used by the deployed elements, in those cases where the modeler wants to make the execution environment software execution environment services explicit.

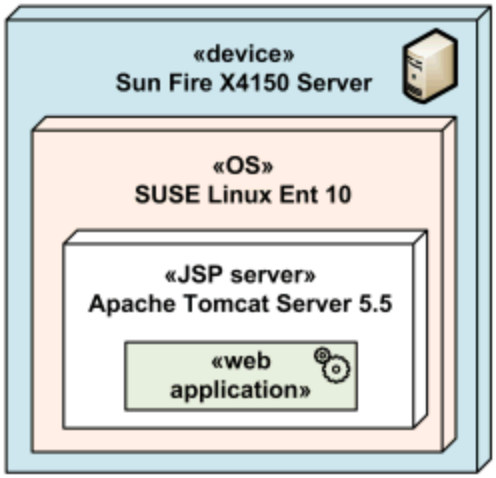
Composition of Nodes

Nodes may have an internal structure defined in terms of parts and **connectors** associated with them for advanced modeling applications. Parts of node could be only of type node. **Hierarchical nodes** (i.e., nodes within nodes) can be modeled using composite associations, or by defining an internal structure for advanced modeling applications.



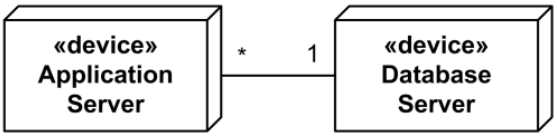
Application server box runs several web servers and J2EE servers

Execution environment is usually part of a general node or «device» which represents the physical hardware environment on which this execution environment resides. Execution environments can be **nested** (e.g., a database execution environment may be nested in an operating system execution environment).

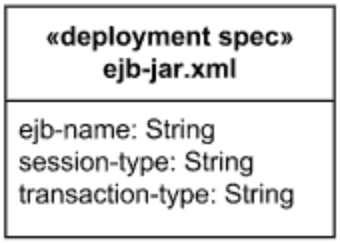


Several execution environments nested into server device
Execution environment instances are assigned to node instances by using composite associations between nodes and execution environments, where the execution environment plays the role of the part.

Communication Path
A **communication path** is an **association** between two **deployment targets**, through which they are able to exchange signals and messages.
Communication path is notated as **association**, and it has no additional notation compared to association.

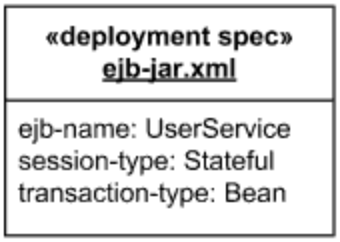


Communication path between several application servers and database server.
Deployment Specification
A **deployment specification** is an **artifact** that specifies a set of deployment properties that determine execution parameters of a **component artifact** that is deployed on a **node**. A deployment specification can be aimed at a specific type of **container** for a **components**.
A deployment specification is a general mechanism to parameterize a **deployment relationship**, as is common in various hardware and software technologies. The **deployment specification** element is expected to be extended in specific component profiles. Non-normative examples of the standard stereotypes that a profile might add to deployment specification are, for example, «**concurrencyMode**» with tagged values {thread, process, none}, or «**transactionMode**» with tagged values {transaction, nestedTransaction, none}.
A **deployment specification** at **specification level** is graphically displayed as a classifier rectangle with optional deployment properties in a compartment.



The ejb-jar.xml deployment specification

An artifact that reifies or implements deployment specification properties is a **deployment descriptor**. A **deployment specification** at **instance level** is graphically displayed as a classifier rectangle with the name underlined and with deployment properties having specific values in a compartment.

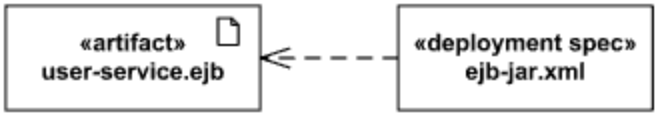


The ejb-jar.xml deployment descriptor

An instance of a deployment specification with specific values for deployment properties may be contained in a complex artifact.

Deployment Specification Dependency

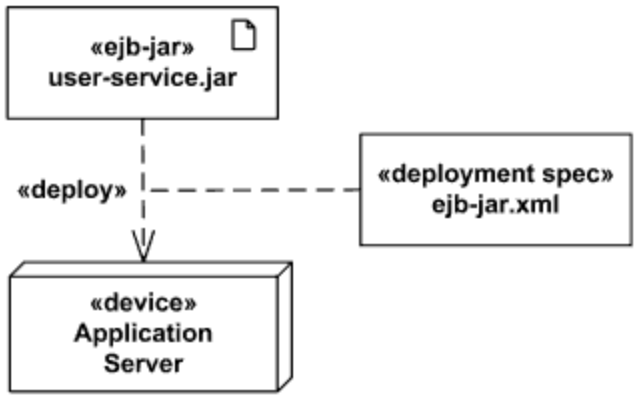
A **deployment specification** could be displayed as a classifier rectangle attached to a **component artifact** using a regular **dependency** arrow pointing to deployed artifact.



The ejb-jar.xml deployment specification for user-service.ejb artifact.

Deployment Specification Association

Deployment Specification could be associated with the **deployment** of a **component artifact** on a **node**. In this case **deployment specification** could be displayed as a classifier rectangle attached to the **deployment**. Note, **UML 2.2 specification** shows this association as a **dashed** line (while association is normally displayed as solid line.)



The ejb-jar.xml deployment specification attached to deployment.

Deployment

A **deployment** is **dependency relationship** which describes allocation of an **artifact** or **artifact instance** to a **deployment target**.

Accordingly, **deployed artifact** is an artifact or artifact instance that has been deployed to a deployment target.

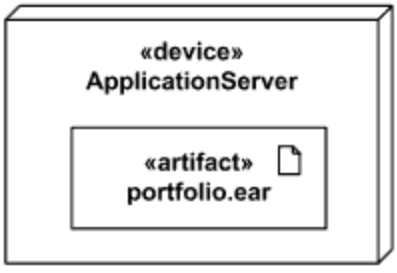
A **component deployment** is the deployment of one or more artifacts or artifact instances to a deployment target, optionally parameterized by a **deployment specification**. Examples are executables and configuration files.

The deployment relationship between a deployed artifact and a deployment target can be defined at the "type" level and at the "instance level."

For example, a 'type level' deployment relationship can be defined between an "application server" Node and an "order entry request handler" executable Artifact. At the 'instance level' 3 specific instances "appserver1" ... "app-server3" may be the deployment target for six "request handler*" instances.

For modeling complex deployment target models consisting of nodes with a composite structure defined through 'parts,' a Property (that functions as a part) may also be the target of a deployment.

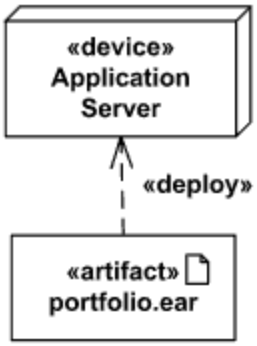
Deployment diagram shows deployed artifacts contained within a **deployment target** symbol.



The portfolio.ear artifact deployed on application server.

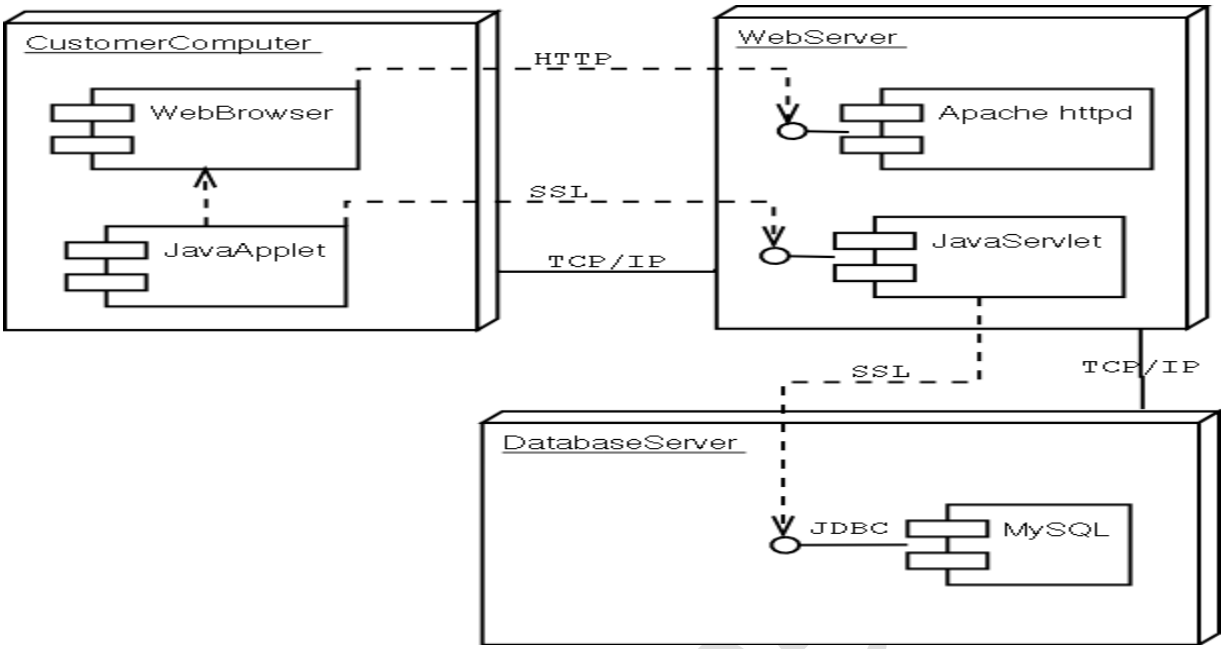
Deployment could also be shown using textual list of deployed artifacts within a **deployment target** symbol.

The portfolio.ear, stocks.ear, weather.ear artifacts deployed on application server.
An alternative notation is to use a dependency labeled «**deploy**» that is drawn from the artifact to the deployment target.



Deployment Diagram Example

- Captures the distinct number of computers involved
- Shows the communication modes employed
- Component diagrams can be embedded into deployment diagrams effectively

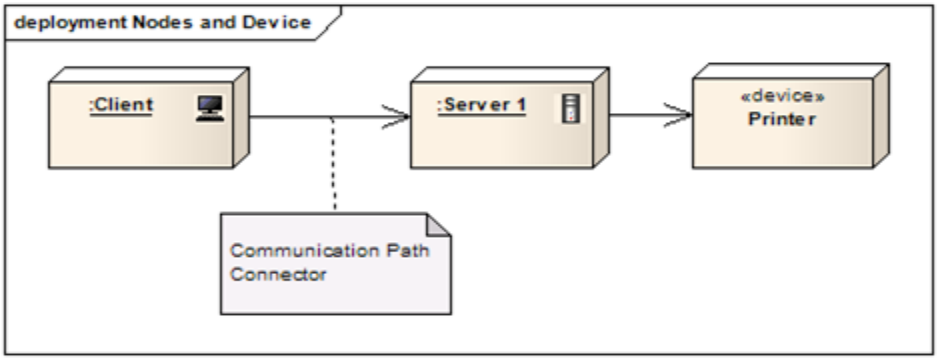


Design Codes

In this post we'll see how deployment diagrams are used to model the physical architecture of a system; we'll start from the most simple use of the deployment diagram in which we only present the nodes and their inter-relationships, and complete the picture by including the components and the applications that run in the nodes.

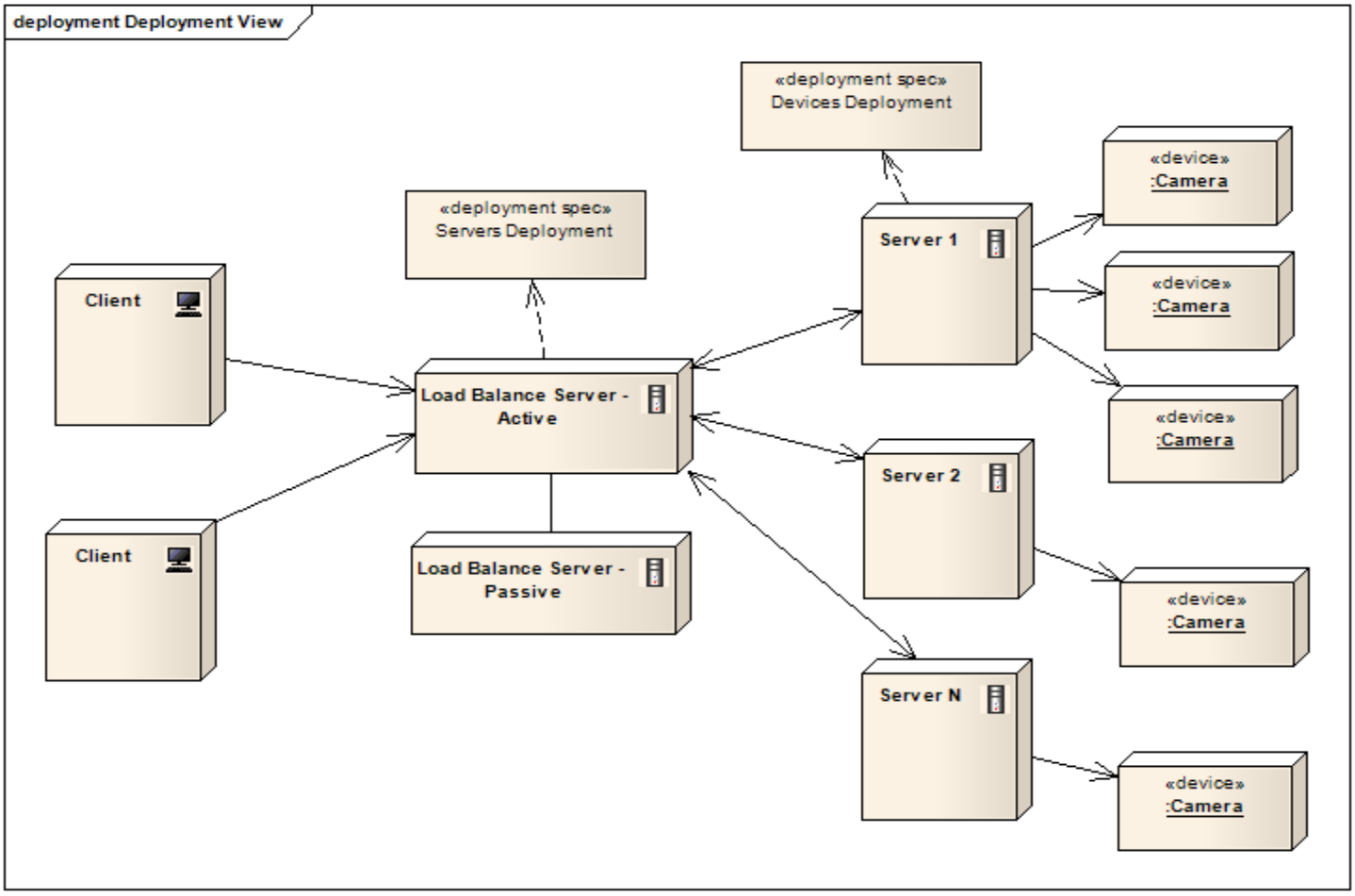
Connecting the Nodes

Very early in the system life time - deployment diagrams are used to show the nodes (computers, virtual machines) and the external devices (if there are any) which construct the system. A 'node' usually refers to a computer which can be stereotyped as server, client, workstation etc. A 'device' is a subclass of 'node' which refers to a resource with processing capability such as camera, printer, measurement instrument etc. The nodes and the devices are usually wired though the 'Communication Path' connector which illustrates the exchange of signals and messages between both ends.



Notice that the client node is stereotyped as 'pc-client' (indicated by the icon) and the server node is stereotyped as 'pc-server'.

The following diagram shows the deployment architecture of a scalable, fault tolerant 'Camera control and image processing' system . The system consist of N servers, load balancer with redundancy, and several clients.

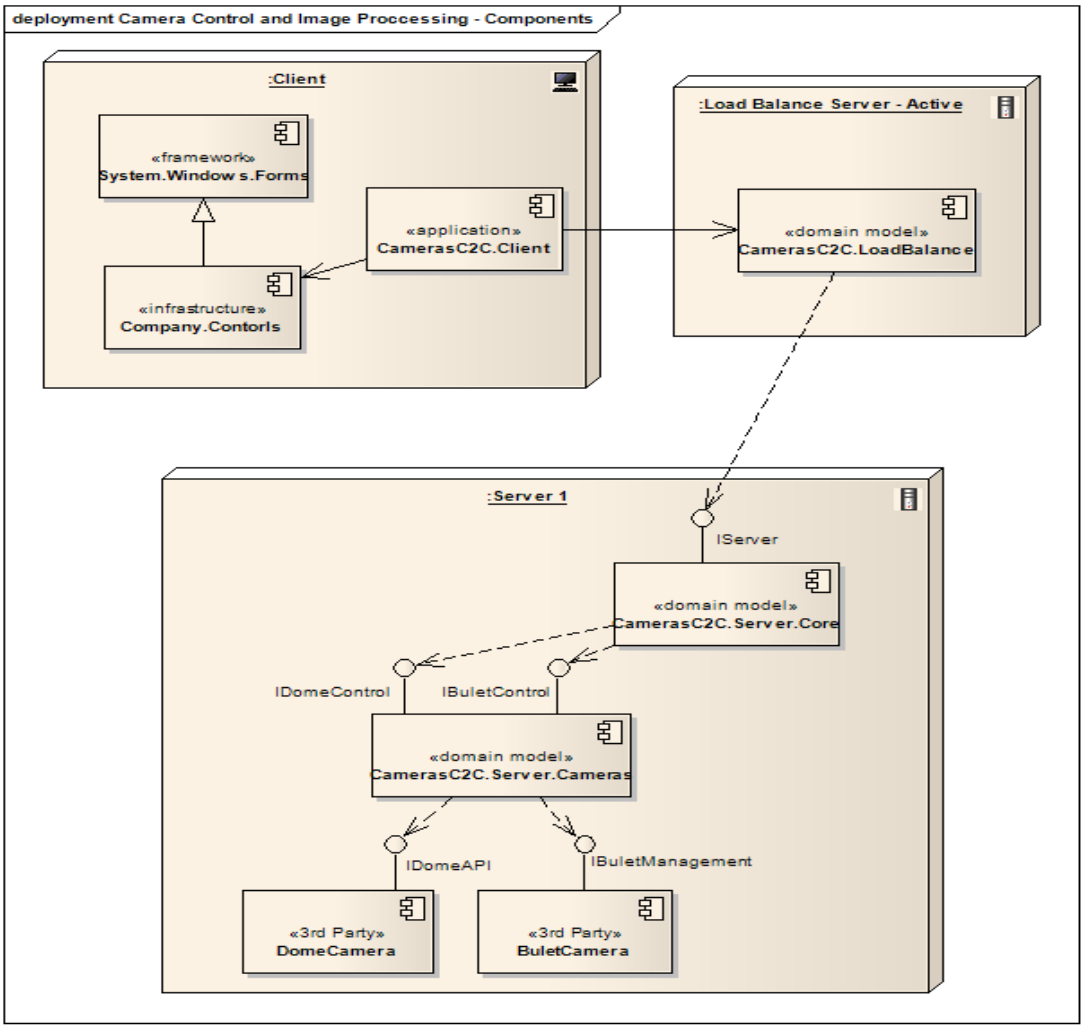


The client machines present live state of all the cameras available in the system, and allow the user to control the cameras and initiate all kind of activities on the servers. The load balancer process the inputs that it receives from the clients and send the appropriate instructions to the appropriate server, it is designed to gracefully scale to increasing number of servers. Since the load balancer is a single point of failure, a passive load balancer (that maintains copy of the active load balancer state) run in the background, ready to replace the active load balancer in case of a crush. All the servers run the same application, they support different kinds of cameras and can be configured to manage up to 200 cameras of different kinds.

Including the Components

In the next stage we are ready to put in the components that run in the physical nodes. As indicated in the previous post, when using components to model the **physical** architecture of a system (as in this case) the term 'component' refers to dll, or some executable.

The following figure shows snapshot of the above diagram with the addition of the components that reside in the nodes.

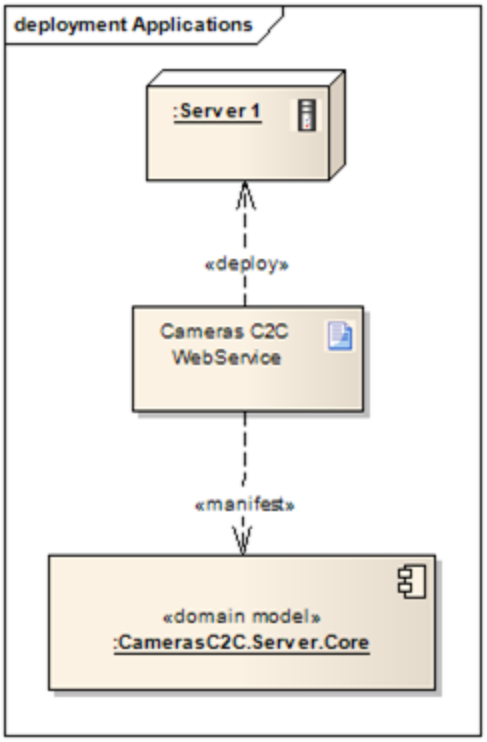


As you can see the client node includes the 'CamerasC2C.Client' component which uses infrastructure level controls reside within 'Company.Contorls' which includes classes which derive from framework level controls (notice the use of stereotypes to divide the components to levels/layers). The 'CamerasC2C.Client' component communicate with the load balancer 'CamerasC2C.LoadBalance' component, which transfer instructions to the appropriate server through the 'IServer' interface. The server consist of 3rd party components that were shipped with the cameras hardware, each component exposes interface though which the camera can be

controlled, the 'CamerasC2C.Server.Cameras' component includes adapter classes which wrap the 3rd party interfaces and expose matching interfaces that fit to the systems requirements and speak the system language (uses system level classes etc), the 'CamerasC2C.Server.Core' component uses the interfaces exposed by the 'CamerasC2C.Server.Cameras' in order to command the cameras as appropriate.

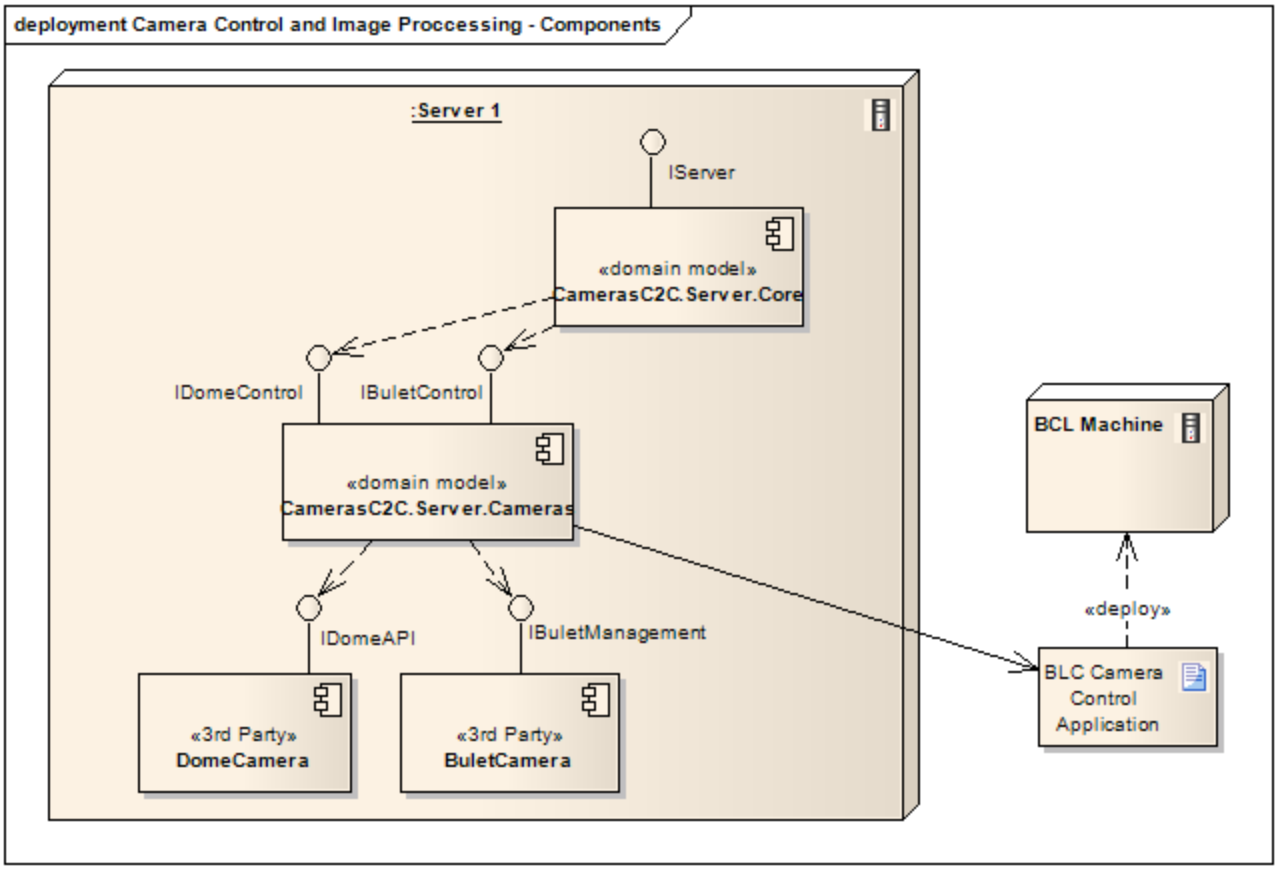
Presenting the Applications

In order to show the applications that run on the different nodes and the components that make up the applications – we use artifact wired to nodes through the 'deploy' connector, and wired to components through the 'manifest' connecter.



Presenting External Applications

In order to show the way in which the system interact with external applications - artifacts can be used to represent the external application as illustrated in the following diagrams.



UML Component Diagram:

Overview:
Component diagrams are different in terms of nature and behaviour. Component diagrams are used to model physical aspects of a system.
Now the question is what are these physical aspects? Physical aspects are the elements like executables, libraries, files, documents etc which resides in a node.
So component diagrams are used to visualize the organization and relationships among components in a system. These diagrams are also used to make executable systems.

Purpose:
Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities. So from that point component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files etc.

Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment.
A single component diagram cannot represent the entire system but a collection of diagrams are used to represent the whole.

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

- Visualize the components of a system.
- Construct executables by using forward and reverse engineering.
- Describe the organization and relationships of the components.

How to draw Component Diagram?
Component diagrams are used to describe the physical artifacts of a system. This artifact includes files, executables, libraries etc. So the purpose of this diagram is different, Component diagrams are used during the implementation phase of an application. But it is prepared well in advance to visualize the implementation details.

Initially the system is designed using different UML diagrams and then when the artifacts are ready component diagrams are used to get an idea of the implementation. This diagram is very important because without it the application cannot be implemented efficiently. A well prepared component diagram is also important for other aspects like application performance, maintenance etc.

So before drawing a component diagram the following artifacts are to be identified clearly:

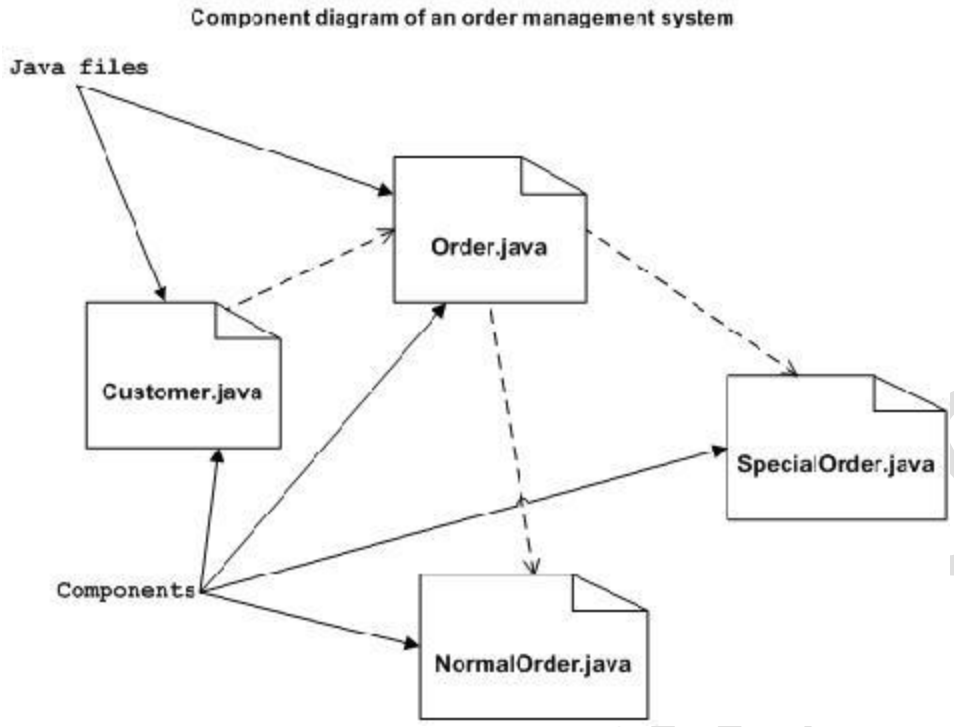
- Files used in the system.
- Libraries and other artifacts relevant to the application.
- Relationships among the artifacts.

Now after identifying the artifacts the following points needs to be followed:

- Use a meaningful name to identify the component for which the diagram is to be drawn.
- Prepare a mental layout before producing using tools.
- Use notes for clarifying important points.

The following is a component diagram for order management system. Here the artifacts are files. So the diagram shows the files in the application and their relationships. In actual the component diagram also contains dlls, libraries, folders etc. In the following diagram four files are identified and their relationships are produced. Component diagram cannot be matched directly with other UML diagrams discussed so far. Because it is drawn for completely different purpose.

So the following component diagram has been drawn considering all the points mentioned above:



Where to use Component Diagrams?
We have already described that component diagrams are used to visualize the static implementation view of a system. Component diagrams are special type of UML diagrams used for different purposes.

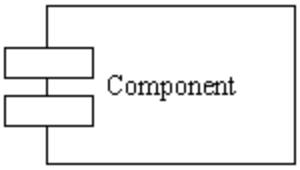
These diagrams show the physical components of a system. To clarify it, we can say that component diagrams describe the organization of the components in a system. Organization can be further described as the location of the components in a system. These components are organized in a special way to meet the system requirements.

As we have already discussed those components are libraries, files, executables etc. Now before implementing the application these components are to be organized. This component organization is also designed separately as a part of project execution.
Component diagrams are very important from implementation perspective. So the implementation team of an application should have a proper knowledge of the component details.
Now the usage of component diagrams can be described as:

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

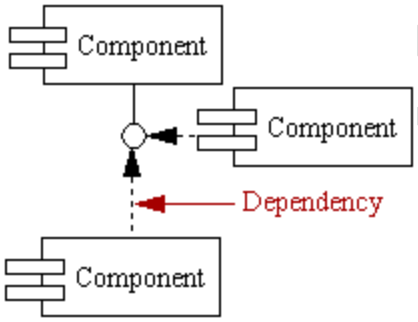
Basic Component Diagram Symbols and Notations

Component
A component is a physical building block of the system. It is represented as a rectangle with tabs.
Learn how to resize grouped objects like components.



Interface
An interface describes a group of operations used or created by components.

Dependencies
Draw dependencies among components using dashed arrows.
Learn about line styles in SmartDraw.



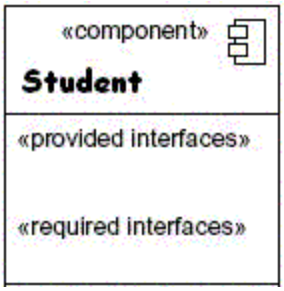
Explain all elements of a Component diagram.

Elements of a component diagram:

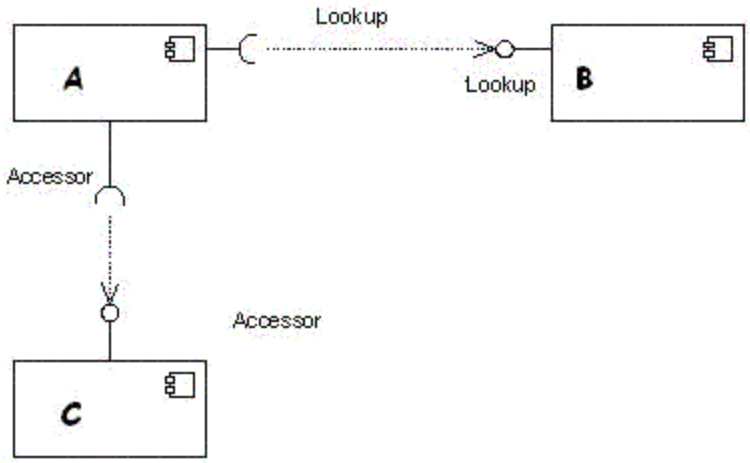
Rectangle: A single component is described using a rectangle and having the component's name inside it. <<Component>>Component Name

Additional compartments: Additional compartments are stacked below the component name.

Interfaces provided/required: Another compartment exists for displaying the interface provided and required by the component.



Relationships: A lollipop and socket notation is used along with showing dependency arrows. Dependency arrow points towards the needed socket and arrowhead connects with provider's lollipop.



Subsystem: A subsystem is represented using a rectangle with stereotype <>subsystem name.