



Cognizant
Passion for making a difference



Handout: UML

Version: UML/Handout/0408/1.0

Date: 20-04-08

Cognizant
500 Glen Pointe Center West
Teaneck, NJ 07666
Ph: 201-801-0233
www.cognizant.com

TABLE OF CONTENTS

Introduction	6
About this Module	6
Target Audience	6
Module Objectives	6
Pre-requisite	6
Session 01: History OOD Methods and Responsibility	7
Learning Objectives	7
History of Objected Oriented Design Methods	7
Responsibility Driven Vs Data Driven Approaches	9
Summary	10
Test Your Understanding	10
Session 02: Types of Classes and Object Structure	11
Learning Objectives	11
Object Structure	11
Messages and Methods	11
Types of classes	12
Relationship between classes	14
Summary	15
Test your Understanding	15
Session 03: Software Development Lifecycle	16
Learning Objectives	16
Iterative and Waterfall Processes	16
V-Model	18
Summary	18
Test your Understanding	19
Session 06: Introduction to UML	20
Learning Objectives	20
What is UML	20
UML History	20
Use of UML	22
UML Characteristics	22

UML Goals and Advantages.....	22
UML Basics and Models.....	24
Summary	24
Test your Understanding	24
Session 07: Diagrams and Notations	25
Learning Objectives.....	25
UML Diagrams.....	25
Use Case Diagrams	25
Class Diagrams	26
Object Diagrams	28
Activity Diagrams	30
Interaction Diagrams	31
State Machine Diagrams	34
Component Diagrams.....	35
Deployment Diagrams	35
UML Notations	36
Summary	39
Test your Understanding	40
Session 08: Introduction to Design Process and CASE Tools.....	41
Learning Objectives.....	41
Object Oriented Design	41
Characteristics of OOD.....	42
Process Stages.....	42
CASE Tools	42
Benefits of CASE Tools	43
Summary	43
Test your Understanding	44
Session 09: Use Case Diagram.....	45
Learning Objectives.....	45
Introduction	45
Elements of a Use Case Diagram	45
Relationships of Use Cases	47
Summary	49
Test your Understanding	49
Session 12: Class Diagram	50

Learning Objectives	50
Introduction	50
Classes	50
Types of Relationship	51
Summary	53
Test your Understanding	53
 Session 15: Sequence Diagram	54
Learning Objectives	54
What is Interaction diagram?	54
When to Use	54
How to Draw Sequence Diagrams	55
Summary	57
Test your Understanding	57
 Session 18: Activity Diagram	58
Learning Objectives	58
Activity Diagram	58
When to Use	58
Describing the Basic Notation	58
Sample Activity Diagram	59
Summary	59
Test your Understanding	60
 Session 20: Component Diagram	61
Learning Objectives	61
Introduction	61
Importance of Component Diagram	61
Elements of a Components Diagram	62
Component's Relationships	63
Summary	64
Test your Understanding	64
 Session 21: Deployment Diagram	65
Learning Objectives	65
Introduction	65
Importance of Deployment Diagram	65
Elements of a Deployment Diagram	66
Relationships in Deployment Diagram	67

Summary	68
Test your Understanding	68
Glossary	69
References	73
Websites	73
Books	73
STUDENT NOTES:	74

Introduction

About this Module

This module provides a brief overview about the following topics:

- ❑ Software development models
- ❑ Introduction to UML
- ❑ UML diagrams and notations

Target Audience

This module is designed for the entry level trainees.

Module Objectives

After completing this module, you will be able to:

- ❑ Explain the concepts of software development models and history of OO
- ❑ Explain the concepts of UML diagrams and notations
- ❑ Draw UML diagrams for a given business requirement

Pre-requisite

This module requires some basic knowledge on object oriented programming languages

Session 01: History OOD Methods and Responsibility

Learning Objectives

After completing this session, you will be able to:

- ❑ Explain the history of Objected Oriented Design Methods (OOD)
- ❑ Differentiate between Responsibility Driven and Data Driven Approaches

History of Objected Oriented Design Methods

Object-oriented design is concerned with developing an object-oriented model of a software system to implement the identified requirements. Object-oriented software design methods include refinements such as the use of design patterns, design by contract, and modeling languages (such as UML). Object Oriented Design methods have been described since the late 1980s.

The most popular Object Oriented Design methods include Booch, Buhr, Wasserman, and the HOOD method developed by the European Space Agency.

Benefits of OOD Methods:

Maintainability through simplified mapping to the problem domain, which provides for less analysis effort. Less complexity in system design, and easier verification by the user.

Reusability of the design artifacts, which saves time and costs. Productivity gains through direct mapping to features of Object-Oriented Programming Languages

Object-oriented design

Input (sources) for object-oriented design

Conceptual model (must have): Conceptual model is the result of object-oriented analysis, it captures concepts in the problem domain. The conceptual model is explicitly chosen to be independent of implementation details, such as concurrency or data storage.

Use case (must have): Use case is description of sequences of events that, taken together, lead to a system doing something useful. Each use case provides one or more scenarios that convey how the system should interact with the users called actors to achieve a specific business goal or function. Use case actors may be end users or other systems.

System Sequence Diagram (should have): System Sequence diagram (SSD) is a picture that shows, for a particular scenario of a use case, the events that external actors generate their order, and possible inter-system events.

User interface documentations (if applicable): Document that shows and describes the look and feel of the end product's user interface. This is not mandatory to have, but helps to visualize the end-product and such helps the designer.

Relational data model (if applicable): A data model is an abstract model that describes how data is represented and used. If an object database is not used, the relational data model should usually be created before the design can start. How the relational to object mapping is done is included to the OO design.

Object-oriented concepts supported by an OO language

The five basic concepts of object-oriented design are the implementation level features that are built into the programming language. These features are often referred to by these common names:

Object/Class: A tight coupling or association of data structures with the methods or functions that act on the data. This is called a class, or object (an object is created based on a class).

Information hiding: The ability to protect some components of the object from external entities. This is realized by language keywords to enable a variable to be declared as private or protected to the owning class.

Inheritance: The ability for a class to extend or override functionality of another class. The so-called sub class has a whole section that is the super class and then it has its own set of functions and data.

Interface: The ability to defer the implementation of a method. The ability to define the functions or methods signatures without implementing them.

Polymorphism: The ability to substitute an object with its sub objects. The ability of an object-variable to contain, not only that object, but also all of its sub objects as well.

Designing concepts

Defining objects, creating class diagram from conceptual diagram: Usually map entity to class.

Identifying attributes

Use design patterns (if applicable): A design pattern is not a finished design, it is a description of a solution to a common problem. The main advantage of using a design pattern is that it can be reused in multiple applications. It can also be thought of as a template for how to solve a problem that can be used in many different situations and/or applications. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

Define application framework (if applicable): Application framework is a term usually used to refer to a set of libraries or classes that are used to implement the standard structure of an application for a specific operating system. By bundling a large amount of reusable code into a framework, much time is saved for the developer, since he/she is saved the task of rewriting large amounts of standard code for each new application that is developed.

Identify persisted objects/data (if applicable): Identify objects that have to be persisted. If relational database is used design the object relation mapping.

Identify, define remote objects (if applicable)

Output (deliverables) of object-oriented design:

Class diagram: A class diagram is a type of static structure UML diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes.

Sequence diagram: Extends the System Sequence Diagram to add specific objects that handle the system events. These are usually created for important and complex system events, not for simple or trivial ones.

A sequence diagram shows, as parallel vertical lines, different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur.

Programming concepts

Aspect-oriented programming: One view of aspect-oriented programming (AOP) is that every major feature of the program, core concern (business logic), or cross-cutting concern (additional features), is an aspect, and by weaving them together (also called composition), you finally produce a whole out of the separate aspects.

Dependency injection: The basic idea is that if an object depends upon having an instance of some other object then the needed object is "injected" into the dependent object. For example, being passed a database connection as an argument to the constructor instead of creating one internally

Acyclic Dependencies Principle: The dependency graph of packages or components should have no cycles. This is also referred to as having a Directed Acyclic Graph. For example, package C depends on package B, which depends on package A. If package A also depended on package C, then you would have a cycle.

Composite Reuse Principle: Favor polymorphic composition of objects over inheritance.

Responsibility Driven Vs Data Driven Approaches

Responsibility-Driven Design Principles

Maximize Abstraction: Initially hide the distinction between data and behavior. Think of objects responsibilities for "knowing", "doing", and "deciding"

Distribute Behavior: Promote delegated control architecture. Make objects smart— have them behave intelligently, not just hold bundles of data.

Preserve Flexibility: Design objects so interior details can be readily changed

Data Driven Approaches

Used to solve specific problems in a particular program or application and often supplement the physical process model described using structured design techniques.

Recommended Use

Data driven design is used for systems with:

- ❑ A well defined, hierarchical structure of information.
- ❑ Heavy input/output flow.
- ❑ Little processing logic (For example, a library catalogue system).

Summary

- ❑ Object-oriented design is concerned with developing an object-oriented model of a software system to implement the identified requirements.
- ❑ Less complexity in system design, and easier verification by the user using OOD methods.

Test Your Understanding

1. What Is Object-Oriented Design?
2. What is Responsibility-driven design approach?

Session 02: Types of Classes and Object Structure

Learning Objectives

After completing this session, you will be able to:

- ☐ Describe the object structure
- ☐ Define Messages and Methods
- ☐ List the types of classes
- ☐ Explain the relationship between classes

Object Structure

An object has associated with it: A set of variables that contain the data for the object. The value of each variable is itself an object. A set of messages to which the object responds each message may have zero, one, or more parameters.

A set of methods, each of which is a body of code to implement a message, a method returns a value as the response to the message.

The physical representation of data is visible only to the implementer of the object. Messages and responses provide the only external interface to an object.

The term message does not necessarily imply physical message passing. Messages can be implemented as procedure invocations.

Messages and Methods

Methods are programs written in general-purpose language with the following features

- ☐ Only variables in the object itself may be referenced directly
- ☐ Data in other objects are referenced only by sending messages

Methods can be read-only or update methods: Read-only methods do not change the value of the object

Strictly speaking, every attribute of an entity must be represented by a variable and two methods, one to read and the other to update the attribute

- ☐ For example, the attribute address is represented by a variable address and two messages get-address and set-address.
- ☐ For convenience, many object-oriented data models permit direct access to variables of other objects

Types of classes

Object Classes

Similar objects are grouped into a class; each such object is called an instance of its class

All objects in a class have the same

- ❑ Variables, with the same types
- ❑ Message interface
- ❑ Methods

Example: Group objects for people into a person class

Class Definition Example

```
class employee {
    /*Variables */
    string name;
    string address;
    date start-date;
    int salary;
    /* Messages */
    int annual-salary();
    string get-name();
    string get-address();
    int set-address (string new-address);
    int employment-length();
};
```

Methods to read and set the other variables are also needed with strict encapsulation

Nested Classes

A class defined within another class is called a nested class. Nested class is illustrated as:

```
class Outerclass
{
    .....
    class Nestedclass
    {
        .....
    }
}
```

Nested classes are divided into static classes and non-static classes. Nested classes that are declared static are called static nested class and non_static nested class are called inner class

Why Nested class used

Several compiling reasons for using nested classes:

- ☐ Logical Grouping Of classes
- ☐ Increased encapsulation
- ☐ More readable maintainable code

Static Nested class

```
class Outer Class
{
    .....
    static class Staticnestedclass
    {
        .....
    }
}
```

Static nested classes are accessed using the enclosing class name:

```
OuterClass.StaticNestedClass
```

Syntax for creating an object for the static nested class:

```
OuterClass.StaticNestedClass nestedObject = new
OuterClass.StaticNestedClass();
```

Inner class

- ☐ Static Nested classes are called as Inner Class
- ☐ An instance of Innerclass can exist only within an instance of Outerclass and has direct access to the methods and fields of its enclosing instance
- ☐ Syntax for instantiating inner class

```
OuterClass.InnerClass innerObject = outerObject.new
InnerClass();
```

Types of Inner class

Two types of inner class:

- ☐ Local inner class
- ☐ Anonymous inner class

Local inner class: Declaring inner class within the body of method is called as Local inner class

Anonymous inner class: Declaring inner class within the body of method without naming it is called as anonymous inner class

Relationship between classes

Association

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

Two types of association:

- ☐ Bi-directional
- ☐ Uni-directional

Example: "department offers courses", is an association relationship.

Aggregation:

Aggregation: Whole-part. Semantics:

A part can exist on its own. The aggregate is incomplete without the part at least in some form
Multiple aggregates may share ownership of a part.

For example, computer has a printer

Drawn with clear diamond on whole side: whole<-----part

Composition:

Composition is used for contains-a relationship. Indicates strong ownership: components live and die with composite owner. Component has only one owner. The composite has responsibility for the creation and destruction of the part. Parts may change composites but must always belong to some composite.

For example, an attribute is a form of a composite

Drawn with solid diamond on whole side

Generalization

Class diagram showing generalization between one super class and two subclasses. The Generalization relationship indicates that one of the two related classes (the subtype) is considered to be a specialized form of the other (the super type) and super type is considered as Generalization of subtype.

The UML graphical representation of a Generalization is a hollow triangle shape on the super type end of the line (or tree of lines) that connects it to one or more subtypes.

The generalization relationship is also known as the inheritance or "is a" relationship.

Summary

- ❑ Object Structure is a set of variables that contain the data for the object. The value of each variable is itself an object.
- ❑ Types of class diagram:
 - Static class
 - Nested class
 - Inner class
- ❑ Various types of relationship between classes:
 - Association
 - Composition
 - Aggregation

Test your Understanding

1. Describe the relationships between the classes?

Session 03: Software Development Lifecycle

Learning Objectives

After completing this session, you will be able to:

- ☐ Describe Waterfall Model
- ☐ Describe Iterative Model
- ☐ Describe Incremental Model
- ☐ Describe V-Model

Iterative and Waterfall Processes

One of the biggest debates about process is that between waterfall and iterative styles. The terms often get misused; particularly as iterative is seen as fashionable, while the waterfall process seems to wear plaid trousers. As a result, many projects claim to do iterative development but are really doing waterfall.

The essential difference between the two is how you break up a project into smaller chunks. If you have a project that you think will take a year, few people are comfortable telling the team to go away for a year and to come back when done. Some breakdown is needed so that people can approach the problem and track progress.

The waterfall style breaks down a project based on activity. To build software, you have to do certain activities: requirements analysis, design, coding, and testing. Our 1-year project might thus have a 2-month analysis phase, followed by a 4-month design phase, followed by a 3-month coding phase, followed by a 3-month testing phase.

The iterative style breaks down a project by subsets of functionality. You might take a year and break it into 3-month iterations. In the first iteration, you'd take a quarter of the requirements and do the complete software life cycle for that quarter: analysis, design, code, and test. At the end of the first iteration, you'd have a system that does a quarter of the needed functionality. Then you'd do a second iteration so that at the end of 6 months, you'd have a system that does half the functionality.

Of course, the above is a simplified description, but it is the essence of the difference. In practice, of course, some impurities leak into the process.

With waterfall development, there is usually some form of formal handoff between each phase, but there are often backflows. During coding, something may come up that causes you to revisit the analysis and design. You certainly should not assume that all design is finished when coding begins. It's inevitable that analysis and design decisions will have to be revisited in later phases. However, these backflows are exceptions and should be minimized as much as possible.

With iteration, you usually see some form of exploration activity before the true iterations begin. At the very least, this will get a high-level view of the requirements: at least enough to break the requirements down into the iterations that will follow. Some high-level design decisions may occur during exploration too. At the other end, although each iteration should produce production-ready integrated software, it often doesn't quite get to that point and needs a stabilization period to iron out the last bugs. Also, some activities, such as user training, are left to the end.

You may well not put the system into production at the end of each iteration, but the system should be of production quality. Often, however, you can put the system into production at regular intervals; this is good because you get value from the system earlier and you get better-quality feedback. In this situation, you often hear of a project having multiple releases, each of which is broken down into several iterations.

Iterative development has come under many names: incremental, spiral, evolutionary, and jacuzzi spring to mind. Various people make distinctions among them, but the distinctions are neither widely agreed on nor that important compared to the iterative/waterfall dichotomy.

You can have hybrid approaches. [McConnell] describes the staged delivery life cycle whereby analysis and high-level design are done first, in a waterfall style, and then the coding and testing are divided up into iterations. Such a project might have 4 months of analysis and design followed by four 2-month iterative builds of the system.

Most writers on software process in the past few years, especially in the object-oriented community, dislike the waterfall approach. Of the many reasons for this, the most fundamental is that it's very difficult to tell whether the project is truly on track with a waterfall process. It's too easy to declare victory with early phases and hide a schedule slip. Usually, the only way you can really tell whether you are on track is to produce tested, integrated software. By doing this repeatedly, an iterative style gives you better warning if something is going awry.

For that reason alone, I strongly recommend that projects do not use a pure waterfall approach. You should at least use staged delivery, if not a more pure iterative technique.

The OO community has long been in favor of iterative development, and it's safe to say that pretty much everyone involved in building the UML is in favor of at least some form of iterative development. My sense of industrial practice is that waterfall development is still the more common approach, however. One reason for this is what I refer to as pseudo iterative development: People claim to be doing iterative development but are in fact doing waterfall.

Common symptoms of this are:

- ❑ "You are doing one analysis iteration followed by two design iterations. . . .
- ❑ "This iteration's code is very buggy, but we'll clean it up at the end."

It is particularly important that each iteration produces tested, integrated code that is as close to production quality as possible. Testing and integration are the hardest activities to estimate, so it's important not to have an open-ended activity like that at the end of the project. The test should be that any iteration that's not scheduled to be released could be released without substantial extra development work.

A common technique with iterations is to use time boxing. This forces an iteration to be a fixed length of time. If it appears that you can't build all you intended to build during an iteration, you must decide to slip some functionality from the iteration; you must not slip the date of the iteration. Most projects that use iterative development use the same iteration length throughout the project; that way, you get a regular rhythm of builds.

I like time boxing because people usually have difficulty slipping functionality. By practicing slipping function regularly, they are in a better position to make an intelligent choice at a big release between slipping a date and slipping function. Slipping function during iterations is also effective at helping people learn what the real requirements priorities are.

One of the most common concerns about iterative development is the issue of rework. Iterative development explicitly assumes that you will be reworking and deleting existing code during the later iterations of a project. In many domains, such as manufacturing, rework is seen as a waste. But software isn't like manufacturing; as a result, it often is more efficient to rework existing code than to patch around code that was poorly designed. A number of technical practices can greatly help make rework be more efficient.

V-Model

The V-model is a graphical representation of the systems development lifecycle. It summarizes the main steps to be taken in conjunction with the corresponding deliverables within computerized system validation framework.

The VEE is a process that represents the sequence of steps in a project life cycle development. It describes the activities and results that have to be produced during product. The left side of the VEE represents the decomposition of requirements, and creation of system specifications. The right side of the V represents integration of parts and their verification

- ❑ The V-model deploys a well-structured method in which each phase can be implemented by the detailed documentation of the previous phase. Testing activities like test designing start at the beginning of the project well before coding and therefore saves a huge amount of the project time.
- ❑ The V-model can be said to have developed as a result of the evolution of software testing. Various testing techniques were defined and various kinds of testing were clearly separated from each other which led to the waterfall model evolving into the V-model. The tests in the ascending (Validation) hand are derived directly from their design or requirements counterparts in the descending (Verification) hand. The 'V' can also stand for the terms Verification and Validation.

Summary

- ❑ In a waterfall model, each phase must be completed in its entirety before the next phase can begin.
- ❑ Iterative and Incremental development is a cyclical software development process developed in response to the weaknesses of the waterfall model.
- ❑ The V-model deploys a well-structured method in which each phase can be implemented by the detailed documentation of the previous phase. Testing activities like test designing start at the beginning of the project well before coding and therefore saves a huge amount of the project time.

Test your Understanding

1. What kind of software life cycle model can be used when requirements are not clear in the requirement analysis phase and chances are more that business owners/customers will come up with revised requirements during the development?

Session 06: Introduction to UML

Learning Objectives

After completing this session, you will be able to:

- ☐ Define UML
- ☐ Explain the history of UML
- ☐ List the use of UML
- ☐ List the characteristics of UML
- ☐ List the UML goals and advantages
- ☐ Describe UML models

What is UML

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

UML is the industry standard OO visual modeling 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, behavior of significant parts of system and their relationships.

The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. The UML is a very important part of developing objects oriented software and the software development process. The UML uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

UML History

Identifiable object-oriented modeling languages began to appear between mid-1970 and the late 1980s as various methodologists experimented with different approaches to object-oriented analysis and design. The number of identified modeling languages increased from less than 10 to more than 50 during the period between 1989-1994. Many users of OO methods had trouble finding complete satisfaction in any one modeling language, fueling the "method wars." By the mid-1990s, new iterations of these methods began to appear and these methods began to incorporate each other's techniques, and a few clearly prominent methods emerged.¹

The development of UML began in late 1994 when Grady Booch and Jim Rumbaugh of Rational Software Corporation began their work on unifying the Booch and OMT (Object Modeling Technique) methods. In the fall of 1995, Ivar Jacobson and his Objectory Company joined Rational and this unification effort, merging in the OOSE (Object-Oriented Software Engineering) method.¹

As the primary authors of the Booch, OMT, and OOSE methods, Grady Booch, Jim Rumbaugh, and Ivar Jacobson were motivated to create a unified modeling language for three reasons. First, these methods were already evolving toward each other independently. It made sense to continue that evolution together rather than apart, eliminating the potential for any unnecessary and gratuitous differences that would further confuse users. Second, by unifying the semantics and notation, they could bring some stability to the object-oriented marketplace, allowing projects to settle on one mature modeling language and letting tool builders focus on delivering more useful features. Third, they expected that their collaboration would yield improvements in all three earlier methods, helping them to capture lessons learned and to address problems that none of their methods previously handled well.

The efforts of Booch, Rumbaugh, and Jacobson resulted in the release of the UML 0.9 and 0.91 documents in June and October of 1996. During 1996, the UML authors invited and received feedback from the general community. They incorporated this feedback, but it was clear that additional focused attention was still required.

While Rational was bringing UML together, efforts were being made on achieving the broader goal of an industry standard modeling language. In early 1995, Ivar Jacobson (then Chief Technology Officer of Objectory) and Richard Soley (then Chief Technology Officer of OMG) decided to push harder to achieve standardization in the methods marketplace. In June 1995, an OMG-hosted meeting of all major methodologists (or their representatives) resulted in the first worldwide agreement to seek methodology standards, under the aegis of the OMG process.¹

During 1996, it became clear that several organizations saw UML as strategic to their business. A Request for Proposal (RFP) issued by the Object Management Group (OMG) provided the catalyst for these organizations to join forces around producing a joint RFP response. Rational established the UML Partners consortium with several organizations willing to dedicate resources to work toward a strong UML 1.0 definition. Those contributing most to the UML 1.0 definition included: Digital Equipment Corp., HP, i-Logix, IntelliCorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI, and Unisys. This collaboration produced UML 1.0, a modeling language that was well defined, expressive, powerful, and generally applicable. This was submitted to the OMG in January 1997 as an initial RFP response.

In January 1997 IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies and Softeam also submitted separate RFP responses to the OMG. These companies joined the UML partners to contribute their ideas, and together the partners produced the revised UML 1.1 response. The focus of the UML 1.1 release was to improve the clarity of the UML 1.0 semantics and to incorporate contributions from the new partners. It was submitted to the OMG for their consideration and adopted in the fall of 1997

Use of UML

As the strategic value of software increases for many companies, the industry looks for techniques to automate the production of software and to improve quality and reduce cost and time-to-market. These techniques include component technology, visual programming, patterns and frameworks. Businesses also seek techniques to manage the complexity of systems as they increase in scope and scale. In particular, they recognize the need to solve recurring architectural problems, such as physical distribution, concurrency, replication, security, load balancing and fault tolerance. Additionally, the development for the World Wide Web, while making some things simpler, has exacerbated these architectural problems. The Unified Modeling Language (UML) was designed to respond to these needs.

UML Characteristics

The followings are the characteristics of the UML:

- ☐ UML is independent of the implementation language.
- ☐ UML could be implemented in various languages
- ☐ Large, useful set of predefined constructs
- ☐ Extensible
- ☐ Semi-formal definition of syntax and semantics
- ☐ Potential for wide adoption, standardization and substantial tool support
- ☐ Basis in experience with mainstream development methods

UML Goals and Advantages

The primary goals in the design of the UML were as follows:

- ☐ Provide users a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
- ☐ Provide extensibility and specialization mechanisms to extend the core concepts.
- ☐ Be independent of particular programming languages and development processes.
- ☐ Provide a formal basis for understanding the modelling language.
- ☐ Encourage the growth of the OO tools market.
- ☐ Support higher-level development concepts such as collaborations, frameworks, patterns, and components.
- ☐ Integrate best practices.

These goals are discussed, as follows.

Provide users a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models. It is important that the OOAD standard support a modeling language that can be used “out of the box” to do normal general-purpose modeling tasks. If the standard merely provides a meta-meta-description that requires tailoring to a particular set of modeling concepts, then it will not achieve the purpose of allowing users to exchange models without losing information or without imposing excessive work to map their models to a very abstract form. The UML consolidates a set of core modeling concepts that are generally accepted across many current methods and modeling tools. These concepts are needed in many or most large applications, although not every concept is needed in every part of every application. Specifying a meta-meta-level format for the concepts is not sufficient for model users, because the

concepts must be made concrete for real modeling to occur. If the concepts in different application areas were substantially different, then such an approach might work, but the core concepts needed by most application areas are similar and should therefore be supported directly by the standard without the need for another layer.

Provide extensibility and specialization mechanisms to extend the core concepts. We expect that the UML will be tailored as new needs are discovered and for specific domains. At the same time, we do not want to force the common core concepts to be redefined or re-implemented for each tailored area. Therefore we believe that the extension mechanisms should support deviations from the common case, rather than being required to implement the core OOA&D concepts themselves. The core concepts should not be changed more than necessary. Users need to be able to 1) build models using core concepts without using extension mechanisms for most normal applications; 2) add new concepts and notations for issues not covered by the core; 3) choose among variant interpretations of existing concepts, when there is no clear consensus; and 4) specialize the concepts, notations, and constraints for particular application domains.

Be independent of particular programming languages and development processes. The UML must and can support all reasonable programming languages. It also must and can support various methods and processes of building models. The UML can support multiple programming languages and development methods without excessive difficulty.

Provide a formal basis for understanding the modeling language. Because users will use formality to help understand the language, it must be both precise and approachable; a lack of either dimension damages its usefulness. The formalisms must not require excessive levels of indirection or layering, use of low-level mathematical notations distant from the modeling domain, such as set-theoretic notation, or operational definitions that are equivalent to programming an implementation. The UML provides a formal definition of the static format of the model using a metamodel expressed in UML class diagrams. This is a popular and widely accepted formal approach for specifying the format of a model and directly leads to the implementation of interchange formats. UML expresses well-formed constraints in precise natural language plus Object Constraint Language expressions. UML expresses the operational meaning of most constructs in precise natural language. The fully formal approach taken to specify languages such as Algol-68 was not approachable enough for most practical usage.

Encourage the growth of the OO tools market. By enabling vendors to support a standard modeling language used by most users and tools, the industry benefits. While vendors still can add value in their tool implementations, enabling interoperability is essential. Interoperability requires that models can be exchanged among users and tools without loss of information. This can only occur if the tools agree on the format and meaning of all of the relevant concepts. Using a higher meta-level is no solution unless the mapping to the user-level concepts is included in the standard.

Support higher-level development concepts such as collaborations, frameworks, patterns, and components. Clearly defined semantics of these concepts is essential to reap the full benefit of OO and reuse. Defining these within the holistic context of a modeling language is a unique contribution of the UML.

Integrate best practices. A key motivation behind the development of the UML has been to integrate the best practices in the industry, encompassing widely varying views based on levels of abstraction, domains, architectures, life cycle stages, implementation technologies, etc. The UML is indeed such an integration of best practices.

UML Basics and Models

The Unified Modeling Language (UML) is a diagramming language or notation to specify, visualize and document models of Object Orientated software systems. UML is not a development method that means it does not tell you what to do first and what to do next or how to design your system, but it helps you to visualize your design and communicate with others. UML is controlled by the Object Management Group (OMG) and is the industry standard for graphically describing software.

UML is designed for Object Orientated software design and has limited use for other programming paradigms.

UML is composed of many model elements that represent the different parts of a software system. The UML elements are used to create diagrams, which represent a certain part, or a point of view of the system.

Use Case Diagrams show actors (people or other users of the system), use cases (the scenarios when they use the system), and their relationships

Class Diagrams show classes and the relationships between them

Sequence Diagrams show objects and a sequence of method calls they make to other objects.

Collaboration Diagrams show objects and their relationship, putting emphasis on the objects that participate in the message exchange

State Diagrams show states, state changes and events in an object or a part of the system

Activity Diagrams show activities and the changes from one activity to another with the events occurring in some part of the system

Component Diagrams show the high level programming components (such as Java Beans).

Deployment Diagrams show the instances of the components and their relationships.

Summary

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

Test your Understanding

1. What is UML?
2. What are diagrams?

Session 07: Diagrams and Notations

Learning Objectives

After completing this session, you will be able to:

- ☐ Draw UML Diagrams
- ☐ Work with UML notations

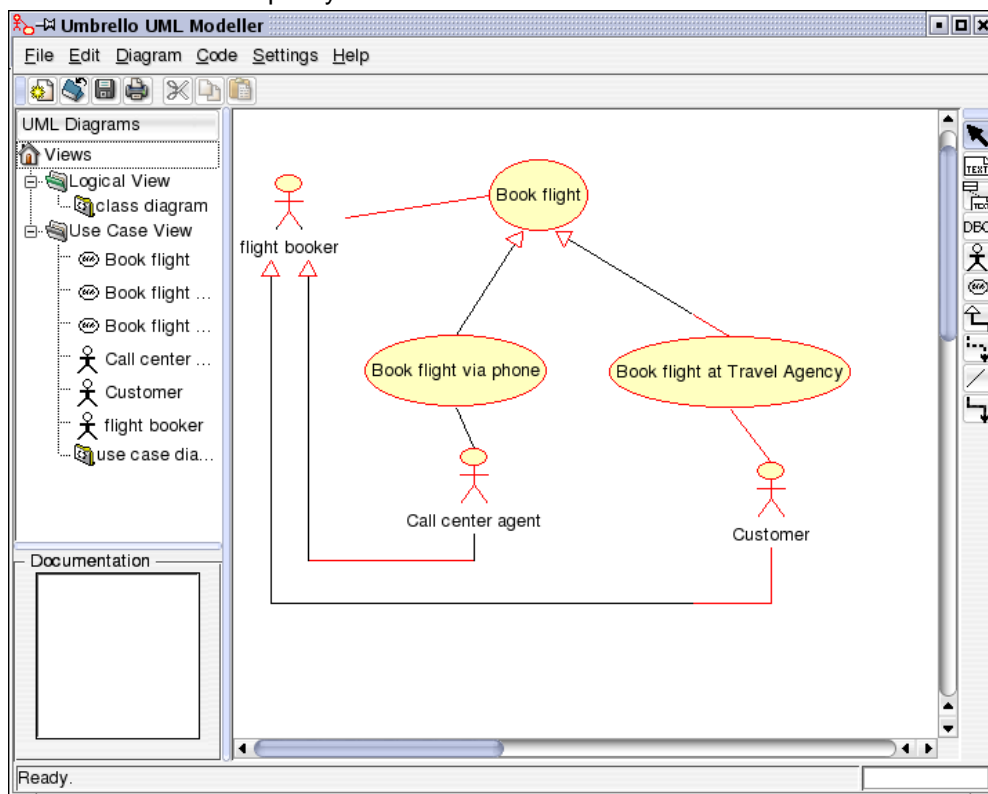
UML Diagrams

Each UML diagram is designed to let developers and customers view a software system from a different perspective and in varying degrees of abstraction. UML diagrams commonly created in visual modeling tools. The followings are different type of UML diagrams.

Use Case Diagrams

Use Case Diagrams describe the relationships and dependencies between a group of Use Cases and the Actors participating in the process.

It is important to notice that Use Case Diagrams are not suited to represent the design, and cannot describe the internals of a system. Use Case Diagrams are meant to facilitate the communication with the future users of the system, and with the customer, and are specially helpful to determine the required features the system is to have. Use Case Diagrams tell, what the system should do but do not — and cannot — specify how this is to be achieved.



Use Case

A Use Case describes — from the point of view of the actors — a group of activities in a system that produces a concrete, tangible result. Use Cases are descriptions of the typical interactions between the users of a system and the system itself. They represent the external interface of the system and specify a form of requirements of what the system has to do (remember, only what, not how).

When working with Use Cases, it is important to remember some simple rules:

- ❑ Each Use Case is related to at least one actor
- ❑ Each Use Case has an initiator (i.e. an actor)
- ❑ Each Use Case leads to a relevant result (a result with “business value”)

Use Cases can also have relationships with other Use Cases. The three most typical types of relationships between Use Cases are:

- ❑ <<include>> which specifies that a Use Case takes place inside another Use Case
- ❑ <<extends>> which specifies that in certain situations or at some point (called an extension point) a Use Case will be extended by another.
- ❑ Generalization specifies that a Use Case inherits the characteristics of the “Super”-Use Case, and can override some of them or add new ones in a similar way as the inheritance between classes.

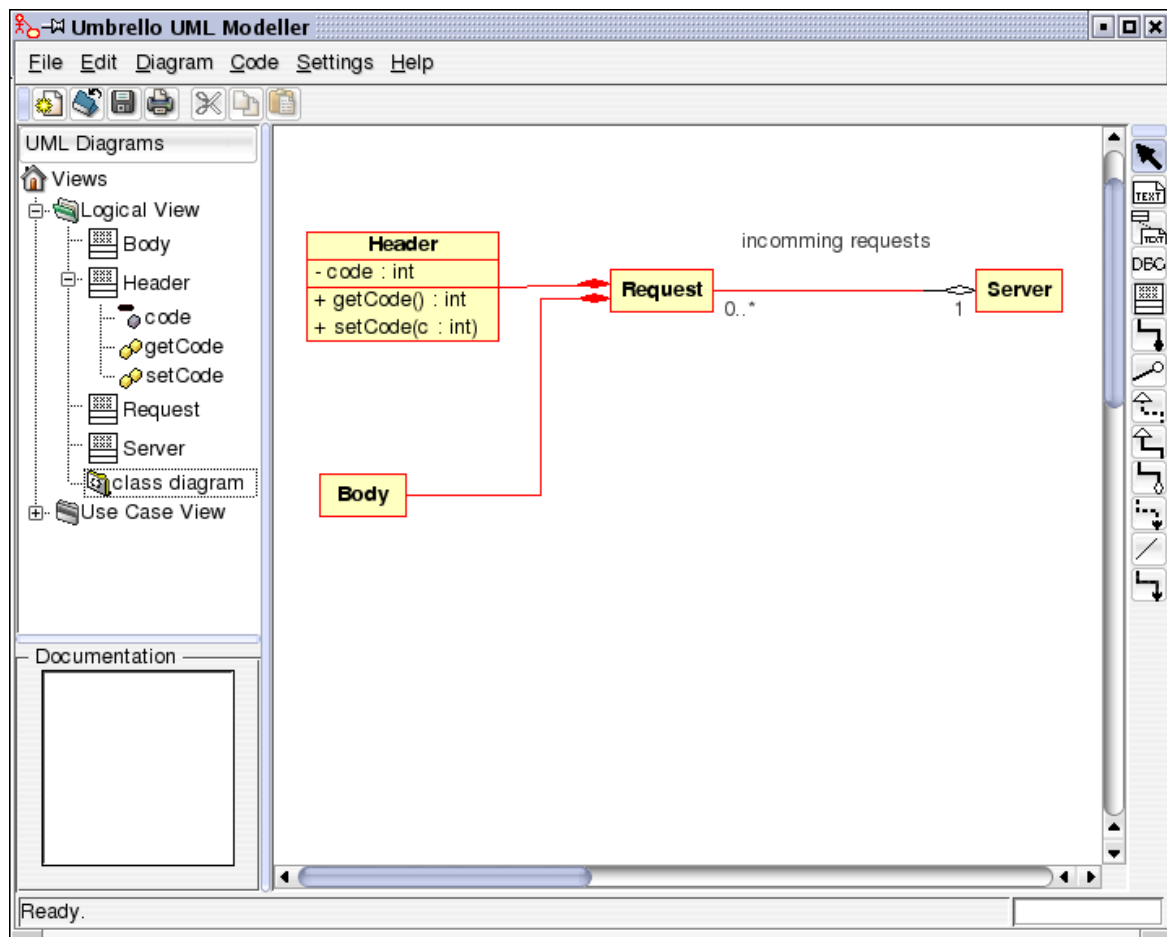
Actor

An actor is an external entity (outside of the system) that interacts with the system by participating (and often initiating) a Use Case. Actors can be in real life people (for example users of the system), other computer systems or external events.

Actors do not represent the physical people or systems, but their role. This means that when a person interacts with the system in different ways (assuming different roles) he will be represented by several actors. For example a person that gives customer support by the telephone and takes orders from the customer into the system would be represented by an actor “Support Staff” and an actor “Sales Representative”

Class Diagrams

Class diagrams show the different classes that make up a system and how they relate to each other. Class diagrams are said to be “static” diagrams because they show the classes, along with their methods and attributes as well as the static relationships between them: which classes “know” about which classes or which classes “are part” of another class, but do not show the method calls between them.



Class

A class defines the attributes and the methods of a set of objects. All objects of this class (instances of this class) share the same behaviour, and have the same set of attributes (each object has its own set). The term “Type” is sometimes used instead of Class, but it is important to mention that these two are not the same, and Type is a more general term.

In UML, classes are represented by rectangles, with the name of the class, and can also show the attributes and operations of the class in two other “compartments” inside the rectangle

Attributes

In UML, attributes are shown with at least their name, and can also show their type, initial value and other properties. Attributes can also be displayed with their visibility:

- ☐ + Stands for public attributes
- ☐ # Stands for protected attributes
- ☐ - Stands for private attributes

Operations

Operations (methods) are also displayed with at least their name, and can also show their parameters and return types. Operations can, just as Attributes, display their visibility:

- ☐ + Stands for public operations
- ☐ # Stands for protected operations

- - Stands for private operations

Templates

Classes can have templates, a value which is used for an unspecified class or type. The template type is specified when a class is initiated (i.e. an object is created). Templates exist in modern C++ and will be introduced in Java 1.5 where they will be called Generics

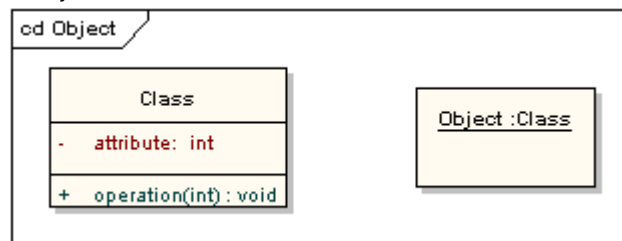
Object Diagrams

A pictorial representation of the relationships between these instantiated classes at any point of time (called objects) is called an "Object diagram." It looks very similar to a class diagram, and uses the similar notations to denote relationships.

An object diagram may be considered a special case of a class diagram. Object diagrams use a subset of the elements of a class diagram in order to emphasize the relationship between instances of classes at some point in time. They are useful in understanding class diagrams. They don't show anything architecturally different to class diagrams, but reflect multiplicity and roles.

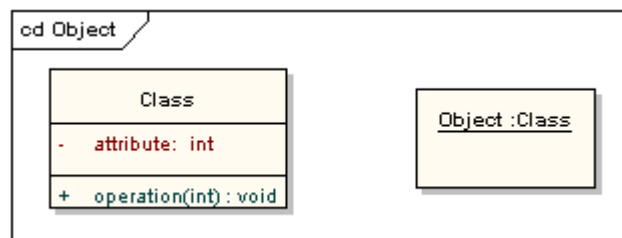
Class and Object Elements

The following diagram shows the differences in appearance between a class element and an object element. Note that the class element consists of three parts, being divided into name, attribute and operation compartments; by default, object elements don't have compartments. The display of names is also different: object names are underlined and may show the name of the classifier from which the object is instantiated.



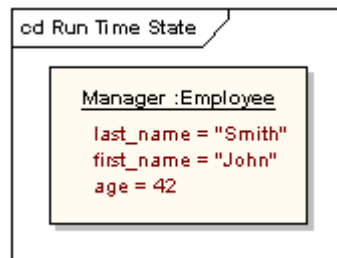
Class and Object Elements

The following diagram shows the differences in appearance between a class element and an object element. Note that the class element consists of three parts, being divided into name, attribute and operation compartments; by default, object elements don't have compartments. The display of names is also different: object names are underlined and may show the name of the classifier from which the object is instantiated.



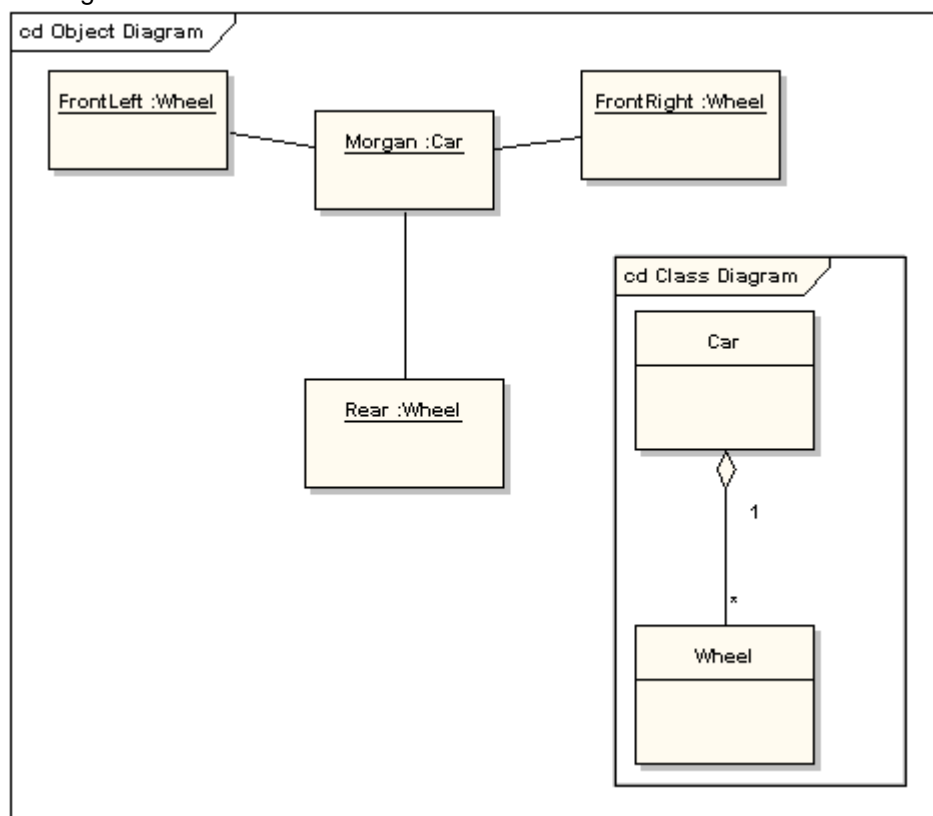
Run Time State

A classifier element can have any number of attributes and operations. These aren't shown in an object instance. It is possible, however, to define an object's run time state, showing the set values of attributes in the particular instance.



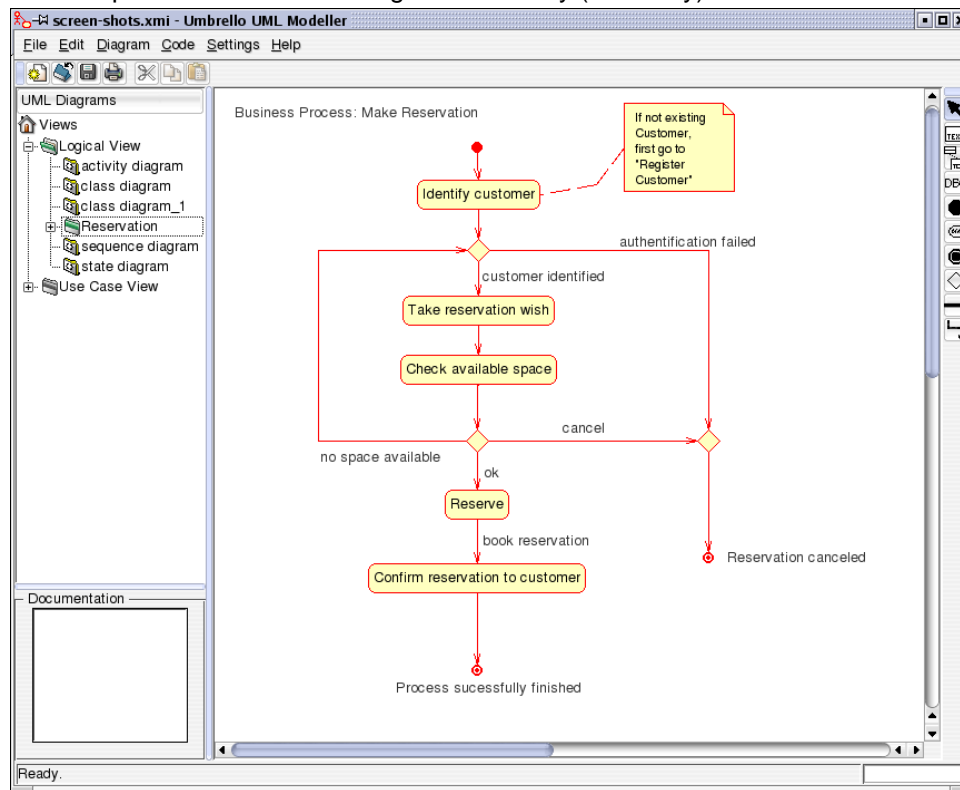
Example Class and Object Diagrams

The following diagram shows an object diagram with its defining class diagram inset, and it illustrates the way in which an object diagram may be used to test the multiplicities of assignments in class diagrams. The car class has a 1-to-many multiplicity to the wheel class, but if a 1-to-4 multiplicity had been chosen instead, that wouldn't have allowed for the three-wheeled car shown in the object diagram.



Activity Diagrams

Activity Diagrams describe the sequence of activities in a system with the help of Activities. Activity Diagrams are a special form of State Diagrams that only (or mostly) contains Activities.



Activity Diagrams are similar to procedural Flux Diagrams, with the difference that all Activities are clearly attached to Objects. Activity Diagrams are always associated to a Class, an Operation or a Use Case.

Activity Diagrams support sequential as well as parallel Activities. Parallel execution is represented via Fork/Wait icons, and for the Activities running in parallel, it is not important the order in which they are carried out (they can be executed at the same time or one after the other)

Activity

An Activity is a single step in a process. One Activity is one state in the system with internal activity and, at least, one outgoing transition. Activities can also have more than one outgoing transition if they have different conditions.

Activities can form hierarchies, this means that an Activity can be composed of several "detail" Activities, in which case the incoming and outgoing transitions should match the incoming and outgoing transitions of the detail diagram.

Helper Elements

There are a few elements in UML that have no real semantic value for the model, but help to clarify parts of the diagram. These elements are

- ❑ Text lines
- ❑ Text Notes and anchors
- ❑ Boxes

Text lines are useful to add short text information to a diagram. It is free-standing text and has no meaning to the Model itself.

Notes are useful to add more detailed information about an object or a specific situation. They have the great advantage that notes can be anchored to UML Elements to show that the note "belongs" to a specific object or situation.

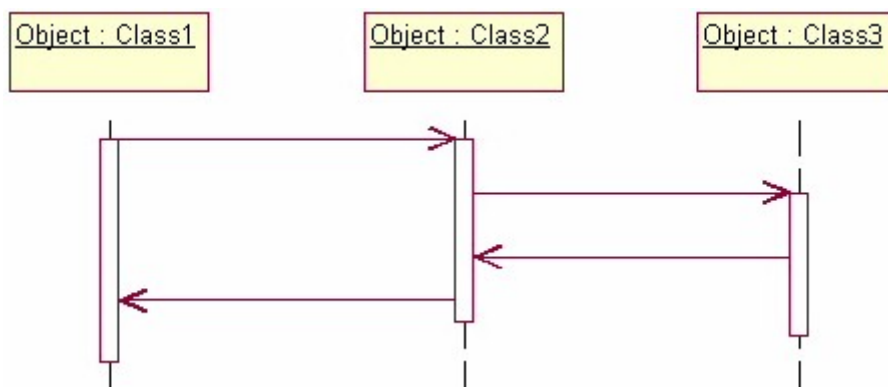
Boxes are free-standing rectangles which can be used to group items together to make diagrams more readable. They have no logical meaning in the model


Interaction Diagrams

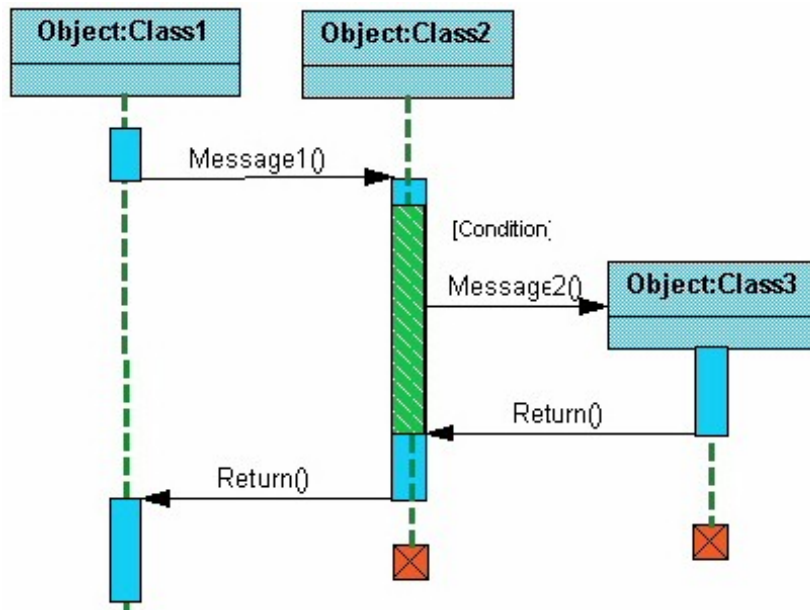
Sequence diagrams, collaboration diagrams, or both diagrams can be used to demonstrate the interaction of objects in a use case. Sequence diagrams generally show the sequence of events that occur. Collaboration diagrams demonstrate how objects are statically connected. Both diagrams are relatively simple to draw and contain similar elements.

Sequence diagrams:

Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass. The diagrams are read left to right and descending. The example below shows an object of class 1 start the behavior by sending a message to an object of class 2. Messages pass between the different objects until the object of class 1 receives the final message.

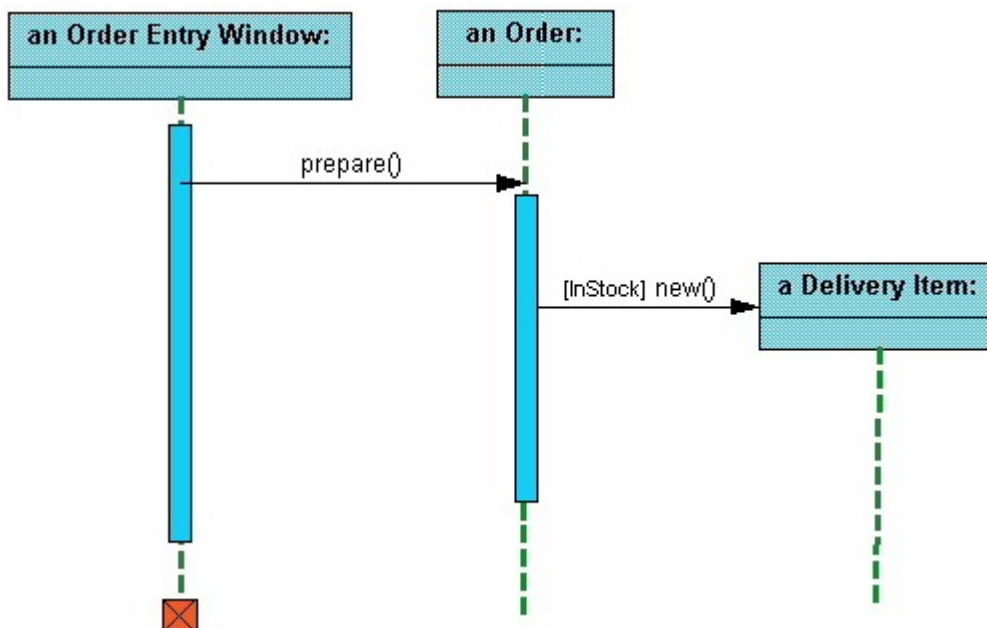


Below is a slightly more complex example. The light blue vertical rectangles represent the objects' activation while the green vertical dashed lines represent the life of the object. The green vertical rectangles represent when a particular object has control. The  represents when the object is destroyed. This diagram also shows conditions for messages to be sent to other objects. The condition is listed between brackets next to the message. For example, a [condition] has to be met before the object of class 2 can send a message() to the object of class 3.

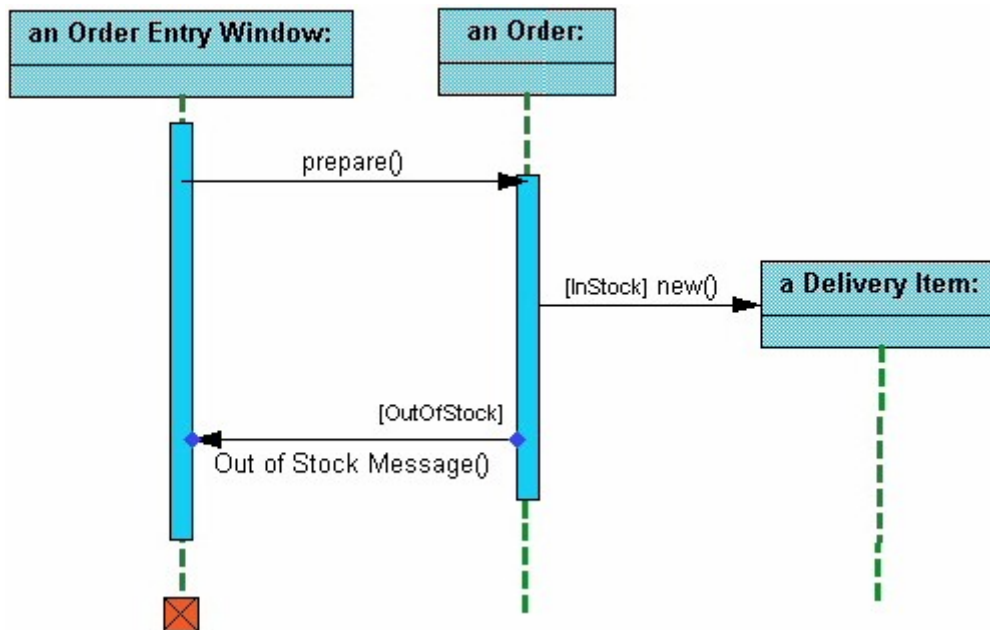


The next diagram shows the beginning of a sequence diagram for placing an order. The object an Order Entry Window is created and sends a message to an Order object to prepare the order. Notice the the names of the objects are followed by a colon. The names of the classes the objects belong to do not have to be listed. However the colon is required to denote that it is the name of an object following the `objectName:className` naming system.

Next the Order object checks to see if the item is in stock and if the [InStock] condition is met it sends a message to create a new Delivery Item object.



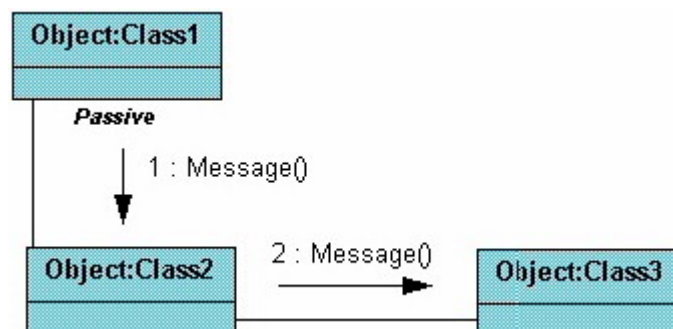
The next diagrams add another conditional message to the Order object. If the item is [OutOfStock] it sends a message back to the Order Entry Window object stating that the object is out of stock.



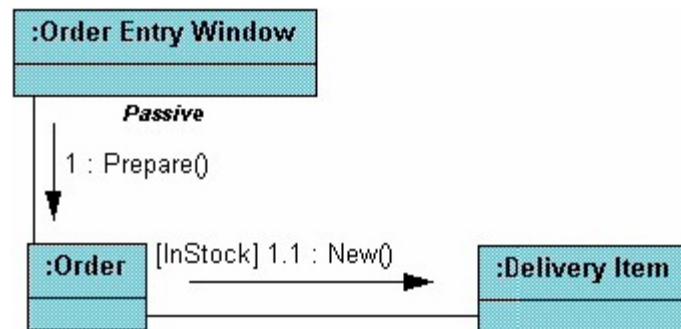
This simple diagram shows the sequence that messages are passed between objects to complete a use case for ordering an item.

Collaboration diagrams:

Collaboration diagrams are also relatively easy to draw. They show the relationship between objects and the order of messages passed between them. The objects are listed as icons and arrows indicate the messages being passed between them. The numbers next to the messages are called sequence numbers. As the name suggests, they show the sequence of the messages as they are passed between the objects. There are many acceptable sequence numbering schemes in UML. A simple 1, 2, 3... format can be used, as the example below shows, or for more detailed and complex diagrams a 1, 1.1, 1.2, 1.2.1... Scheme can be used.



The example below shows a simple collaboration diagram for the placing an order use case. This time the names of the objects appear after the colon, such as :Order Entry Window following the objectName:className naming convention. This time the class name is shown to demonstrate that all of objects of that class will behave the same way.



State Machine Diagrams

State Diagrams show the different states of an Object during its life and the stimuli that cause the Object to change its state.

State Diagrams exhibit Objects as state machines or finite automata that can be in one of a finite set of states and that can change its state via one of a finite set of stimuli. For example an Object of type NetServer can be in one of following states during its life:

- ☐ Ready
- ☐ Listening
- ☐ Working
- ☐ Stopped

And the events that can cause the object to change states are as follows:

- ☐ Object is created
- ☐ Object receives message listen
- ☐ A Client requests a connection over the network
- ☐ A Client terminates a request
- ☐ The request is executed and terminated
- ☐ Object receives message stop

State

States are the building block of State Diagrams. A State belongs to exactly one class and represents a summary of the values the attributes of a class can take. A UML State describes the internal state of an object of one particular class.

Note that not every change in one of the attributes of an object should be represented by a State but only those changes that can significantly affect the workings of the object.

There are two special types of States: Start and End. They are special in that there is no event that can cause an Object to return to its Start state, in the same way as there is no event that can possibly take an Object out of its End state once it has reached it.

Component Diagrams

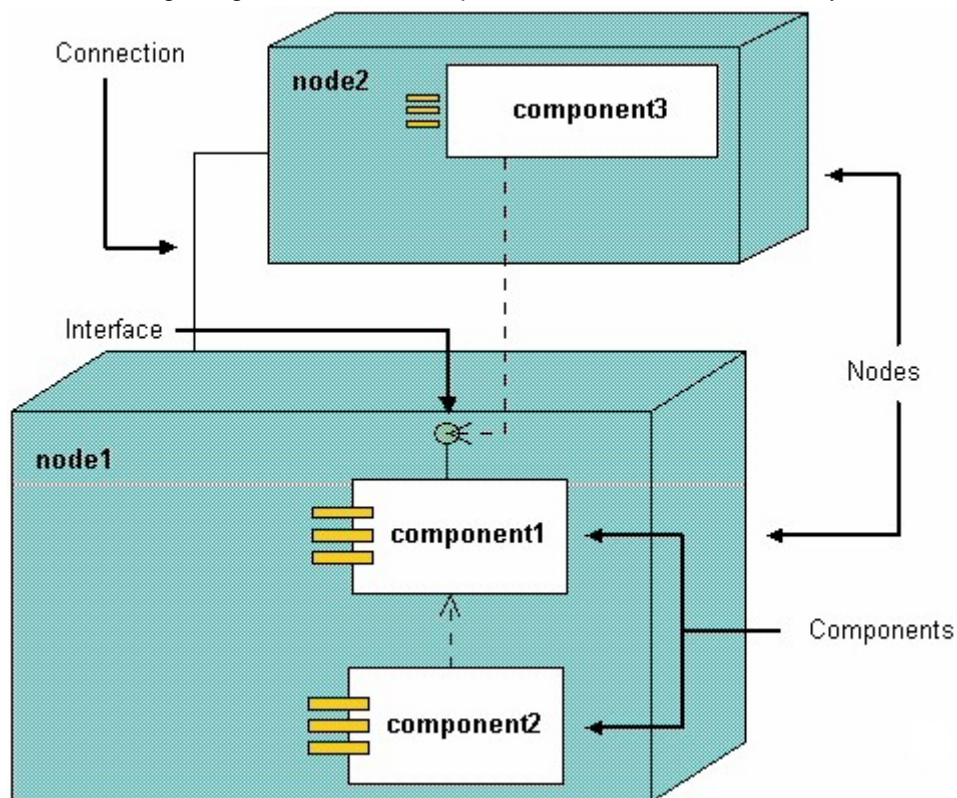
Component Diagrams show the software components (either component technologies such as KParts, CORBA components or Java Beans or just sections of the system which are clearly distinguishable) and the artefacts they are made out of such as source code files, programming libraries or relational database tables.

Components can have interfaces (i.e. abstract classes with operations) that allow associations between components.

Deployment Diagrams

Deployment diagrams show the runtime component instances and their associations. They include Nodes which are physical resources, typically a single computer. They also show interfaces and objects (class instances)

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 Notations

The UML notation is rich and full bodied. It is comprised of two major subdivisions. There is a notation for modeling the static elements of a design such as classes, attributes, and relationships.






There is also a notation for modeling the dynamic elements of a design such as objects, messages, and finite state machines.


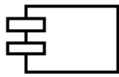
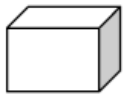
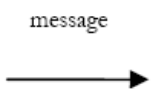
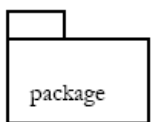
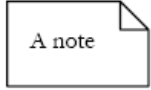
The basic building blocks in UML are things and relationships; these are combined in different ways following different rules to create different types of diagrams. In UML there are nine types of diagrams, below is a list and brief description of them. The more in depth descriptions in the document, will focus on the first five diagrams in the list, which can be seen as the most general, sometimes also referred to as the UML core diagrams.

- ❑ Use case diagrams; shows a set of use cases, and how actors can use them
- ❑ Class diagrams; describes the structure of the system, divided in classes with different connections and relationships
- ❑ Sequence diagrams; shows the interaction between a set of objects, through the messages that may be dispatched between them
- ❑ State chart diagrams; state machines, consisting of states, transitions, events and activities
- ❑ Activity diagrams; shows the flow through a program from an defined start point to an end point
- ❑ Object diagrams; a set of objects and their relationships, this is a snapshot of instances of the things found in the class diagrams
- ❑ Collaboration diagrams; collaboration diagram emphasize structural ordering of objects that send and receive messages.
- ❑ Component diagrams; shows organizations and dependencies among a set of components. These diagrams address static implementation view of the system.
- ❑ Deployment diagrams; show the configuration of run-time processing nodes and components that live on them

Things

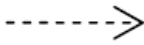
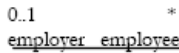



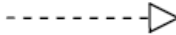
Things are used to describe different parts of a system; existing types of things in UML are presented in following table.

<i>Types of Things</i>			
Name	Symbol	Description	Variations/other related elements
Class		Description of a set of objects that share the same: attributes, operations, relationships and semantics.	<ul style="list-style-type: none"> - actors - signals - utilities
Interface		A collection of operations that specify a service of a class or component.	
Collaboration		An interaction and a society or roles and other elements that work together to provide some cooperative behavior that is bigger than the sum of all the elements. Represent implementation of patterns that make up the system.	
Actor		The outside entity that communicates with a system, typically a person playing a role or an external device	
Use Case		A description of set of sequence of actions that a system perform that produces an observable result of value to a particular actor. Used to structure behavioral things in the model.	

<i>Types of Things</i>			
Name	Symbol	Description	Variations/other related elements
Active class		A class whose objects own a process or execution thread and therefore can initiate a control activity on their own.	- processes - threads
Component		A component is a physical and replacable part that conforms to and provides the realisation of a set of interfaces.	
Node		A physical resource that exists in run time and represents a computational resource.	
Interaction		Set of messages exchanged among a set of objects within a particular context to	- messages - action sequences - links
Packages		General purpose mechanism of organizing elements into groups.	- frameworks - models - subsystems
Note		A symbol for rendering notes and constraints attached to an element or a collection of elements.	

Relationships

The types of UML relationships are shown in the table 2, relationships are used to connect things into well defined models (UML diagrams).

Types of <i>Relationships</i>			
Name	Symbol	Description	Specialization
Dependency		A semantic relationship between two things in which a change to one thing may affect the semantics of the dependent thing.	
Association		<p>Structural relationship that describes a set of links, where a link is a connection between objects.</p> <p>Aggregation and composition are “has-a” relationship. Aggregation (white diamond) is an association indicating that one object is temporarily subordinate or the other, while the composition (black diamond) indicates that an object is a subordinate of another through its lifetime.</p>	<p>Aggregation</p>  <p>Composition</p> 
Generalization		Specialization/generalization relationship in which objects of the specialized element are substitutable for objects of the generalized element.	
Realization		Semantic relationship between two classifiers, where one or them specifies a contract and the other guarantees to carry out the contract.	

Summary

- ❑ Notation is a collection of graphical symbols for expressing model of the system.
- ❑ The deployment diagram shows the configuration of run-time processing elements and the software processes living on them.
- ❑ The description of a use case specifies the flow of events that occur in a use case. This flow of events can be represented as an ‘Activity Diagram’.

Test your Understanding

1. Mention the different kinds of modeling diagrams used?
2. What is a Use Case?

Session 08: Introduction to Design Process and CASE Tools

Learning Objectives

After completing this session, you will be able to:

- ❑ Describe the activities in the object-oriented design process
- ❑ Describe the CASE tools and its benefits
- ❑ Explain the component based software engineering

Object Oriented Design

Object-oriented design is a design strategy where system designers think in terms of 'things' instead of operations or functions. The executing system is made up of interacting objects that maintain their own local state and provide operations on that state information. They hide information about the representation of the state and hence limit access to it. An object-oriented design process involves designing the object classes and the relationships between these classes. When the design is realized as an executing program, the required objects are created dynamically using the class definitions.

Object-oriented design is part of object-oriented development where an object-oriented strategy is used throughout the development process:

- ❑ Object-oriented analysis is concerned with developing an object-oriented model of the application domain. The identified objects reflect entities and operations that are associated with the problem to be solved.
- ❑ Object-oriented design is concerned with developing an object-oriented model of a software system to implement the identified requirements. The objects in an object-oriented design are related to the solution to the problem that is being solved. There may be close relationships between some problem objects and some solution objects but the designer inevitably has to add new objects and to transform problem objects to implement the solution.
- ❑ Object-oriented programming is concerned with realising a software design using an object-oriented programming language. An object-oriented programming language, such as Java, supports the direct implementation of objects and provides facilities to define object classes.

The transition between these stages of development should be a seamless one with the same notation used at each stage. Moving to the next stage involves refining the previous stage by adding detail to existing object classes and devising new classes to provide additional functionality. As information is concealed within objects, detailed design decisions about the representation of data can be delayed until the system is implemented. In some cases, decisions on the distribution of objects and whether or not objects can be sequential or concurrent may also be delayed. This means that software designers are not constrained by details of the system implementation. They can devise designs that can be adapted to different execution environments.

Object-oriented systems should be maintainable as the objects are independent. They may be understood and modified as stand-alone entities.

Changing the implementation of an object or adding services should not affect other system objects. Because objects are associated with things, there is often a clear mapping between real-world entities (such as hardware components) and their controlling objects in the system. This improves the understandability and hence the maintainability of the design.

Objects are potentially reusable components because they are independent encapsulations of state and operations. Designs can be developed using objects that have been created in previous designs. This reduces design, programming and validation costs. It may also lead to the use of standard objects (hence improving design understandability) and reduces the risks involved in software development.

Characteristics of OOD

- ❑ Objects are abstractions of real-world or system entities and manage themselves.
- ❑ Objects are independent and encapsulate state and representation information.
- ❑ System functionality is expressed in terms of object services.
- ❑ Shared data areas are eliminated. Objects communicate by message passing.
- ❑ Objects may be distributed and may execute sequentially or in parallel.

Process Stages

Highlights key activities without being tied to any proprietary process such as the RUP.

- ❑ Define the context and modes of use of the system;
- ❑ Design the system architecture;
- ❑ Identify the principal system objects;
- ❑ Develop design models;
- ❑ Specify object interfaces.

CASE Tools

The basic idea behind CASE is to support each phase of the life cycle with a set of labor saving tools. Some CASE tools concentrate on supporting the early phases of the life cycle. They give automated assistance in the form of automated diagram drawing, screen painting, and corrections checking. Others focus on the implementation phases of the life cycle. They include automated code and test case generators.

In some cases, these tools are used in conjunction with third-and fourth generation languages. In other cases they replace them by allowing the developer to create high-level program specifications from which program code can be generated.

During the development of a software product a number of very different operations have to be carried out, as reflected by the software development life cycle model. A number of different models have been introduced; Rapid Prototyping, Incremental, and the Capability Maturity models are all well known examples. The Waterfall model was one of the earliest, which has been developed and become in fairly widespread use amongst the software industry. In addition to these models, the Spiral model is now receiving a considerable amount of attention within the industry. A key part of the Waterfall model is that no phase is complete until the documentation of each phase has been completed. Typical activities within phases would include estimating resources requirements, drawing up the specification document, coding, carrying out integration testing, and writing the user manual.

With the aid of CASE much of the phases can be carried out considerably speedier, thereby becoming a lot less labor intensive. But CASE is not restricted to assisting with documentation.

This is verified by the fact that CASE changes the software development environment by operations being carried out in a work-station environment. Whereby software is developed in an interactive atmosphere allowing error checks to be carried out early. CASE involves all aspects of computer support for software engineering. Software engineering is the premises on which tried and tested engineering methods, good management practice and the application of appropriate tools reside at.

Benefits of CASE Tools

The introduction of CASE into any organization can cause considerable changes to the time taken by each phase and the distribution of cost within the software life cycle. CASE has a greater emphasis on analysis and design, thereby having a vast impact on reducing maintenance costs.

"CASE can assist directly in the design and support of system development and also provide management information, documentation and control of the project as it develops.

CASE tools ensure that the three C's are achieved:

- ☐ Consistency
- ☐ Completeness
- ☐ Conformance to standards."

Summary

To summarize the potential benefits of CASE are:-

- ☐ Encourages an interactive and workstation environment.
- ☐ Reduces cost especially maintenance.
- ☐ Improves software quality.
- ☐ Speeds up the development process.
- ☐ Precision is allowed to be repeated.
- ☐ Makes structured techniques practical.
- ☐ Increases productivity.

Test your Understanding

1. Which is concerned with realizing a software design using an object-oriented programming language?

Session 09: Use Case Diagram

Learning Objectives

After completing this session, you will be able to:

- ❑ Identify actors, usecase and scope of release.
- ❑ Draw usecase diagram by identifying all the possible relationship that exists.

Introduction

Use case diagrams overview the usage requirements for a system. They are useful for presentations to management and/or project stakeholders. A use case describes event sequences for an actor to use the system. It is a narrative description of the process. A use case is normally actor or event based. An actor will begin a process or an event will happen that the system must respond to.

The Use case diagram is used to identify the primary elements and processes that form the system. The primary elements are termed as "actors" and the processes are called "use cases." The Use case diagram shows which actors interact with each use case.

Elements of a Use Case Diagram

Actors:

An actor portrays any entity (or entities) that perform certain roles in a given system. An actor may be a person, a device, another system or sub-system, or time. An individual in the real world can be represented by several actors if they have several different roles and goals in regards to a system. The different roles the actor represents are the actual business roles of users in a given system. Actors are drawn as stick figures in usecase diagram depicted outside the system boundary.

Example

In the statement "patients visit the doctor in the clinic for medical tests," "doctor" and "patients" are the business roles and can be easily identified as actors in the system.



UseCase:

A use case in a use case diagram is a visual representation of distinct business functionality in a system. The key term here is "distinct business functionality." Use cases describe the behavior of the system when one of these actors sends one particular stimulus. This behavior is described textually. It describes the nature of the stimulus that triggers the use case; the inputs from and

outputs to other actors, and the behaviors that convert the inputs to the outputs. Usecase is shown as horizontal ellipse.

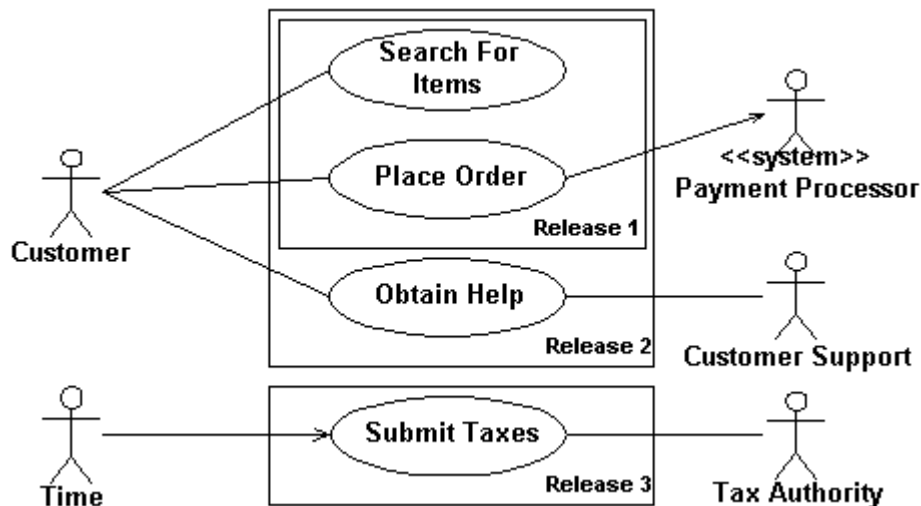
Example

"Make appointment" and "Perform medical tests" in the use case diagram of a clinic system.



System boundary:

A system boundary defines the scope of what a system will be. A system cannot have infinite functionality. So, it follows that use cases also need to have definitive limits defined. A system boundary of a use case diagram defines the limits of the system. The system boundary is shown as a rectangle spanning all the use cases in the system. Also indicate release scope with a System Boundary Box.



Relationships of Use Cases

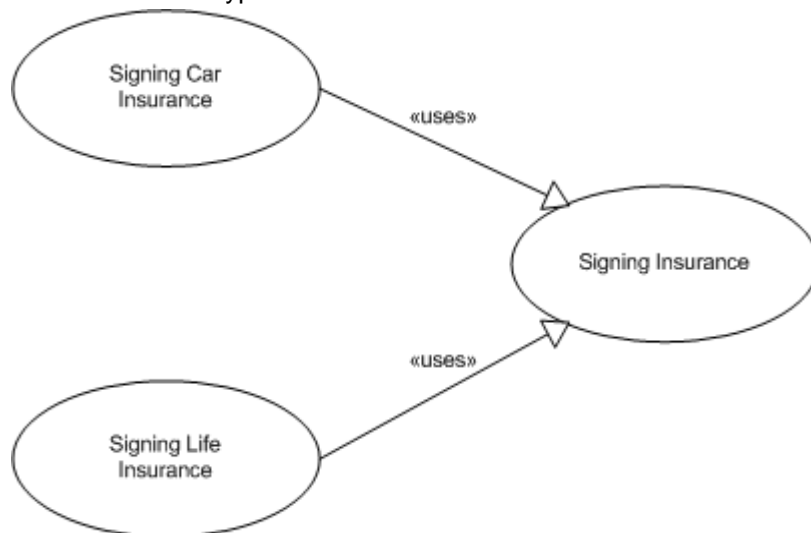
A relationship between two use cases is basically a dependency between the two use cases. This reuse of an existing use case using different types of relationships reduces the overall effort required in defining use cases in a system.

Types of Relationships are:

1. Include
2. Extend
3. Generalization

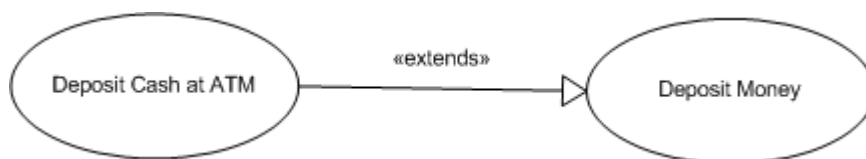
Include:

Behavior common to many use cases - factored off into another use case used by the others. Extra behavior is added into a base use case. This behavior describes the insertion explicitly. The included use case is not a complete process. Use "include" when multiple use cases have a common function that can be used by all. Dashed line with arrow points to subroutine use case. <<include>> or <<uses>> stereotype are used to define the include relation.



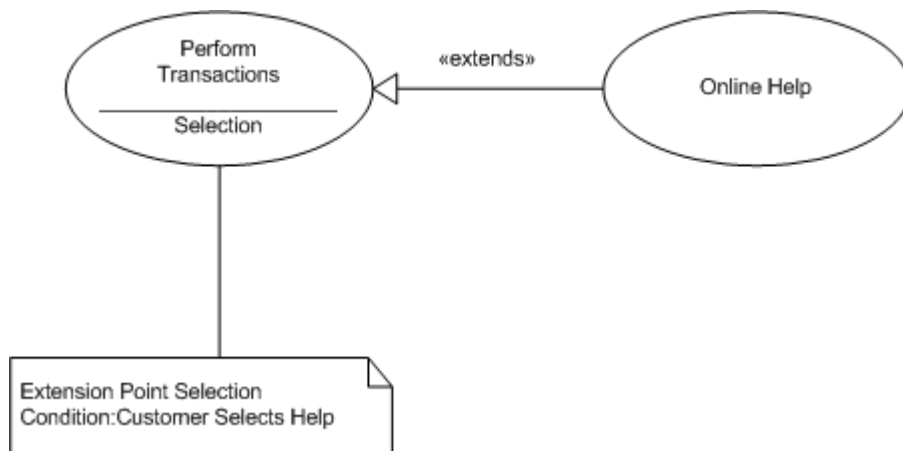
Extend:

An Extend link shows a relationship from one use case to another. It specifies how the behavior defined for the first use case can be inserted into the behavior defined for the second use case. In the banking system example, the Deposit Cash at ATM use case extends the Deposit Money use case. The Deposit Cash at ATM use case enhances the functionality of the Deposit Money. The Deposit Cash at ATM is a specialized version of the generic Deposit Money use case. The extend relationship is depicted with a dotted arrow line. The tip of the arrow points to the parent use case; the child use case is connected at the base of the arrow.

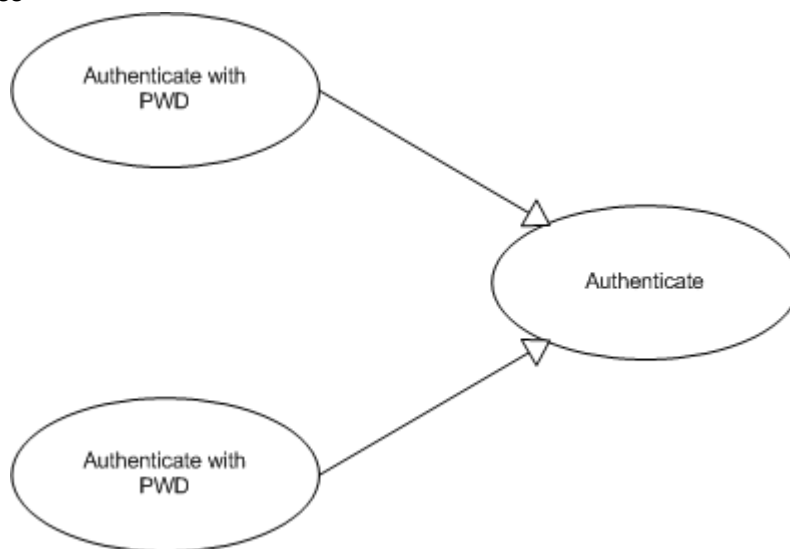


Extension Point:

The point at which an extending usecase is added can be defined by means of an extension point

**Generalization:**

A generalization relationship is a relationship in which one model element (the child) is based on another model element (the parent). Generalization relationships are used in class, component, deployment, and use-case diagrams to indicate that the child receives all of the attributes, operations, and relationships that are defined in the parent. A Generalization relationship is the equivalent of an *inheritance* relationship in object-oriented terms (an "is-a" relationship). A Generalization relationship is indicated by an arrow with a hollow arrowhead pointing to the base, or "parent", class



A usecase can specialize a more general one by adding more detail. The "Authenticate" usecase is a high level usecase describing in general terms, the process of authentication.

Summary

This session helps you to draw a usecase diagram for any given problem statement by applying apt relationships wherever necessary.

Test your Understanding

1. Which statements are true for an actor?
 - a) An actor is a role a user plays with respect to the system
 - b) Generalization is not applicable to actors
 - c) An actor does not need to be human. A subsystem or external system can be modelled as an actor

Session 12: Class Diagram

Learning Objectives

After completing this session, you will be able to:

- ❑ Define class diagram
- ❑ List the notations to draw class diagram
- ❑ List the various types of relationship between classes

Introduction

The class diagram shows the building blocks of any object-orientated system. Class diagrams depict a static view of the model, or part of the model, describing what attributes and behavior it has rather than detailing the methods for achieving operations. Class diagrams are most useful in illustrating relationships between classes and interfaces. Generalizations, aggregations, and associations are all valuable in reflecting inheritance, composition or usage, and connections respectively

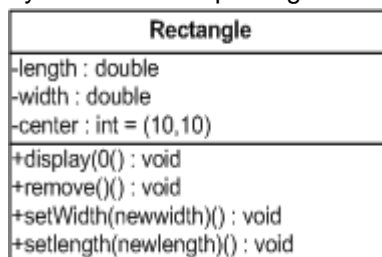
Classes

A class is an element that defines the attributes and behaviors that an object is able to generate. The behavior is described by the possible messages the class is able to understand, along with operations that are appropriate for each message. Classes may also have definitions of constraints, tagged values and stereotypes.

Class Notation

Classes are represented by rectangles which show the name of the class and optionally the name of the operations and attributes. Compartments are used to divide the class name, attributes and operations.

In the diagram below the class contains the class name in the topmost compartment, the next compartment details the attributes, with the "center" attribute showing initial values. The final compartment shows the operations setWidth, setLength and setPosition and their parameters. The notation that precedes the attribute, or operation name, indicates the visibility of the element: if the + symbol is used, the attribute, or operation, has a public level of visibility; if a - symbol is used, the attribute, or operation, is private. In addition the # symbol allows an operation, or attribute, to be defined as protected, while the ~ symbol indicates package visibility.



Visibility

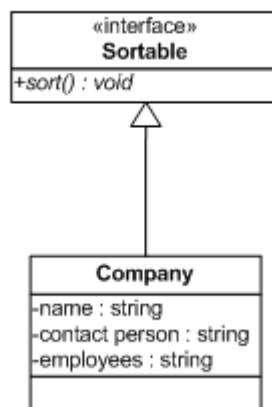
Use visibility markers to signify who can access the information contained within a class. Private visibility hides information from anything outside the class partition. Public visibility allows all other classes to view the marked information. Protected visibility allows child classes to access information they inherited from a parent class.

- ❑ + -> Public
- ❑ -> Private
- ❑ # -> Protected

Types of Relationship

Interface: (Realization)

Many object oriented programming languages do not allow for multiple inheritance. The interface is used to solve the limitations posed by this. Interfaces are very similar to abstract classes with the exception that they do not have any attributes. As well, unlike a class, all of the operations in an interface have no implementation.



Association

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

An association can be named, and the ends of an association can be adorned with role names, ownership indicators, multiplicity, visibility, and other properties. Example: "department offers courses", is an association relationship

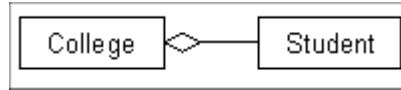


Aggregation

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

Aggregation can occur when a class is a collection or container of other classes, but where the contained classes do not have a strong *life cycle dependency* on the container--essentially, if the container is destroyed, its contents are not.

In UML, it is graphically represented as a *clear* diamond shape on the containing class end of the tree of lines that connect contained class(es) to the containing class.



Composition

Composition is a stronger variant of the "has a" or association relationship; composition is more specific than aggregation. It is represented with solid diamond shape. Composition has a strong life cycle dependency between instances of the container class and instances of the contained class(es): If the container is destroyed, every instance that it contains is destroyed as well.

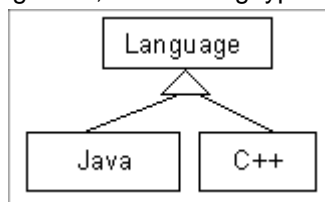
The UML graphical representation of a composition relationship is a filled diamond shape on the containing class end of the tree of lines that connect contained class(es) to the containing class.



Generalization

The Generalization relationship indicates that one of the two related classes (the *subtype*) is considered to be a specialized form of the other (the *supertype*) and supertype is considered as Generalization of subtype. The UML graphical representation of a Generalization is a hollow triangle shape on the supertype end of the line (or tree of lines) that connects it to one or more subtypes.

The generalization relationship is also known as the inheritance or "is a" relationship. The supertype in the generalization relationship is also known as the "parent", super class, base class, or base type. The subtype in the generalization relationship is also known as the "child", subclass, derived class, derived type, inheriting class, or inheriting type.



Multiplicity Indicators

Indicator	Meaning
0..1	Zero or One
1	One Only
0..*	Zero or more
1..*	One or More
3	Three Only
0..5	Zero to Five
5..15	Five to Fifteen

Summary

This session helps you to draw a class diagram for any given problem statement by applying apt relationships wherever necessary.

Test your Understanding

1. Which of the following statements is true for visibility?
 - a) UML uses # for public element
 - b) UML uses - for private element
 - c) UML uses * for protected element
 - d) All of the above

Session 15: Sequence Diagram

Learning Objectives

After completing this session, you will be able to:

- ❑ Create Sequence diagrams.

What is Interaction diagram?

Interaction diagrams model the behavior of use cases by describing the way groups of objects interact to complete the task. The two kinds of interaction diagrams are **sequence** and **collaboration** diagrams.

Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass.

Collaboration diagrams show the relationship between objects and the order of messages passed between them.

Class and object diagrams are static model views. Interaction diagrams are dynamic. They describe how objects collaborate.

When to Use

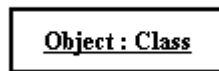
Interaction diagrams are used when you want to model the behavior of several objects in a use case. They demonstrate how the objects collaborate for the behavior. Interaction diagrams do not give an in-depth representation of the behavior. If you want to see what a specific object is doing for several use cases use a state diagram.

Sequence diagrams can be used in a number of ways during the software development process. During requirements capture, they can help define messages sent between the actors and the system, their allowable sequences (sometimes called the "protocol of interaction"), and quality of service constraints. During system modeling, sequence diagrams are used to demonstrate how internal structural elements interact to provide higher level behavior, such as to "realize" a use case. And finally, during testing, sequence diagrams can be used to specify expected behavior (given a set of preconditions and an ordered set of stimuli) and validate output.

How to Draw Sequence Diagrams

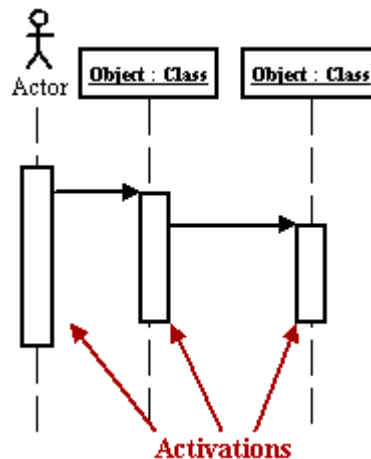
Class roles

Class roles describe the way an object will behave in context. Use the UML object symbol to illustrate class roles, but don't list object attributes.



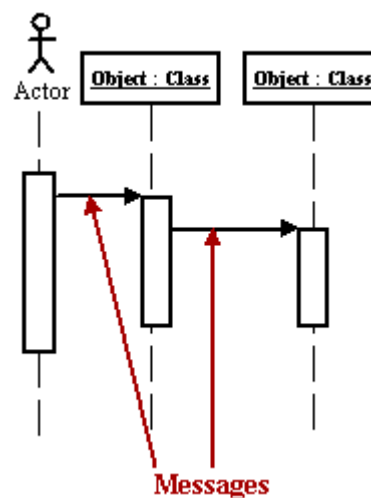
Activation

Activation boxes represent the time an object needs to complete a task.



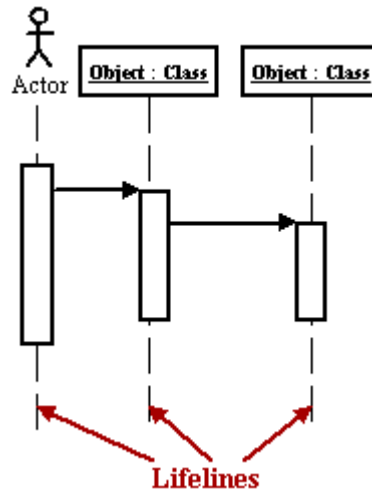
Messages:

Messages are arrows that represent communication between objects. Use half-arrowed lines to represent asynchronous messages. Asynchronous messages are sent from an object that will not wait for a response from the receiver before continuing its tasks.

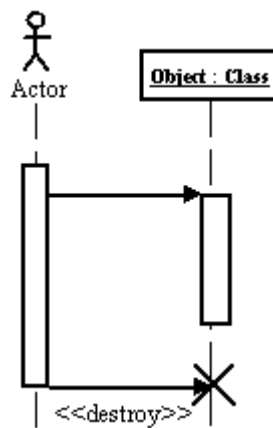


Lifelines:

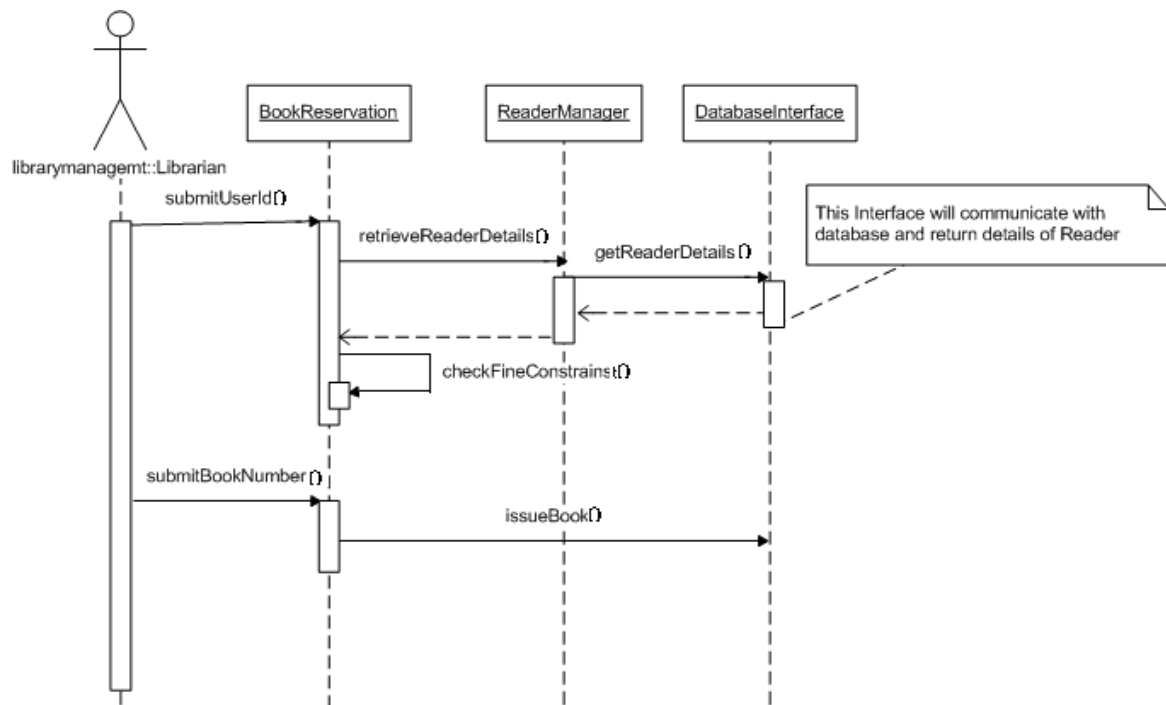
Lifelines are vertical dashed lines that indicate the object's presence over time.

**Destroying Objects**

Objects can be terminated early using an arrow labeled "<<destroy>>" that points to an X.



Sample Sequence Diagram



Summary

- ❑ The sequence diagram is a good diagram to use to document a system's requirements and to flush out a system's design.
- ❑ The reason the sequence diagram is so useful is because it shows the interaction logic between the objects in the system in the time order that the interactions take place.
- ❑ Various standard set of Notations are defined for to create sequence diagram.

Test your Understanding

1. What is sequence diagram? When is it appropriate to use this diagram?

Session 18: Activity Diagram

Learning Objectives

After completing this session, you will be able to:

- ❑ Draw activity diagram
- ❑ Explain notations used for activity diagram

Activity Diagram

Activity Diagram displays a special state diagram where most of the states are action states and most of the transitions are triggered by completion of the actions in the source states. This diagram focuses on flows driven by internal processing. An activity diagram is essentially a fancy flowchart. Activity diagrams and statechart diagrams are related. While a statechart diagram focuses attention on an object undergoing a process (or on a process as an object), an activity diagram focuses on the flow of activities involved in a single process. The activity diagram shows the how those activities depend on one another.

When to Use

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 needs to take place and when they should occur; describing a complicated sequential algorithm; and modeling applications with parallel processes.

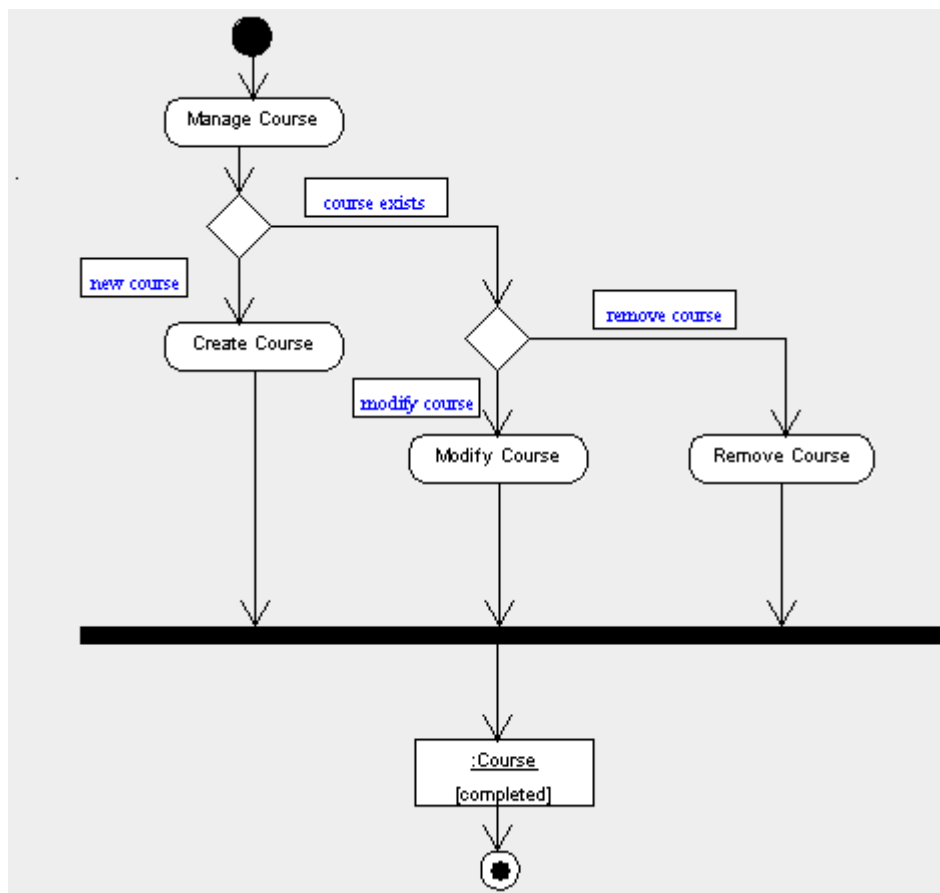
However, activity diagrams should not take the place of interaction diagrams and state diagrams. Activity diagrams do not give detail about how objects behave or how objects collaborate.

Describing the Basic Notation

- ❑ **Initial node:** The filled in circle is the starting point of the diagram. An initial node isn't required although it does make it significantly easier to read the diagram.
- ❑ **Activity final node:** The filled circle with a border is the ending point. An activity diagram can have zero or more activity final nodes.
- ❑ **Activity:** The rounded rectangles represent activities that occur. An activity may be physical, such as Inspect Forms, or electronic, such as Display Create Student Screen.
- ❑ **Flow/edge:** The arrows on the diagram. Although there is a subtle difference between flows and edges I have never seen a practical purpose for the difference although I have no doubt one exists. I'll use the term flow.
- ❑ **Fork:** A black bar with one flow going into it and several leaving it. This denotes the beginning of parallel activity.
- ❑ **Join:** A black bar with several flows entering it and one leaving it. All flows going into the join must reach it before processing may continue. This denotes the end of parallel processing.

- ❑ **Condition:** Text such as [Incorrect Form] on a flow, defining a guard which must evaluate to true in order to traverse the node.
- ❑ **Decision:** A diamond with one flow entering and several leaving. The flows leaving include conditions although some modellers will not indicate the conditions if it is obvious.
- ❑ **Merge:** A diamond with several flows entering and one leaving. The implication is that one or more incoming flows must reach this point until processing continues, based on any guards on the outgoing flow.
- ❑ **Partition:** Organized into partitions, also called swimlanes, indicating who/what is performing the activities.
- ❑ **Flow final:** The circle with the X through it. This indicates that the process stops at this point.

Sample Activity Diagram



Summary

An activity diagram represents the business and operational step-by-step workflows of components in a system. It shows the overall flow of control.

The "states" represent activities, and the transitions represent the completion of those activities.

Test your Understanding

1. Which of the following is not a purpose for using activity diagrams?
 - a) To show the sub-systems that makes up a system
 - b) To model a task
 - c) To describe the logic of an operation

Session 20: Component Diagram

Learning Objectives

After completing this session, you will be able to

- ❑ Identify components, interface and ports.
- ❑ Draw component diagram by identifying all the possible relationship that exists.

Introduction

In the Unified Modeling Language, a component diagram depicts how a software system is split up into components and shows the dependencies among these components. Physical components could be, for example, files, headers, link libraries, modules, executables, or packages. Component diagrams can be used to model and document any system's architecture but are prevalent in the field of software architecture.

Component diagrams illustrate the pieces of software, embedded controllers, etc., that will make up a system. A component diagram has a higher level of abstraction than a Class Diagram - usually a component is implemented by one or more classes (or objects) at runtime.

Importance of Component Diagram

The component diagram's main purpose is to show the structural relationships between the components of a system. In UML 1.1, a component represented implementation items, such as files and executables. Unfortunately, this conflicted with the more common use of the term component," which refers to things such as COM components. Over time and across successive releases of UML, the original UML meaning of components was mostly lost. UML 2 officially changes the essential meaning of the component concept; in UML 2, components are considered autonomous, encapsulated units within a system or subsystem that provide one or more interfaces. Although the UML 2 specification does not strictly state it, components are larger design units that represent things that will typically be implemented using replaceable" modules. But, unlike UML 1.x, components are now strictly logical, design-time constructs.

In component-based development (CBD), component diagrams offer architects a natural format to begin modeling a solution. Component diagrams allow an architect to verify that a system's required functionality is being implemented by components, thus ensuring that the eventual system will be acceptable.

In addition, component diagrams are useful communication tools for various groups. The diagrams can be presented to key project stakeholders and implementation staff. While component diagrams are generally geared towards a system's implementation staff, component diagrams can generally put stakeholders at ease because the diagram presents an early understanding of the overall system that is being built.

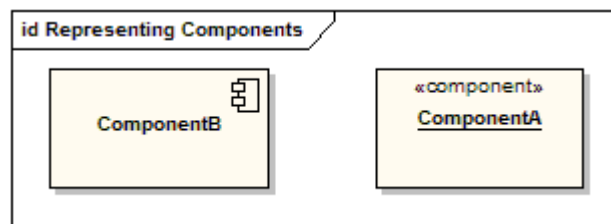
Developers find the component diagram useful because it provides them with a high-level, architectural view of the system that they will be building, which helps developers begin formalizing a roadmap for the implementation, and make decisions about task assignments and/or needed

skill enhancements. System administrators find component diagrams useful because they get an early view of the logical software components that will be running on their systems. Although system administrators will not be able to identify the physical machines or the physical executables from the diagram, a component diagram will nevertheless be welcomed because it provides early information about the components and their relationships (which allows sys-admins to loosely plan ahead).

Elements of a Components Diagram

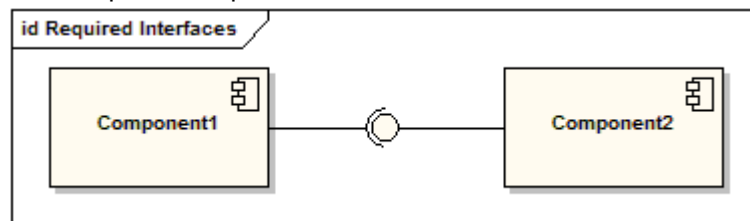
Components:

Components are represented as a rectangular classifier with the keyword «component»; optionally the component may be displayed as a rectangle with a component icon in the right-hand upper corner.



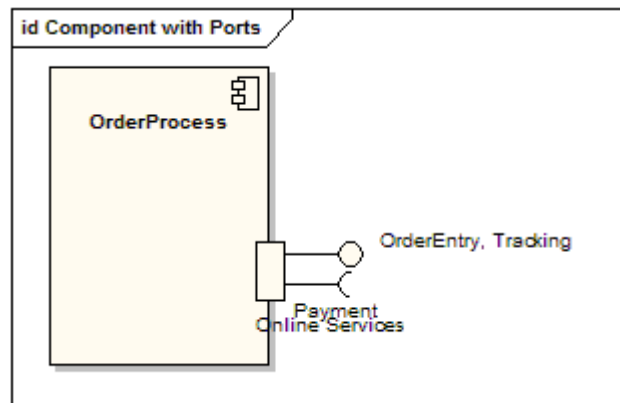
Assembly Connector:

The assembly connector bridges a component's required interface (Component1) with the provided interface of another component (Component2); this allows one component to provide the services that another component requires.



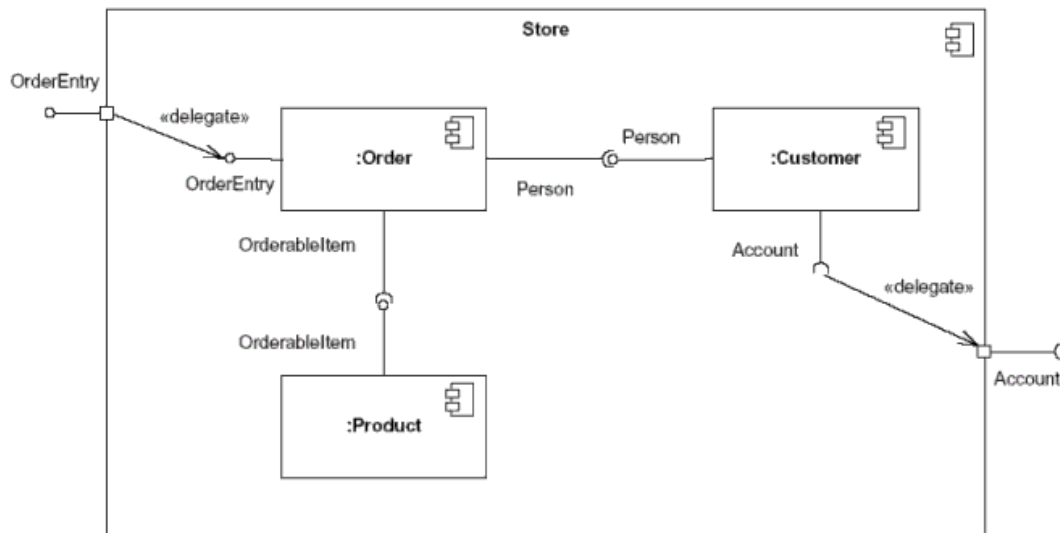
Components with Ports:

Using Ports with component diagrams allows for a service or behavior to be specified to its environment as well as a service or behavior that a component requires. Ports may specify inputs and outputs as they can operate bi-directionally. The following diagram details a component with a port for online services payment along with two provided interfaces order entry and tracking as well as a required interface payment.



Example

Store component provides the interface of OrderEntry and requires the interface of Account. The Store component is made up of three components: Order, Customer, and Product components. Notice how the Store's OrderEntry and Account interface symbols have a square on the edge of the component.



Component's Relationships

When showing a component's relationship with other components, the lollipop and socket notation must also include a dependency arrow (as used in the class diagram). On a component diagram with lollipops and sockets, note that the dependency arrow comes out of the consuming (requiring) socket and its arrow head connects with the provider's lollipop, as shown in Figure.

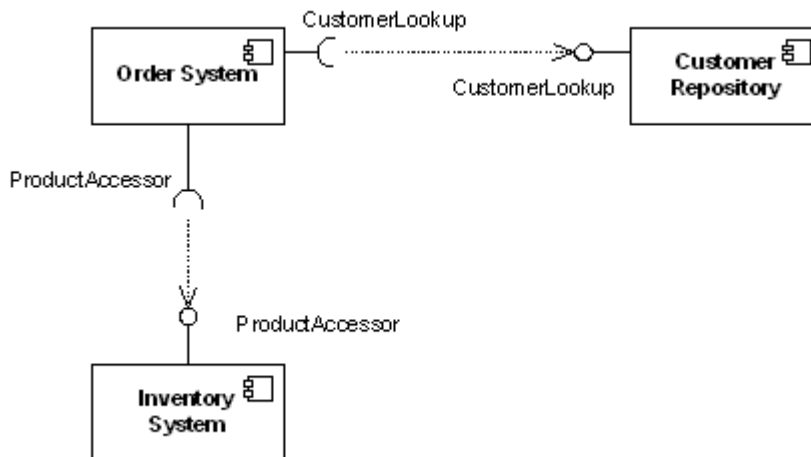


Figure shows that the Order System component depends both on the Customer Repository and Inventory System components. Notice in Figure the duplicated names of the interfaces "CustomerLookup" and "ProductAccessor." While this may seem unnecessarily repetitive in this example, the notation actually allows for different interfaces (and differing names) on each component depending on the implementation differences (e.g., one component provides an interface that is a subclass of a smaller required interface).

Summary

The component diagram is a very important diagram that architects will often create early in a project. However, the component diagram's usefulness spans the life of the system. Component diagrams are invaluable because they model and document a system's architecture.

Component diagrams document a system's architecture, the developers and the eventual system administrators of the system find this work product-critical in helping them understand the system.

Test your Understanding

1. Which statements are true for a Component?
 - a) Components representing the functional elements of the application system.
 - b) Components representing the external interfaces on which the system relies for services.
 - c) Components representing system service interfaces.

Session 21: Deployment Diagram

Learning Objectives

After completing this session, you will be able to:

- ❑ After completing this session you will be able to
- ❑ Identify components and nodes.
- ❑ Draw deployment diagram by identifying all the possible relationship that exists.

Introduction

Deployment diagram depicts a static view of the run-time configuration of processing nodes and the components that run on those nodes. In other words, deployment diagrams show the hardware for your system, the software that is installed on that hardware, and the middleware used to connect the disparate machines to one another.

Deployment diagrams can also be created to explore the architecture of embedded systems, showing how the hardware and software components work together. In short, you may want to consider creating a deployment diagram for all but the most trivial of systems.

Importance of Deployment Diagram

Deployment diagram provides a different perspective of the application. The deployment diagram captures the configuration of the runtime elements of the application.

This diagram is by far more useful when a system is built and ready to be deployed. But, this does not mean that you should start on your deployment diagram after your system is built. On the contrary, your deployment diagram should start from the time your static design is being formalized using, say, class diagrams. This deployment diagram then evolves and is revised until the system is built. It is always a best practice to have visibility of what your deployment environment is going to be before the system is built so that any deployment-related issues are identified to be resolved and not crop up at the last minute. The general rule of thumb is that correction costs due to changes increase as the project nears completion.

So, how are deployment diagrams and component diagrams related? Essentially, the components in a component diagram are contained in the deployment diagram elements. Hence, while components provide the application functionality, the deployment diagram elements provide the necessary environment for the components to execute in.

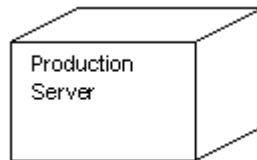
The basic deployment diagram element is the node. The node represents the environment in which a component or a set of components execute. This means that a node in a deployment diagram can represent a multitude of things—physical hardware such as a server machine, a system software like an operating system, or even application infrastructure software like a Web server, application server, database server, and so forth. The different nodes in the deployment diagram can be interconnected to represent interdependencies, thus providing a deployment

diagram that is easy to comprehend and provides the complete deployment environment of a system.

Elements of a Deployment Diagram

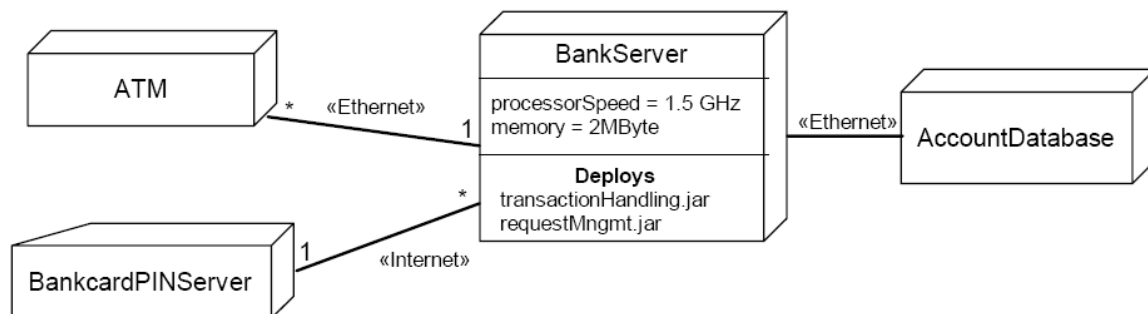
Node

A node represents a computational resource, generally having at least some memory and, often, processing capability. Nodes are used to model the topology of the hardware on which the system executes.



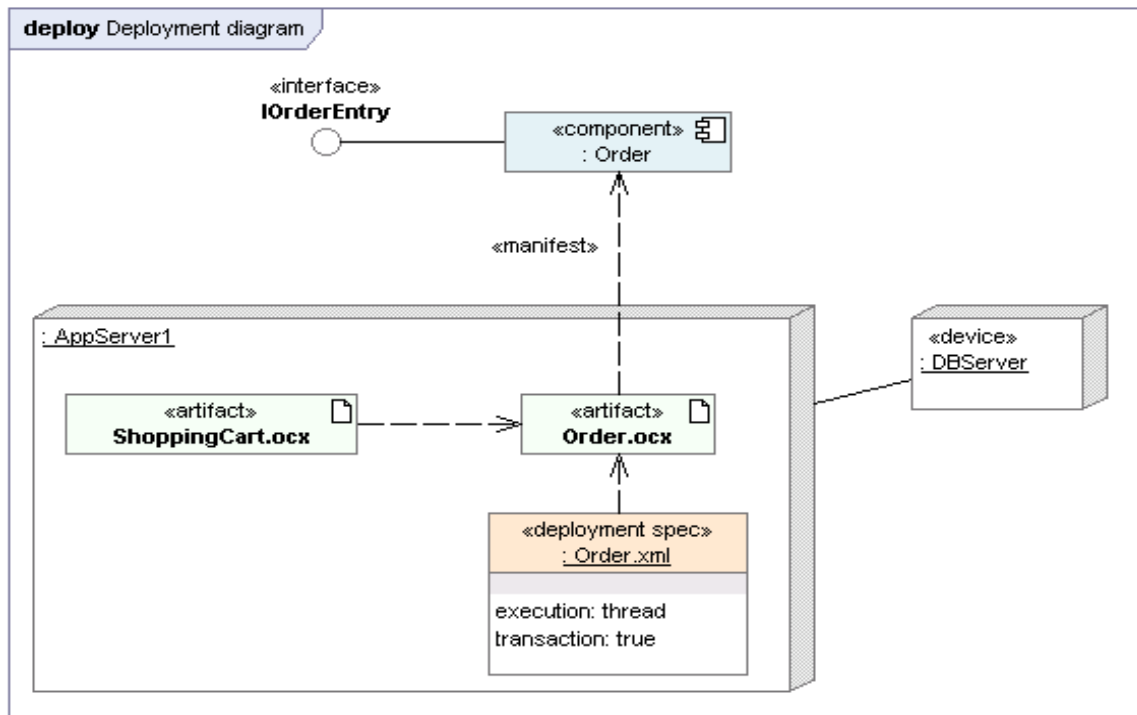
Connector

Nodes can be connected each other in order to exchange data and commands. The communication channels between nodes are modeled by associations known from class diagrams. Even role names and multiplicities can be attached to associations. In the context of deployment diagrams, an association represents a physical connection among nodes, such as an Ethernet connection, a shared bus, or even a satellite link between distant processors.



Example

Store component provides the interface of Order Entry and requires the interface of Account. The Store component is made up of three components: Order, Customer, and Product components. Notice how the Store's Order Entry and Account interface symbols have a square on the edge of the component.



Relationships in Deployment Diagram

In UML, a relationship is a connection between model elements. A UML relationship is a type of model element that adds semantics to a model by defining the structure and behavior between model elements. UML relationships are grouped into the following categories:

Association relationships

In UML models, an association is a relationship between two classifiers, such as classes or use cases, which describes the reasons for the relationship and the rules that govern the relationship.

Communication paths:

In UML modeling, a communication path is a type of association between nodes in a deployment diagram that shows how the nodes exchange messages and signals.

Deploy relationships in UML:

In UML, deploy relationships specify that a particular node type supports the deployment of an artifact type.

Generalization relationships

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

Manifestation relationships:

In UML modeling, a manifestation relationship shows which model elements, such as components or classes, are manifested in an artifact. The artifact manifests, or includes, a specific implementation for, the features of one or several physical software components.





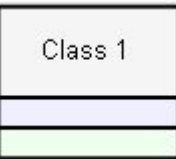
Summary

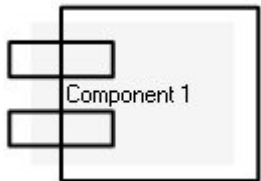





Deployment diagrams show the hardware for your system, the software that is installed on that hardware, and the middleware used to connect the disparate machines to one another.






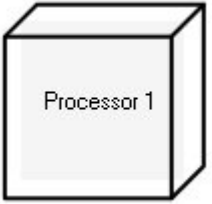
Test your Understanding




1. Which statements are true for a Component?
 - a) Explore the dependencies that your system has with other systems that are currently in, or planned for, your production environment.
 - b) Depict the hardware/network infrastructure of an organization.
 - c) Depict a major deployment configuration of a business application.

Glossary

<input type="checkbox"/> Activity: A step or action within an Activity Diagram. Represents an action taken by the system or by an Actor.	
<input type="checkbox"/> Activity Diagram: A glorified flowchart that shows the steps and decisions and parallel operations within a process, such as an algorithm or a business process.	
<input type="checkbox"/> Actor: A person or external computer system that interacts with the system under design.	
<input type="checkbox"/> Association: A connection between two elements of a Model. This might represent a member variable in code, or the association between a personnel record and the person it represents, or a relation between two categories of workers, or any similar relationship. By default, both elements in an Association are equal, and are aware of each other through the Association. An Association can also be a Navigable Association, meaning that the source end of the association is aware of the target end, but not vice versa.	
<input type="checkbox"/> Association Class: A Class that represents and adds information to the Association between two other Classes.	
<input type="checkbox"/> Attribute: A data field or property that represents information about a Classifier.	
<input type="checkbox"/> Base Class: A Class which defines Attributes and Operations that are inherited by a Subclass via a Generalization relationship.	
<input type="checkbox"/> Branch: A decision point in an Activity Diagram. Multiple Transitions emerge from the Branch, each with a Guard Condition. When control reaches the Branch, exactly one Guard Condition must be true; and control follows the corresponding Transition.	
<input type="checkbox"/> Class: A category of similar Objects, all described by the same Attributes and Operations and all assignment-compatible.	
<input type="checkbox"/> Class Diagram: A diagram that shows relationships between various Classes and Interfaces.	
<input type="checkbox"/> Classifier: A UML element that has Attributes and Operations. Specifically, Actors, Classes, and Interfaces.	
<input type="checkbox"/> Collaboration: A relation between two Objects in a Collaboration Diagram, indicating that Messages can pass back and forth between the Objects.	

<input type="checkbox"/> Collaboration Diagram: A diagram that shows Collaborations between Objects and the Messages that pass along those Collaborations to carry out some behavior.	
<input type="checkbox"/> Component: A deployable unit of code within the system.	
<input type="checkbox"/> Component Diagram: A diagram that shows relations between various Components and Interfaces.	
<input type="checkbox"/> Dependence: A relationship that indicates one Classifier knows the Attributes and Operations of another Classifier, but isn't directly connected to any instance of the second Classifier.	
<input type="checkbox"/> Deployment Diagram: A diagram that shows relations between various Processors.	
<input type="checkbox"/> Element: Any item that appears in a Model.	
<input type="checkbox"/> Event: In a State Diagram, this represents a signal or event or input that causes the system to take an action or switch States.	
<input type="checkbox"/> Final State: In a State Diagram or an Activity Diagram, this indicates a point at which the diagram completes.	
<input type="checkbox"/> Fork: A point in an Activity Diagram where multiple parallel control threads begin.	
<input type="checkbox"/> Generalization: An inheritance relationship, in which a Subclass inherits and adds to the Attributes and Operations of a Base Class.	
<input type="checkbox"/> Initial State: In a State Diagram or an Activity Diagram, this indicates the point at which the diagram begins.	
<input type="checkbox"/> Interface: A Classifier that defines Attributes and Operations that form a contract for behavior. A provider Class or Component may elect to Realize an Interface (i.e., implement its Attributes and Operations). A client Class or Component may then Depend upon the Interface and thus use the provider without any details of the true Class of the provider.	
<input type="checkbox"/> Join: A point in an Activity Diagram where multiple parallel control threads synchronize and rejoin.	
<input type="checkbox"/> Lifeline: A line in a Sequence Diagram that indicates the duration of an Object.	
<input type="checkbox"/> Member: An Attribute or an Operation within a Classifier.	
<input type="checkbox"/> Merge: A point in an Activity Diagram where different control paths come together.	

<input type="checkbox"/> Message: In a Sequence Diagram or a Collaboration Diagram, a communication from one Object to another, delivering information or requesting a service.	
<input type="checkbox"/> Model: The central UML artifact. Consists of various elements arranged in a hierarchy by Packages, with relations between elements as well.	
<input type="checkbox"/> Navigability: Indicates which end of a relationship is aware of the other end. Relationships can have bidirectional Navigability (each end is aware of the other) or single directional Navigability (one end is aware of the other, but not vice versa).	
<input type="checkbox"/> Navigable Association: An Association with single directional Navigability.	
<input type="checkbox"/> Note: A text note added to a diagram to explain the diagram in more detail.	
<input type="checkbox"/> Note Attachment: A dashed line connecting a Note to an element that it describes.	
<input type="checkbox"/> Object: In an Activity Diagram, an object that receives information from Activities or provides information to Activities. In a Collaboration Diagram or a Sequence Diagram, an object that participates in the scenario depicted in the diagram. In general: one instance or example of a given Classifier (Actor, Class, or Interface).	
<input type="checkbox"/> Operation: A method or function that a Classifier can perform.	
<input type="checkbox"/> Package: A Model element that divides the Model into a hierarchy.	
<input type="checkbox"/> Package Diagram: A Class Diagram in which all of the elements are Packages and Dependencies.	
<input type="checkbox"/> Parameter: An argument to an Operation.	
<input type="checkbox"/> Private: A Visibility level applied to an Attribute or an Operation, indicating that only code for the Classifier that contains the member can access the member.	
<input type="checkbox"/> Processor: In a Deployment Diagram, this represents a computer or other programmable device where code may be deployed.	
<input type="checkbox"/> Protected: A Visibility level applied to an Attribute or an Operation, indicating that only code for the Classifier that contains the member or for its Subclasses can access the member.	
<input type="checkbox"/> Public: A Visibility level applied to an Attribute or an Operation, indicating that any code can access the member.	

<input type="checkbox"/> Realization: Indicates that a Component or a Class provides a given Interface.	
<input type="checkbox"/> Sequence Diagram: A diagram that shows the existence of Objects over time, and the Messages that pass between those Objects over time to carry out some behavior.	
<input type="checkbox"/> State: In a State Diagram, this represents one state of a system or subsystem: what it is doing at a point in time, as well as the values of its data.	
<input type="checkbox"/> State Diagram: A diagram that shows States of a system or subsystem, Transitions between States, and the Events that cause the Transitions.	
<input type="checkbox"/> Static: A modifier to an Attribute to indicate that there's only one copy of the Attribute shared among all instances of the Classifier. A modifier to an Operation to indicate that the Operation stands on its own and doesn't operate on one specific instance of the Classifier.	
<input type="checkbox"/> Stereotype: A modifier applied to a Model element indicating something about it which can't normally be expressed in UML. In essence, Stereotypes allow you to define your own "dialect" of UML.	
<input type="checkbox"/> Subclass: A Class which inherits Attributes and Operations that are defined by a Subclass via a Generalization relationship.	
<input type="checkbox"/> Swimlane: An element of an Activity Diagram that indicates what parts of a system or a domain perform particular Activities. All Activities within a Swimlane are the responsibility of the Object, Component, or Actor represented by the Swimlane.	
<input type="checkbox"/> Transition: In an Activity Diagram, represents a flow of control from one Activity or Branch or Merge or Fork or Join to another. In a State Diagram, represents a change from one State to another.	
<input type="checkbox"/> Use Case: In a Use Case Diagram, represents an action that the system takes in response to some request from an Actor.	
<input type="checkbox"/> Use Case Diagram: A diagram that shows relations between Actors and Use Cases.	
<input type="checkbox"/> Visibility: A modifier to an Attribute or Operation that indicates what code has access to the member. Visibility levels include Public, Protected, and Private.	

References

Websites

- ❑ http://atlas.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/diagrams.htm
- ❑ <http://www.objectmentor.com/resources/articles/umlClassDiagrams.pdf>
- ❑ <http://www.ece.rutgers.edu/~parashar/Classes/00-01/ece452/slides/1-UMLDiagramTypes.pdf>

Books

- ❑ **UML Distilled:** A Brief Guide to the Standard Object Modelling Language by Martin Fowler
- ❑ The Unified Modelling Language User Guide by Grady Booch
- ❑ UML 2.0 in a Nutshell (In a Nutshell (O'Reilly)) by Dan Pilone

STUDENT NOTES: