UML Tutorial

Part 1: Theory

From the Web site
http://odl-skopje.etf.ukim.edu.mk/uml-help/

Developed by Slobodan Kalajdziski

Before You Start

Welcome to the Unified Modeling Language Tutorial in 7 days. The goal of this course is to give you the basic knowledge about UML diagrams. It consists of 7 days, 4 days have theory material, and 3 days are practical using of learned theory. The material is divided into parts fitting into one day.

Several lessons were introduced every day. After that a list of answered questions are given to you, and a list of questions and exercises to do by yourself, to strengthen your understanding and put your newfound knowledge in use.

Day 1 gives you UML basics, introduces you with GRAPPLE and shortly explains every UML diagram.

DAY 2, DAY 3 and **DAY 4** presents ways of creating Class diagrams and Use Case diagrams, State diagrams, Sequence diagrams and Collaboration diagrams, Activity diagrams, Component diagrams and Deployment diagrams respectively. For all of the diagrams a simple example is provided, which will guide you through the process of creating learned diagram.

DAY 5 and **DAY 6** dives into practical use of the UML diagram during the process of modeling a system - the **Digital Library**. During these two days you'll meet with all the steps in process of modeling a business system. Parts of diagrams given in these lessons are not developed completely. This gives you a way of completing them and put you into a process of developing a model for a Digital Library.

In **DAY 7** finished and relax yourself

DAY1

D1.1 Introducing the UML (Unified Modeling Language)

As the world becomes more complex, the computer-based systems that inhabit the world also must increase in complexity. They often involve multiple pieces of hardware and software, networked across great distances, linked to databases that contain mountains of information. If you want to make systems that deal with this, how do you get your hands around the complexity?

The key is to organize the design process in a way that clients, analysts, programmers and other involved in system development can understand and agree on. **The UML provides the organization.**

Consider this: Would you tell a building contractor that you want a 4 bedroom, 3 bath home, about 2000 square feet - Start building it! We'll hammer out the details as we go along? We all know this is ludicrous. But sadly, this method of development is all too common in the software industry. **Just as you would work with an architect to design a blueprint that would diagram exactly how the house is to be built, you will work with us on an UML diagram that will document exactly how your custom software system will be built.**

The UML was released in 1997 as a method to diagram software design. It was designed by a consortium of the best minds in object oriented analysis and design. It is by far the most exciting thing to happen to the software industry in recent years. Every other engineering discipline has a standard method of documentation. Electronic engineers have schematic diagrams, architects and mechanical engineers have blueprints and mechanical diagrams. The software industry now has UML.

Before we move on next lesson consider **some of the benefits of UML:**

1. Your software system is professionally designed and documented before it is coded. You will know exactly what you are getting, in advance.
2. Since system design comes first, reusable code is easily spotted and coded with the highest efficiency. You will have lower development costs.
3. Logic 'holes' can be spotted in the design drawings. Your software will behave as you expect it to. There are fewer surprises.
4. The overall system design will dictate the way the software is developed. The right decisions are made before you are married to poorly written code. Again, your overall costs will be less.
5. UML lets us see the big picture. We can develop more memory and processor efficient systems.
6. When we come back to make modifications to your system, it is much easier to work on a system that has UML documentation. Much less 'relearning' takes place. Your

system maintenance costs will be lower.

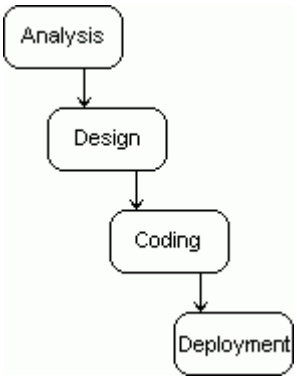
7. If you should find the need to work with another developer, the UML diagrams will allow them to get up to speed quickly in your custom system. Think of it as a schematic to a radio. How could a tech fix it without it?
8. If we need to communicate with outside contractors or even your own programmers, it is much more efficient.

Using the Unified Modeling Language will result in lower overall costs, more reliable and efficient software, and a better relationship with all parties involved. Software documented with UML can be modified much more efficiently. Your software will have a future.

Modeling of systems, old way vs. new way

System is a combination of software and hardware that provides a solution for a business problem.

Process of developing that system involves a lot of people. First of all is the ***client***, the person who has the problem to be solved. An ***analyst*** documents the client's problem and relays it to ***developers***, programmers who build the software that solves the problem, test it and deploy it on computer hardware. This is necessary because systems today are so complex, knowledge has become so specialized that one person can't know all the facets of a business, understand the problem, design a solution, translate it into a program, deploy the program onto hardware, and make sure the hardware components all work together correctly.



The waterfall method for modeling of systems

- The old way of system modeling, known as the ***waterfall method***, specifies that analysis, design, coding and deployment follow one another. Only when one is complete can the next one begin. If an analyst hands off analysis to a designer, who hands off a design to a developer, chances are that the three team members will rarely work together and share important insights. Usually the adherents of the waterfall method give coding a big amount of project time. It takes a valuable time away from analysis and design.

- In the new way, contemporary software engineering stress continuing interplay among the stages of development. Analysts and designers, for example, go back and forth to evolve a solid foundation for the programmers. Programmers, in turn, interact with analysts and designers to share their insights, modify designs, and strengthen their code. The advantage is that as understanding grows, the team incorporates new ideas and builds a stronger system.

D1.2 RAD (Rapid Application Development)

RAD consists of five segments:

1. Requirements gathering
2. Analysis
3. Design
4. Development
5. Deployment

Requirements gathering

This segment consists of a several actions. Before moving on to actions, it is important to know that if you don't understand what the client wants, you'll never build the right system.

- **Discover Business Processes**: Here analysts gain an understanding of the client's business processes, by interviewing the client or a knowledgeable client-designated person and asks the interviewee to go through the relevant processes step-by-step. ***Product*: Activity diagram(s).**
- **Perform Domain Analysis**: The analyst interviews the client with the goal of understanding the major entities in the client's domain. During the conversation between the client and the analyst, another team member takes notes. The object modeler listens for nouns and starts by making each noun a class. Ultimately, some nouns will become attributes. He or she also listens for verbs, which will become operations of the classes. ***Product*: High-level class diagram and a set of meeting notes.**
- **Identify Cooperating Systems**: Early in the process the development team finds out exactly which systems the new system will depend on, and which systems will depend on it. A system engineer takes care of this action. ***Product*: Deployment diagram.**
- **Discover System Requirements**: In this action team goes through its first **Joint Application Development (JAD)** session. This session brings together decision makers from client's organization, potential users and the members of the development team. A facilitator moderates the session. The facilitator's job is to elicit from the decision-makers and the users what they want the system to do. At

least two team members should be taking notes, and the object modeler should be refining the class diagram derived earlier. ***Product*: Package diagram.**

- **Present Results to Client**: When the team finishes all the Requirements actions, the project manager presents the results to the client.

Analysis

Now the team drills down into the results of the Requirements segment and increases its understanding of the problem. In fact, some of the actions begin during the Requirements segment, while the object modeler begins refining the class diagram.

- **Understand System Usage**: In a JAD session with potential users, the development team works with the users to discover the actors who initiate each use case from the Requirements JAD session, and the actors who benefit from those use cases (Actor can be a system as well as a person). ***Product*: Use case diagram(s).**
- **Flesh Out Use Cases**: Lets continue to work with users. The objective is to analyze the sequence of steps in each use case. ***Product*: Text description of the steps in each use case diagram.**
- **Refine the Class Diagrams**: The object modeler, during the JAD sessions listening all the discussions, refines the class diagram. He or she should be filling in the names of the associations, abstract classes, multiplicities, generalizations and aggregations. ***Product*: Refined class diagram.**
- **Analyze Changes of State in Objects**: Also object modeler refines the model by showing changes of state wherever necessary. ***Product*: State diagram.**
- **Define the Interactions Among Objects**: At this moment development team has a set of use cases and refined class diagram. it's time to define how the objects interact. The object modeler develops a set of diagrams which includes state changes. ***Product*: Sequence and collaboration diagrams.**
- **Analyze Integration with Cooperating Systems**: The system engineer, proceeding in parallel with all the preceding steps, uncovers specific details of the integration with the cooperating systems. What type of communication is involved? What is the network architecture? If the system has to access databases, and if which are the types of databases.
  ***Product*: Detailed deployment diagram and if necessary data models.**

Design

In this segment, the team works with the results of the Analysis segment to design the solution. Design and Analysis should go back and forth until the design is complete.

- **Develop and Refine Object Diagrams**: Programmers take the class diagram and generate any necessary object diagrams by examining each operation and developing a corresponding activity diagram. This activity diagrams will serve as the basis for much of the coding in the Development segment. ***Product*: Activity diagrams.**

- **Develop Component Diagrams**: In this action programmers also play a major role. The task here is to visualize the components that will result from the next segment and show the dependencies among them. ***Product*: Component diagrams.**
- **Plan for Deployment**: After aiming the component diagrams, the system engineer begins planning for deployment and for integration with cooperating systems. Created diagram shows where the components will reside. ***Product*: Part of the deployment diagram developed earlier.**
- **Design and Prototype User Interface**: This involves another JAD session with the users, continuation of the prior JAD sessions. This is a typical indication of the interplay between Analysis and Design. A GUI analyst works with the users to develop paper prototypes of screen that correspond to groups of use cases. ***Product*: Screen shots of the screen prototypes.**
- **Design Tests**: Preferably, a developer or test specialist from outside the development team uses the uses case diagrams to develop test scripts for automated test tools. ***Product*: Test scripts.**
- **Begin Documentation**: Documentation specialists work with the designers to begin story-boarding the documentation and arriving at a high-level structure for each document. ***Product*: Document structure.**

Development

With enough analysis and design, this segment should go quickly and smoothly. In this segment programmers take over.

- **Construct Code**: With the class, object, activity and component diagrams in hand, programmers construct the code for the system. ***Product*: The code.**
- **Test Code**: This action feeds back into the preceding action and vice versa, until code passes all levels of testing, developed in previous segment (Design Tests). ***Product*: Test results.**
- **Construct User Interfaces, Connect to Code and Test**: Also and this action is a connection between Design and Development. Here a GUI specialist construct approved interface prototypes and connects them to code. Also and testing the interface ensures that the interfaces work correctly. ***Product*: Functioning system, complete with user interfaces.**
- **Complete Documentation**: During the development segment, documentation experts work in parallel with programmers to complete of all documentation. ***Product*: System documentation.**

Deployment

When development is complete, the system is deployed on the appropriate hardware and integrated with the cooperation systems.

- **Plan for Backup and Recovery**: This action can start long before the development segment begins. The system engineer creates a plan for steps to follow in case the system crashes. ***Product*: The crash recovery plan.**
- **Install the Finished System on Appropriate Hardware**: This step performs the system engineer with any necessary help from the programmers. ***Product*: Fully deployed system.**
- **Test the Installed System**: After installing the software on appropriate computer(s), the development team tests the installed system. Does it perform as it's supposed to? Does the backup and recovery plan work? Results of the tests determine whether further refinement is necessary. ***Product*: Test results.**
- **Celebrate**: When all of the work is finished, the development team may go somewhere to celebrate for their success.

A skeleton of development process is GRAPPLE (Guidelines for Rapid APPLication Engineering), which consist of five segments: Requirements gathering, Analysis, Design, Development and Deployment. Each segment consists of a number of actions, and each action results in a work-product. UML diagrams are work products for many of the actions.

D1.3 UML Components

The UML certain number of graphical elements combined into diagrams. Because it is a language, the UML has rules for combining these elements.

The purpose of the diagrams is to present multiple views of a system, and this set of multiple views is called a **model**.

UML model describes what a system is supposed to do. It doesn't tell how to implement the system.

Let's take a brief look of all the UML diagrams. UML consists of nine basic diagrams, but bear in mind that hybrids of these diagrams are possible.

- **Class Diagram**
  Things naturally fall into categories (computers, automobiles, trees...). We refer to these categories as classes. **A class is a category or group of things that have similar attributes and common behaviors.**

- **Object Diagram**
  An object is an instance of a class - a specific thing that has specific values of the attributes and behavior.
- **Use Case Diagram**
  A use case is a description of a system's behavior from a user's standpoint. For system developers, this is a valuable tool: it's a tried-and-true technique for gathering system requirements from a user's point of view. That's important if the goal is to build a system that real people can use. In graphical representations of

use cases a symbol for the actor is used. **The actor is the entity that initiates the use case. It can be a person or another system.**

- **State Diagram**
  At any given time, an object is in a particular state. State diagrams represent these states and their changes during time. Every state diagram starts with a symbol that represents start state, and ends with symbol for the end state. For example every person can be a newborn, infant, child, adolescent, teenager or adult.
- **Sequence Diagram**
  Class diagrams and object diagrams represent static information. In a functioning system, however, objects interact with one another, and these interactions occur over time. **The UML sequence diagram shows the time-based dynamics of the interaction.**
- **Activity Diagram**
  The activities that occur within a use case or within an object's behavior typically occur in a sequence. This sequence is represented with activity diagrams.
- **Collaboration Diagram**
  The elements of a system work together to accomplish the system's objectives, and a modeling language must have a way of representing this. The UML collaboration diagram is designed for this purpose.
- **Component Diagram**
  Today in software engineering we have team-based development efforts, where everyone has to work on different component. That's important to have a component diagram in modeling process of the system.
- **Deployment Diagram**
  The UML deployment diagram shows the physical architecture of a computer-based system. It can depict the computers and devices, show their connections with one another, and show the software that sits on each machine.

System development is a human activity. Without an easy-to-understand notation system, the development process has great potential for error.
Consisting of a set of diagrams, the UML provides a standard that enables the system analyst to build a multifaceted blueprint that's comprehensible to clients, programmers, and everyone involved in the development process. It's necessary to have all these diagrams because each one speaks to a different stakeholder in the system.
The UML model tells what a system is supposed to do. It doesn't tell how.

DAY 2

D2.1 Class Diagrams

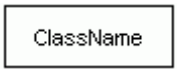
- How to extract classes, their attributes and methods from interviewing a client. And then bind the classes with relationships to form the class diagram. You'll learn more about:

1. Visualizing a Class (attributes and operations)
2. Associations
3. Inheritance & Generalization
4. Aggregations
5. Interfaces & Realizations
6. Visibility

1. Visualizing a Class

Things fall into categories - classes. The UML class diagram consists of several classes connected with relationships. But how UML represents a class?

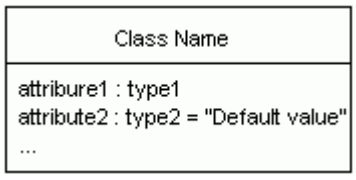
The rectangle is the icon for the class. The name of the class is, by convention, a word with an initial uppercase letter. It appears near the top of the rectangle. If your class name has more than one word name, then join the words together and capitalize the first letter of the every word.



Class name is written at the top of the rectangle

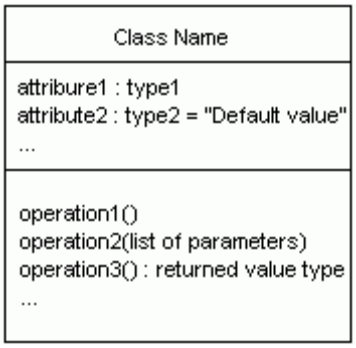
An **attribute** is a property of a class. It describes a range of values that the property may hold in objects of the class. A class may have zero or more attributes.

A one-word attribute name is written in lowercase letter. If the name consists of more than one word, the words are joined and each word other than the first word begins with an uppercase letter. The list of attribute names begins below a line separating them from the class name.



In the class icon, you can specify a type for each attribute's value (string, floating-point, number, integer, boolean or user defined types). Also you can indicate a default value for an attribute.

An **operation** is something that a class can do, or that you (or another class) can do to a class. Like an attribute name, an operation's name is written in lowercase letter. If the name consists of more than one word, the words are joined and each word except the first word begins with an uppercase letter. The list of operations begins below a line separating operations from the attributes.



Also operations may have some additional information. In the parentheses that follow an operation name, you can show the parameter that the operation works on, along with the parameter's type. If the operation is function, then we must specify the type of the returned value.

Other additional features that can be attached to attributes of the class are constraints and notes.
Constraints are free form text enclosed in braces. **Notes** usually are attached to attributes as well as operations, and they give additional information to a class. A note can contain a graphic as well as text.

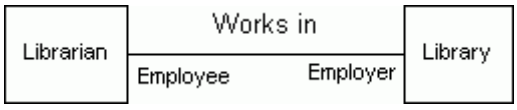
How can we derive classes from interviewing the clients?

In conversations with clients, be alert to the nouns they use to describe the entities in their business. Those nouns will become the classes in your model. Be alert also to the verbs that you hear, because these will constitute the operations in those classes. The attributes will emerge as nouns related to the class nouns.

2. Associations

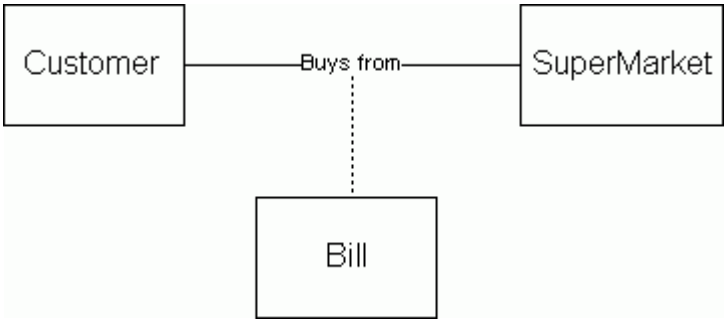
When classes are connected together conceptually, the connection is called an association.

You visualize the association as a line connecting the two classes, with **the name of the association just above the line.**



When one class associates with another, each one usually plays a role within that association. You can show those roles on the diagram by writing them near the line next to the class that plays the role.

Association may be more complex than just one class connected to another. Several classes can connect to one class.
Sometimes an association between two classes has to follow a rule. You indicate that rule by putting a constraint near the association line.



In this case we have an association class.

You visualize association class the same way you show a regular class, and you use dotted line to connect it to the association line.

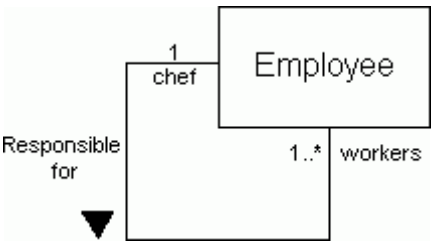
Multiplicity shows the number of objects from one class that relate with a number of objects in an associated class.

One class can be related to another in a

- one-to-one
- one-to-many
- one-to-one or more
- one-to-zero or one
- one-to-a bounded interval (one-to-two through twenty)
- one-to-exactly n
- one-to-a set of choices (one-to-five or eight)

The UML uses an asterisk (*****) to represent ***more*** and to represent ***many***.

Sometimes, a class is in association with itself. This can happen when a class has objects that can play a variety of roles. These associations are called **reflexive associations**.



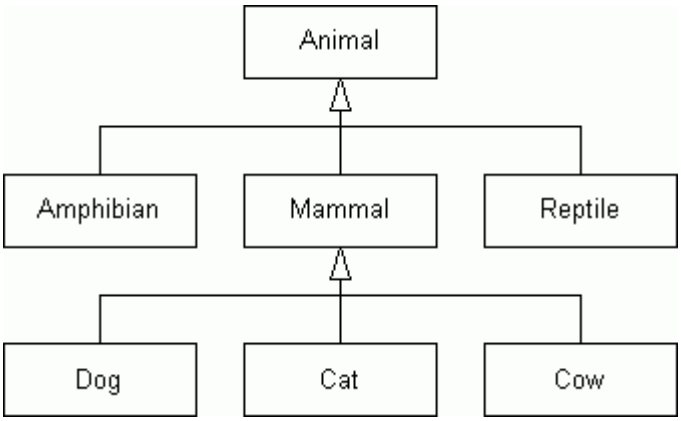
Reflexive association

Without relationships, a class model would be a list of rectangles that represent a vocabulary of system. Relationships show how the terms in the vocabulary connect with one another to provide a picture of the slice of the world you're modeling. The association is the fundamental conceptual connection between classes. Each class in an association plays a role, and multiplicity specifies how many objects in one class relate to one object in the associated class. Many types of associates are possible.

3. Inheritance & Generalization

If you know something about a category of things, you automatically know some things you can transfer to other categories.

If you know something is an animal, you take for granted that it eats, sleeps, has a way of being born, and has a way of getting from one place to another... But imagine that mammals, amphibians and reptiles are all animals. Also cows, dogs, cats... are grouped in category mammals. Object-orientation refers to this as **inheritance**.



An inheritance hierarchy in the animal kingdom

One class (the child class or subclass) can inherit attributes and operations from another (the parent class or superclass). The parent class is more general than the child class. In generalization, a child is substitutable for a parent. That is, anywhere the parent appears, the child may appear. The reverse isn't true, however.

In UML inheritance is represented with a line that connects the parent to a child class, and on the parent's side you put an open triangle.

If we look from the association's side, the inheritance stands for *is a kind of* association.

What should analysts do to discover inheritance?
The analyst has to realize that the attributes and operations of one class are general and apply to perhaps several other classes - which may add attributes and operations of their own. Another possibility is that the analyst notes that two or more classes have a number of common attributes and operations.

Classes that provide no objects are said to be abstract classes. You indicate an abstract class by writing its name in italics.

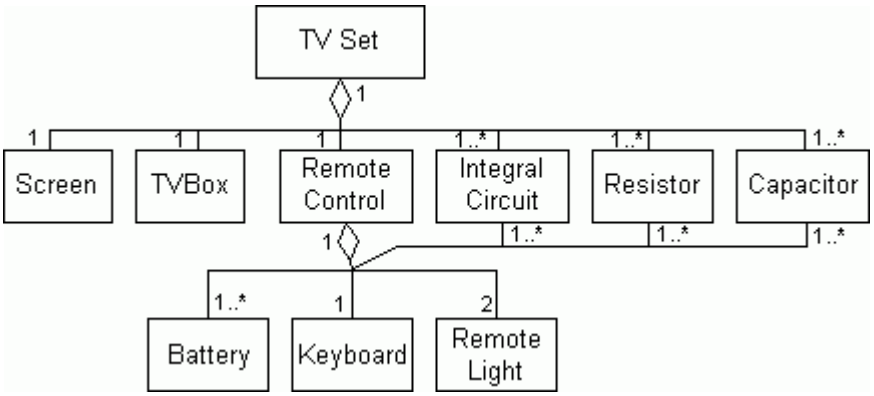
A class can inherit attributes and operations from another class. The inheriting class is the child of the parent class it inherits from. Abstract classes are intended only as basis for inheritance and provide no objects of their own.

4. Aggregations

Sometimes a class consists of a number of component classes. This is a special type of relationship called aggregation. **The components and the class they constitute are in a part-whole association.**

Aggregation is represented as a hierarchy with the "whole" class at the top, and the component below. A line joins a whole to a component with an open diamond on the line near the whole.

Let's take a look at the parts of a TV set. Every TV has a TV box, screen, speaker(s), resistors, capacitors, transistors, ICs... and possibly a remote control. Remote control can have these parts: resistors, capacitors, transistors, ICs, battery, keyboard and remote lights.

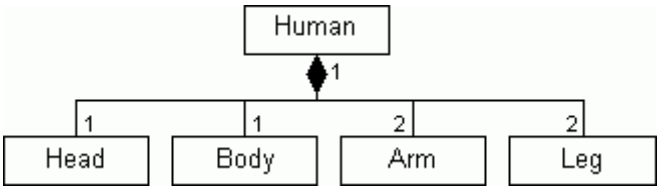


An aggregation association in the TV Set system

Sometimes the set of possible components in an aggregation falls into an OR relationship. To model this, you would use a constraint - the word OR within braces on a dotted line that connects the two part-whole lines.

A composite is a strong type of aggregation. Each component in a composite can belong to just one whole. The symbol for a composite is the same as the symbol for an aggregation except the diamond is filled.

If you examine the human's outside you'll find out that every person has: head, body, arms and legs. This is shown on this picture.



A composite association. In this association each component belongs to exactly one whole.

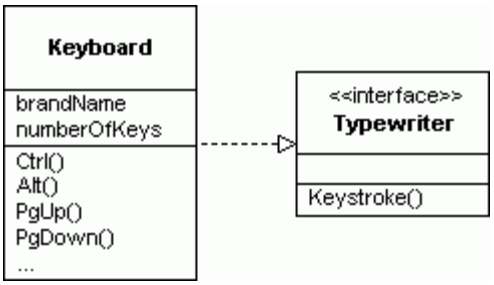
An aggregation specifies a part-whole association. A "whole" class is made up of component classes. A composite is a strong form of aggregation, and a component in a composite can be part of only one whole. Aggregations and composites are represented as lines joining the whole and the component with open and filled diamond, respectively, on the whole side.

5. Interfaces and Realizations

In previous lessons we learned how to refine classes from the interview with client, and relate them with different relationships between them. But it's possible that some of classes are not related to a particular parent, but their behaviors might include some of the same operations with the same signatures. You can code the operations for one of the classes and reuse them in the others.

An interface is a set of operations that specifies some aspect of a class behavior, and it's a set of operations a class presents to other classes.
You model an interface the same way you model a class, with the rectangle icon, but interface has no attributes, only operations. Another way is with a small circle joined with line to a class.

The computer's keyboard is a reusable interface. Its keystroke operation has been reused from the typewriter. The placement of keys is the same as on a typewriter, but the main point is that the keystroke operation has been transferred from one system to another. Also on computer's keyboard you'll find a number of operations that you won't find on a typewriter (Ctrl, Alt, PageUp, PageDown...)



An interface is a collection of operations that a class carries out

To distinguish interfaces from classes, in stereotype construct we put *<<interface>>* or *I* at the beginning of the name of any interface.

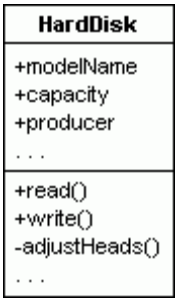
The relationship between a class and an interface is called realization. This relationship is modeled as a dashed line with a large open triangle adjoining and pointing to the interface.

6. Visibility

Visibility applies to attributes or operations, and specifies the extent to which other classes can use a given class's attributes or operations.

Three levels of visibility are possible (last symbols are used in UML classes to indicate different levels of visibility):

- **public level** usability extends to other classes **+**
- **protected level** usability is open only to classes that inherit from original class **#**
- **private level** only the original class can use the attribute or operation **-**



Public and private operations in a HardDisk

7. Class Diagram - Example

Here is a brief description for writing a text document:

Suppose that you're writing a document in some of famous text processing tools, like Microsoft-Word for example. You can start typing a new document, or open an existing one. You type a text by using your keyboard.

Every document consists of several pages, and every page consists of header, document's body or/and footer. In header and footer you may add date, time, page number, file location etc.

Document's body has sentences. Sentences are made up of words and punctual signs. A word consists of letters, numbers and/or special characters. Also in the text you may insert pictures and tables. Table consists of rows and columns. Every cell from table may hold up text or pictures.

After finishing the document, user can choose to save or to print the document.

This is simplified explanation of creating a text document. If we extract the list of nouns form previous text, a following list can be obtained:

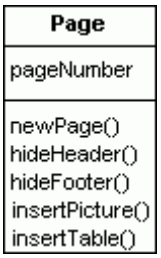
document, text processing tool, Microsoft-Word, text, keyboard, header, footer, document's body, date, time, page number, location of file, page, sentence, word, punctual sign, letter, number, special character, picture, table, row, column, cell, user

Nouns colored in red are candidate classes, and candidate attributes for our model.

Let's start with the document. As you can see this example deals with documents, thus a document will be the central class in our class diagram. Document has several pages. Therefore a ***numberOfPages*** will be one of the attributes for the **Document class**. For the operations we have: ***open()***, ***save()***, ***print()*** and ***new()***. Every document consists of pages. The Page will be also a candidate for the class.

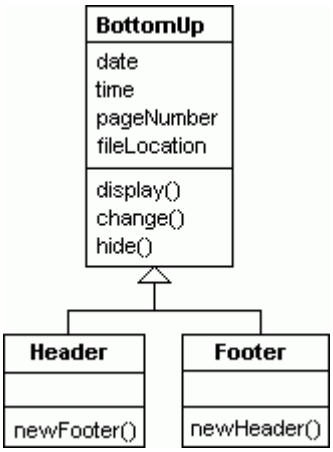
The Document class

The **Page class** will hold ***pageNumber*** as an attribute, and operations allowed here can be: ***newPage()***, ***hideHeader()*** and ***hideFooter()***. Operations for the header and footer tell us that the Header and the Footer can be also a classes.



The Page class

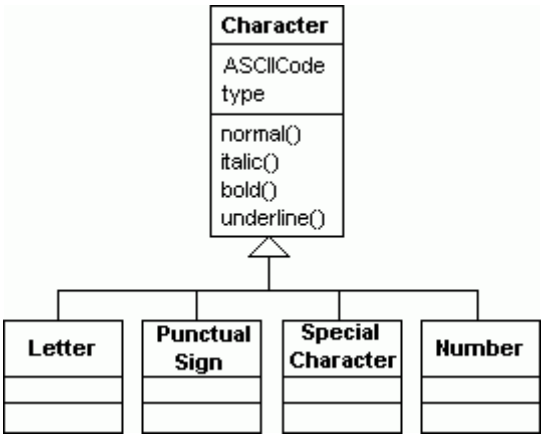
The **Header class** and the **Footer class** have common attributes: ***date***, ***time***, ***pageNumber*** and ***fileLocation***. These attributes are optional for every header or footer and user may configure them. This will guide us that a common class can be introduced. This will be a good time to make an inheritance. Parent class will be **BottomUp** (this name is chosen because headers and footer appear in upper and bottom parts of every page) and will hold common attributes for header and footer, and these operations: ***display()***, ***hide()*** and ***change()***. Header and Footer classes (children of this class) will have only operations: ***newHeader()*** and ***newFooter()***.



The BottomUp class and it's children

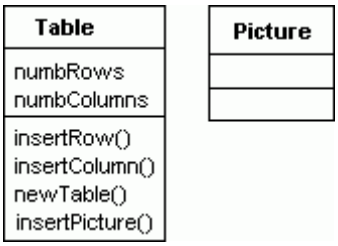
Document's text is made up of sentences. Sentences are made up of words and words are made up of characters. If words are array of characters and sentence is array of words, then a sentence is also an array of characters. Therefore a document's body can be an array of characters. For this purpose to make a document's text we'll use the Character class with its children. The **Character class** will have ***ASCIIcode*** and ***type*** as attributes (type tells the type of the character - normal, italic, bold or underline), and ***normal()***, ***bold()***, ***italic()*** and ***underline()*** as operations. The Character class children

will be: **Letter**, **PunctualSign**, **SpecialCharacter** and **Number**. Also in the document's body, there exist tables and pictures. These are new classes in our class diagram.



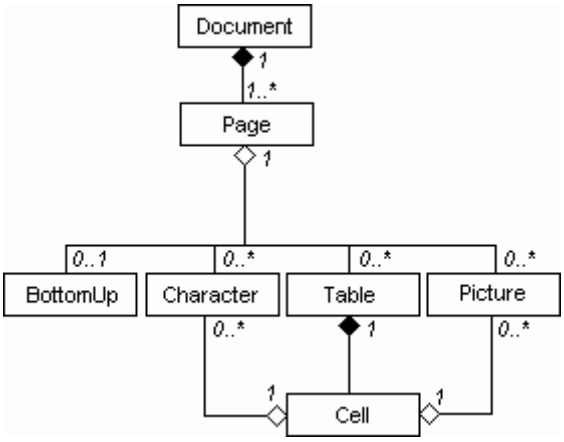
The character hierarchy

The **Table class** has ***numbRows*** and ***numbColumns*** as attributes and ***newRow()***, ***newColumn()*** and ***newTable()*** as operations. Every table consists of one or more cells. And in every cell, text or pictures can be placed.



The Table class and the Picture class

Having all these classes in front of us we can draw out the associations between them and gain the completed class diagram for this problem.



The class diagram for the making of text document

Given class diagram is not finished in detail. If you know more about typing a document, then add information you know to the given diagram. This model can grow and grow, and making all the necessary steps of analysis and design you may reach the point, where a new text processing tool is modeled.

D.2.2 Use Case Diagrams

Class diagrams provide a static view of the classes in a system. Next diagrams provide a dynamic view and show how the system and its classes change over time.

The static view helps an analyst communicate with a client. The dynamic view, helps an analyst communicate with a team of developers, and helps the developers create programs.

Just as the class diagram is a great way to stimulate a client to talk about a system from his or her viewpoint, the use case is an excellent tool for stimulating potential users to talk about a system from their own viewpoints. It's not always easy for users to articulate how they intend to use a system. It's a fact of life that users often know more than they can articulate: **The use case helps break the ice.**

Interviews with users begin in the terminology of the domain, but should then shift into the terminology of the users. The initial results of the interviews should reveal actors and high-level use cases that describe functional requirement in general terms. This information provides the boundaries and scope of the system.

Later interviews with users delve into these requirements more closely, **resulting in use case models that show the scenarios and sequences in detail.** This might result in additional use cases that satisfy inclusion and extension relationships. In this phase it is important to your understanding of the domain, because if you don't understand it well you may create too much use cases and that could impede the analysis process.

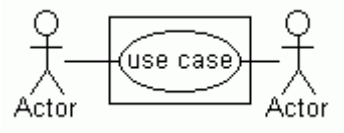
Use cases are a collection of scenarios about system use. Each scenario describes a sequence of events. Each sequence is initiated by a person, another system, a piece of hardware, or by the passage of time. Entities that initiate sequences are called ***actors***. The result of the sequence has to be something of use either to the actor who initiated it or to another actor.

It's possible to reuse use cases. One way - **inclusion**, is to use the steps from one use case as part of the sequence of steps in another use case. Another way - **extension**, is to create a new use case by adding steps to an existing use case.

Next lessons will guide you through representing a use case model and visualizing relationships among use cases. At the end a use case example will be given to you.

1. Introducing a Use Case Model

An actor initiates a use case, and an actor (possibly the initiator, but not necessarily) receives something of value from the use case.



Use Case model

An **ellipse** represents a use case, and a stick figure represents an actor. The initiating actor is on the left of the use case, and the receiving actor is on the right. The actor's name appears just below the actor. The name of the use case appears either inside the ellipse or just below it. An association line connects an actor to the use case, and represents communication between the actor and the use case. The association line is solid, like the line that connects associated classes.

Each use case is a list of scenarios, and each scenario is a sequence of steps. Each scenario of each use case will also have its own page, listing in text from the:

- Actor who initiates the use case
- Preconditions for the use case
- Steps in the scenario
- Post-conditions when the scenario is complete
- Actor who benefits from the use case

Use case diagrams add more power to the requirements gathering. They visualize use cases. They also facilitate communication between analysts and users and between analysts and clients. In a use case diagram, symbol for the use case is an ellipse, actor has a stick figure as a symbol, and association line joins an actor to a use case. The use cases are usually inside a rectangle that represents the system boundary.

2. Relationships Among Use Cases

In lesson Use case diagrams, we mentioned that it is possible to reuse use case diagrams. One way was inclusion, another was extension. These are relationships among use cases, and there are two other kinds of relationships: generalization and grouping.

- **Inclusion**
  Inclusion enables you to reuse one use case's steps inside another use case. To represent inclusion, you use the symbol you used for dependency between classes - dotted line connecting the classes with an arrowhead pointing to the depended-on class. Just above the line, you add a stereotype-the word <<include>>.
-
- **Extension**
  Extension allows you to create a new use case by adding steps to an existing use case. Also here a dotted arrowhead line is used to represent extension, along with a stereotype that shows <<extends>>. Within the base use case, the extension point appears below the name of the use case.
-
- **Generalization**
  Classes can inherit from one another and so can use cases. In use case

inheritance, the child use case inherits behavior and meaning from the parent, and adds its own behavior. You can apply the child wherever you apply the parent. Generalization is modeled the same way you model class generalization - with a solid line that has an open triangle pointing at the parent. The generalization relationship can exist between actors as well as use cases.

-
- **Grouping**
  In some use case diagrams, you might have a multiple of use cases and you'll want to organize them. This could happen when a system consists of a number of subsystems. The most straightforward way is to organize group related use cases into a ***package*** (a tabbed folder).

3. Use Case Diagram - Example

In this example we're going to model out use case diagrams for a self-service machine.

The main functions of self-service machine are to allow a customer to by a product(s) from the machine (candy, chocolate, juice...). Every user that you asked for a set of scenarios happening during usage of machine, can tell you that main use case can be labeled as "Buy a product". Let's examine every possible scenario in this use case. These scenarios would be revealed trough conversations with users.

The "Buy a product" use case

The actor in this use case is customer. This customer wants to buy some of the products offered by the self-service machine. First of all he/she inserts money into the machine, selects one or more products, and machine presents a selected product(s) to the customer. Use case diagram for this scenario can be represented as:

The** Buy a Product **use case diagram

But if we think a little, others scenarios immediately come to mind. It's possible that the self-service machine is out of one or more products, or the machine hasn't the exact amount of money to return to customer.

-Are we supposed to handle with these scenarios?

If we are, then let's think about the moment when the customer inserts money into machine and enters his or hers selection. After this imagine that machine is out of brand. In this case it's preferable to present a message to the customer that machine is out of brand and allow him or her to make another selection or return money back. If incorrect-amount-of-money scenario has happened, then self-service machine is supposed to return original amount of money to the customer.
The precondition here is hungry or thirsty customer, and post-condition is either product from the machine or the returned money.

This is use case scenario from one user's viewpoint (the customer). But there are other users too. A supplier has to restock the machine, and a collector has to collect the accumulated money from the machine. This tells us to create at least two more use cases: "Restock" and "Collect money". Let's look closely at both of them, by interviewing both suppliers and collectors.

The "Restock" use case

Actions that supplier must do every time interval (say, one or two weeks) are: Supplier un-secures the machine, opens the front of the machine, and fills each brand's compartment to capacity. (The supplier may fill each brand according to consuming of article). Then he/she closes the front of the machine and secures it.
The precondition is the passage of the interval, and the post-condition is that the supplier has a new set of potential sales.
This use case diagram is presented on the following picture:

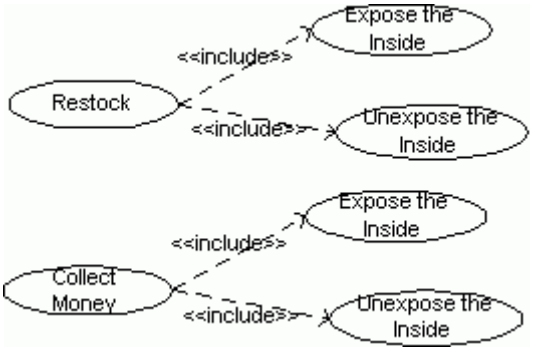
The** Restock **use case diagram

The "Collect Money" use case

Responsible for this use case is collector. Collector may be the same person as the supplier. The steps that collector must do are same as the steps of supplier, but the collector don't deal with products, he/ she deals with money. When the time interval has passed this person collects the necessary amount of money from the machine.
The post-condition here is the money in hands of the collector.
Collect money use case diagram is following:

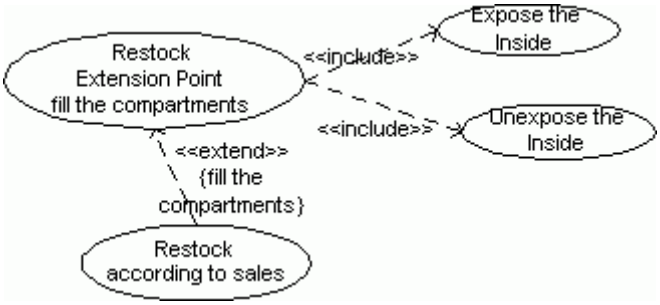
The** Collect Money **use case diagram

The steps of un-securing the machine, front opening, closing and securing the machine are the same that supplier and collector must do. This is a good place to include a use case. Let's combine the "un-secure" and "pull open" steps into use case called "Expose the inside" and the "close machine" and "secure" with use case "Un-expose the inside". By including a use case Restock and Collect use cases may look as this:



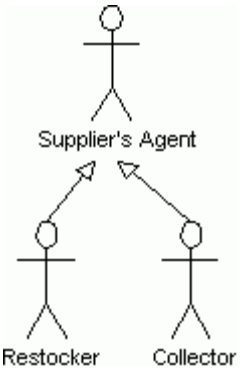
The inclusion process in Restock and Collect Money use cases

The Restock use case could be basis of another use case: "Restock according to sales". Here supplier may fill up brands with new products according to the sale of those products. This is an extension of a use case.
After inclusion and extension the Restock use case can be:



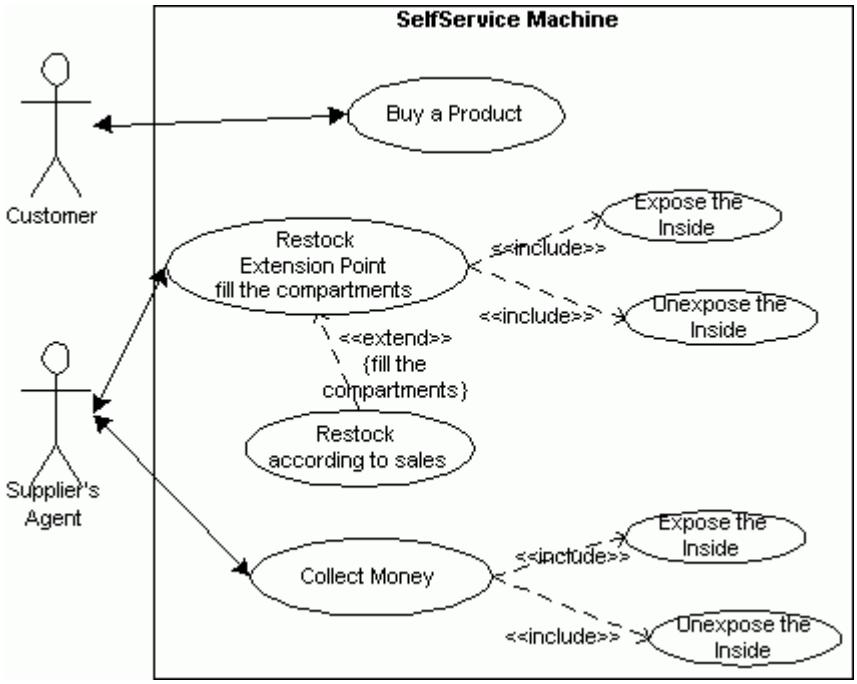
The extension process in Restock use case

Also and generalization may take place to the actors supplier and collector. If both of them are the same person, let's say Supplier Agent, then the restocker and the supplier are both children of the Supplier Agent. This is shown in the next picture:



A generalization between supplier and collector

The following completed use case diagram shows the functional requirements of the system.


SelfService Machine
Customer
Buy a Product
Restock
Extension Point
fill the compartments
Expose the Inside
<<include>>
<<include>>
Unexpose the Inside
<<extend>>
{fill the compartments}
Restock according to sales
Supplier's Agent
Collect Money
<<include>>
Expose the Inside
<<include>>
Unexpose the Inside

DAY 3

D.3.1 State Diagrams

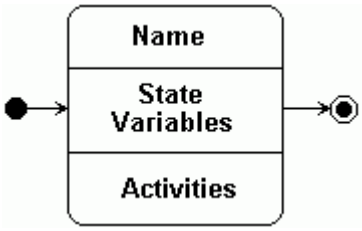
The ***behavioral elements*** show how parts of a UML model change over time. As the system interacts with users and possibly with other systems, the objects that make up the system go through necessary changes to accommodate the interactions. If you're going to model systems, you must have a mechanism to model change. That mechanism in UML is State diagrams.

- When you pull a switch, a light changes its state from off to on.
- When you make keystroke or mouse-movement in screen saver mode on your computer, your computer leaves screen saver mode and returns to working - active mode.

The UML State diagram presents the states an object can be in along with the transitions between the states, and shows the starting point and endpoint of a sequence of state changes.

A state diagram shows the states of a single object.

States in the state diagram are represented with a rounded rectangle, and the symbol for the transition is a solid arrowhead line. A solid circle stands for starting point of a sequence of states, and bull's eye represents the endpoint.



The UML symbols in a state diagram. The top area stands for the state's name, middle area is for state variables, and the bottom is for activities. Transitions are represented with solid arrowhead lines, and circles are for starting and ending state.

Also you can add details to a state icon by dividing it to a three areas. The top area holds the name of the state (this name you must supply whatever the class is divided or not), the middle area hold state variables (timers, counters, dates...), and the bottom area hold activities (***entry***-what happens when the system enters the state, ***exit***-what happens when the system leaves the state, ***do***-what happens when the system is in the state).

1. State Details and Transitions

You can indicate an event that causes a transition to occur (***a trigger event***) and the computation (***the action***) that executes and makes the state change happen.

To add events and actions you write them near the transition line, using a slash to separate a triggering event from an action.

Sometimes an event causes a transition without an associated action, and sometimes a transition occurs because a state completes an activity (rather than because of an event). This type of transition is called ***triggerless transition***.
Also possible is a ***guard condition***. When it's met, the transition takes place. (For example screen saving mode is activated when time interval of n idle minutes has passed).

Some of the states may be more complex that is represented with a rounded rectangle. It is possible to have states inside one state. These states are called ***substates*** (sequential and concurrent).

Sequential substates come in sequences of states which occur one after another. Concurrent substates must consist of two or more sequential substates which occur at the same time. This is represented with dotted line between the concurrent states.

The UML supplies a symbol that shows that a composite state remembers its active substate when the object transitions out of the composite state. The symbol is Ⓗ connected by a solid line to the remembered substate, with an arrowhead that points to the substate.

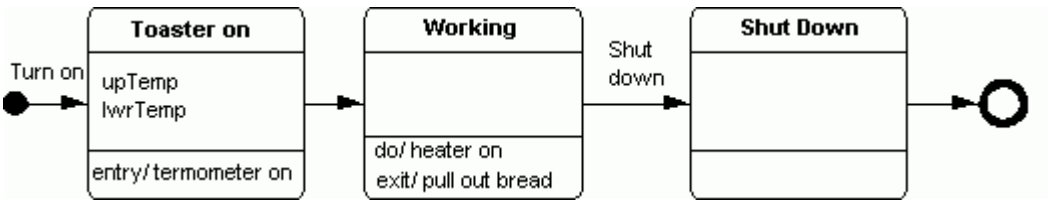
A message that triggers a transition in the receiving object's state diagram is called a ***signal***. In the object-oriented world, sending a signal is the same as creating a signal object and transmitting it to the receiving object. The signal object has properties that are represented as attributes. Because a signal is an object, it's possible to create inheritance hierarchies of signals.

2. State Diagram - Example

Suppose you're designing a toaster. You would build a plenty of UML diagrams, but here only state diagrams will be of our interest.

- What are the steps of making a toast?

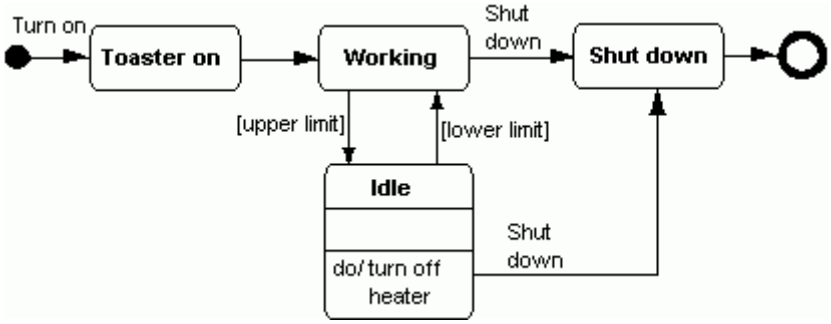
First of all we must turn on the toaster, put in the bread and wait for several minutes to bake it. The initial state diagram is shown below:



The initial state diagram for making a toast

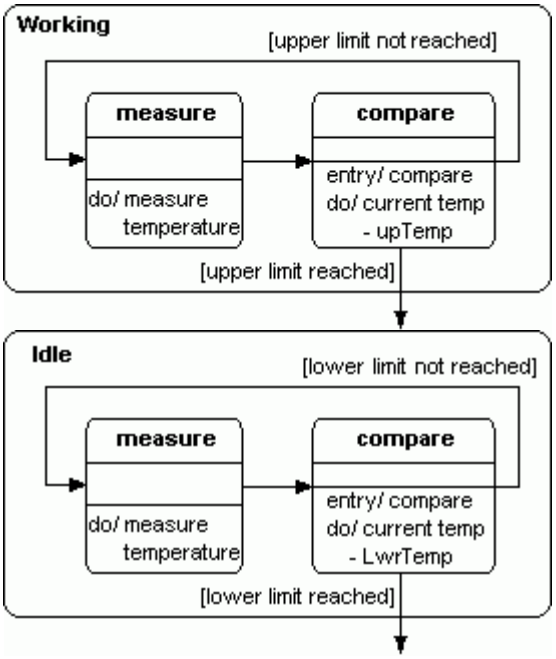
But this is not the final state diagram. To prevent burning out the bread, heater of the toaster must produce heat in temperature interval (upper and lower temperature limits). For this purpose thermometer measures the temperature of heater, and when the upper limit of temperature is reached then heater must go into idle state. This state resists until

heater's temperature decreases to lower limit, and then working state is again aimed. With this new state, extended state diagram will be:



The extended state diagram for making a toast

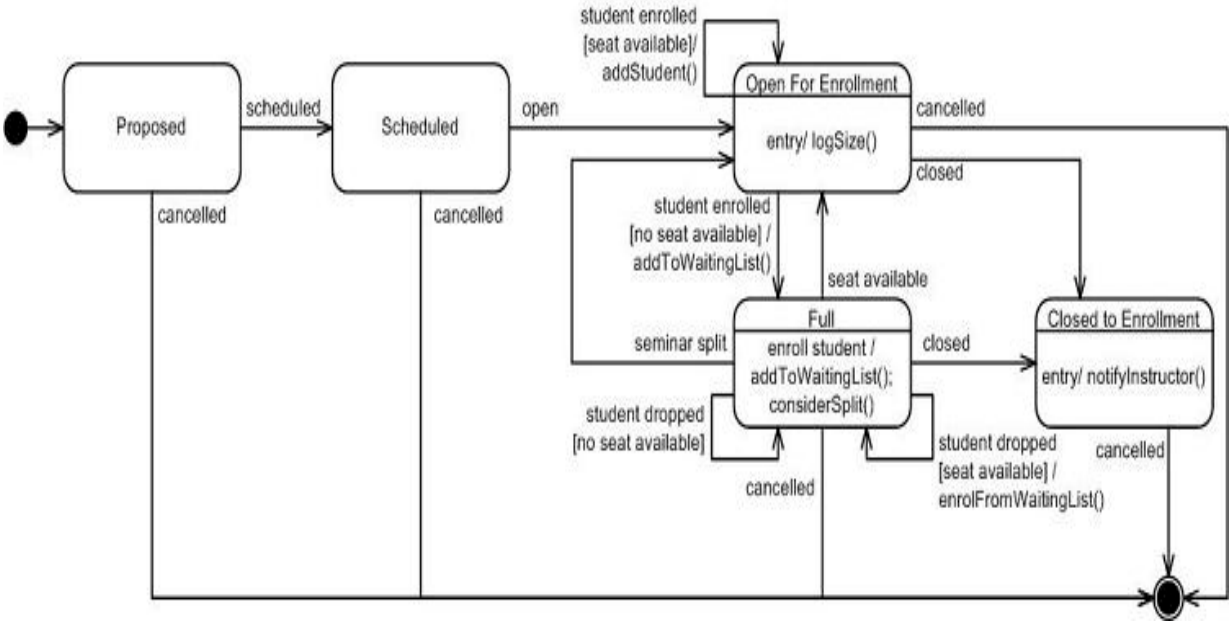
Transition between working and idle state is not presented in details. To do this, substates are added.



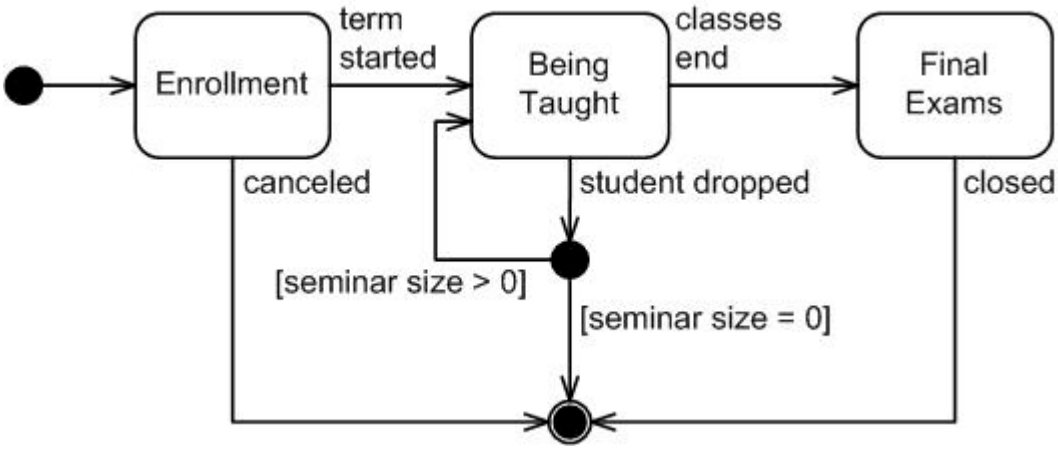
Substates in** Working **and** Idle **states

Substates in working and idle states are very similar. both had measure and compare states, but differentiates in process of temperature comparison. Working state must compare current temperature with upper temperature limit (if it is reached, working state goes into idle state), and idle state compares current temperature with lower temperature limit (idle state is replaced with working state when temperature falls under lower limit).

A seminar during registration.



Top-level state machine diagram.



It's necessary to have state diagrams because they help analysts, designers and developers understand the behavior of the objects in a system. Developers, in particular, have to know how objects are supposed to behave because they have to implement these behaviors in software. It's not enough to implement an object: Developers have to make that object do something.

D.3.2 Sequence Diagrams

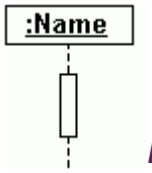
In previous lesson we learned state diagrams, which focus on the states of an object. But it's necessary to have communication between objects. The UML sequence diagram goes the next step and shows how objects communicate with one another over time. In this expanded field of view, you'll include an important dimension: ***time***. The key idea here is that interactions among objects take place in a specified sequence, and the sequence takes time to go from beginning to end.

The sequence diagram consists of ***objects*** represented in the usual way - as named rectangles (with the name underlined), ***messages*** represented as solid-line arrows, and ***time*** represented as a vertical progression.

Let's take a close look at this parts of sequence diagrams.

Objects

The objects are laid out near the top of the diagram from left to right. They're arranged in any order that simplifies the diagram. Extending down-forward from each object is a dashed line called the object's ***lifeline***. Along the lifeline is narrow rectangle called an ***activation***. The activation represents an execution of an operation the object carries out. The length of the rectangle signifies the activation's duration. This is shown on next picture.



Representing an object in a sequence diagram

Messages

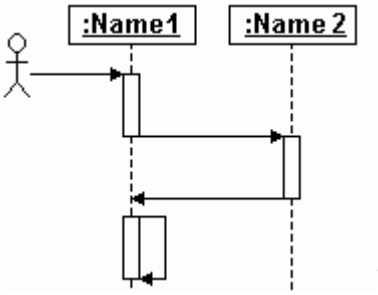
A message goes from one object to another goes from one object's lifeline to the other object's lifeline. An object can send a message to itself-that is, from its lifeline back to its own lifeline. This is called ***recursion***.

Following table lists all of possible messages existing in the UML sequence diagrams, with their graphical representations:

simple	This is a transfer of control from one object to another.	
synchronous	If an object sends a synchronous message, it waits for an answer to that message before it proceeds with its business.	
asynchronous	If an object sends an asynchronous message, it doesn't wait for an answer before it proceeds.	

Time

The diagram represents time in the vertical direction. Time starts at the top and progresses toward the bottom. A message that's closer to the top occurs earlier in time than a message that's closer to the bottom.



The essential symbol set of the sequence diagram, with the symbols working together.

The actor-symbol initiates the sequence, but the stick figure isn't part of the sequence diagram symbol set.

In a sequence diagram, the objects are laid out from left to right across the top. Each object' lifeline is a dashed line extending downward from the object. A solid line with an arrowhead connects one lifeline to another, and represents a message from one object to another. Time starts at the top an proceeds downward. Although an actor typically initiates the sequence, the actor symbol isn't part of the sequence diagram's symbol set.

1. Ways of Creating Sequences

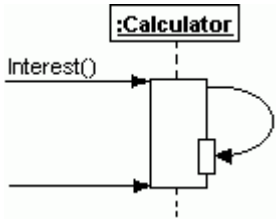
As use case diagram can show either an instance (one scenario) of a use case, or it can be generic and incorporate all of the use case's scenarios, and sequence diagrams can be ***instance*** or ***generic*** sequence diagrams. Generic sequence diagrams often provide opportunities to represent ***if*** statements and ***while*** loops. Enclose each condition for an "if" statement in square brackets. Do the same for the condition that satisfies a "while" loop, and prefix the left bracket with an asterisk.

It's possible in sequence diagrams to create an object. When this occurs, you represent a created object in the usual way - as a named rectangle.

The difference is that you don't position it at the top of the sequence diagram as you do with the other objects. Instead, you position it along the vertical dimension so that its location corresponds to the time when it's created. The message that creates the object is labeled ***Create()***.

Sometimes an object has an operation that invokes itself. This is called ***recursion***.

Suppose one of the objects in your system is a calculator, and suppose one of its operations computes interest. In order to compute compound interest for a timeframe that encompasses several compouding periods, the object's interest-computation operation has to invoke itself a number of times. Sequence diagram for this is following:



Representing recursion in a sequence diagram

2. Sequence Diagram - Example

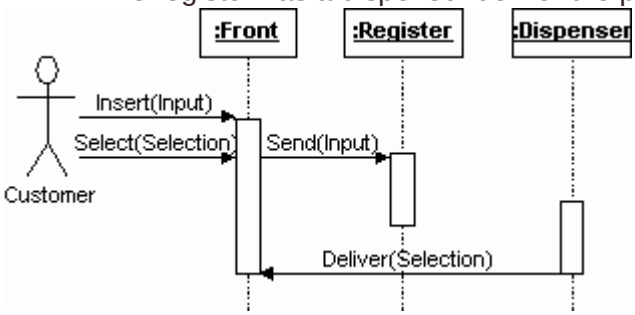
In this example, a sequence diagram for one of the previous examples - Self-service machine will be given. See Use Case Diagram – Example.

Let's assume that in the self-service machine, three objects do the work we're concerned with:

- **the front** the interface the self-service machine presents to the customer
- **the money register** part of the machine where moneys are collected
- **the dispenser** which delivers the selected product to the customer

The *instance sequence diagram* may be sketched by using these sequences:

1. The customer inserts money in the money slot
2. The customer makes a selection
3. The money travels to the register
4. The register checks to see whether the selected product is in the dispenser
5. The register updates its cash reserve
6. The register has a dispenser deliver the product to the front of the machine



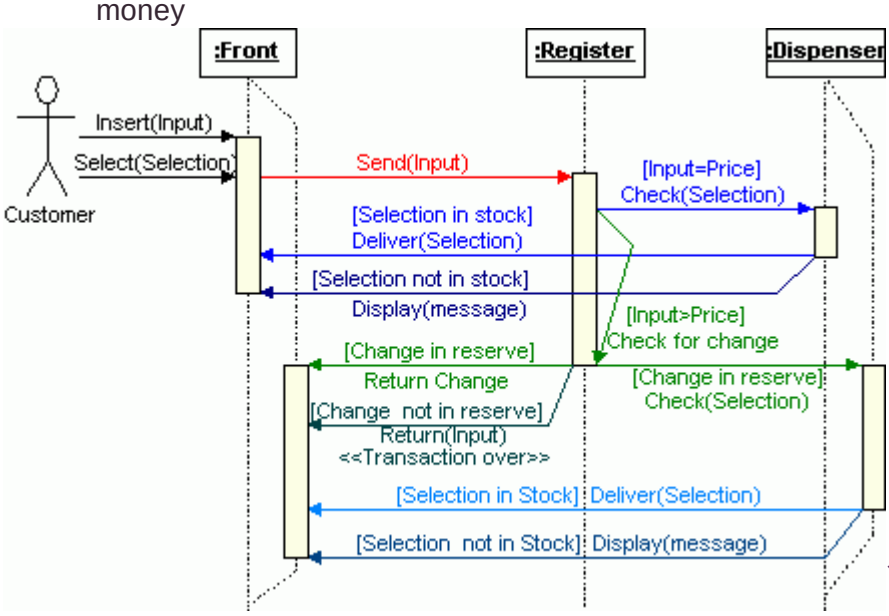
The "Buy a product" scenario. Because this is the best-case scenario, it's an instance sequence diagram

But, when use case diagrams were developed, two additional scenarios were introduced to represent out-of-product case and incorrect-amount-of-money case. If you consider all of a use case's scenarios when you draw a sequence diagram, you create a *generic sequence diagram*.

The out-of-product scenario will produce the following sequences:

1. After selecting a sold-out brand, the "SOLD OUT" message flashes
2. Prompt for another selection must be displayed
3. The customer has the option of pushing a button that returns money

4. If the customer selects an in-stock brand, everything proceeds as in the best-case scenario if the input amount is correct. If not, the machine follows the incorrect-amount-of-money scenario
5. If the customer selects another sold-out brand, the process repeats until the customer selects an in-stock brand or pushes a button that returns his or her money

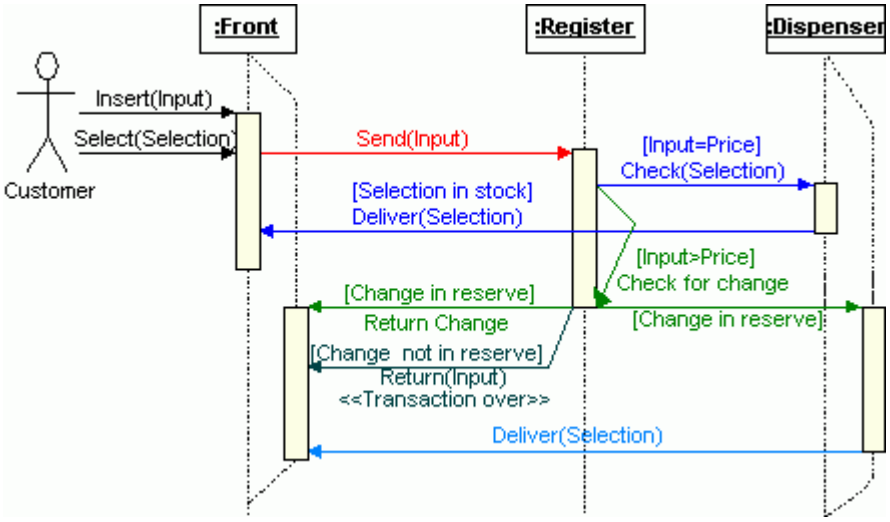


The "Out of product" scenario

Make attention on "fork" of control in the message, caused by if conditions. Because each path goes to the same object, the fork causes a "branch" of control in the receiving object's lifeline, separating the lifeline into separate paths. At some point in the sequence, the branches in the message merge, as do the forks in the lifeline.

For the incorrect-amount-of-money scenario, the list of sequences is given in the following table:

1. The register checks customer's input with the price of the product
2. If the amount is greater then the price, the difference is calculated, and register checks its cash reserve
3. If the difference is present in the cash reserve, the register returns the change to the customer and everything proceeds as before
4. Otherwise, the register returns the input amount and displays a message that prompts the customer for the correct amount
5. If the amount is less than the price, the register does nothing and the machine waits for more money



The "Incorrect amount of money" scenario

D.3.3 Collaboration Diagrams

Collaboration diagrams like the sequence diagrams, shows how objects interact. It shows the objects along with the messages that travel from one to another. But, if sequence diagram does that, why UML need another diagram? Don't they do the same thing?

The sequence diagram and the collaboration diagram are similar. They're semantically equivalent, that is, the present the same information, and you can turn a sequence to a collaboration diagram and vice versa. **The main distinction between them is that the sequence diagram is arranged according to time, the collaboration diagram according to space.**

A collaboration diagram is an extension of object diagram. In addition to the associations among objects, collaboration diagram shows the messages the objects send each other.

Messages among objects are represented with arrows that points to receiving object, near the association line between two objects. A label near the arrow shows what the message is. The message typically tells the receiving object to execute one of its operations. A pair of parentheses ends the message. Inside the parentheses, you put the parameters the operation works on.

Collaboration diagrams can be turned into sequence diagrams and vice versa. Thus, you have to be able to represent sequence information in a collaboration diagram. To do this you must know that:

Add a number to the label of a message, with the number corresponding to the message's order in the sequence. A colon separates the number from the message.

1. Writing Collaboration Diagrams

You can show an **object's change of state** in a collaboration diagram. You can also show **conditions** the same way you represent them in sequence diagram.

In the object rectangle, indicate the state of the object. To the diagram, add another rectangle that stands for the object and indicate the changed state. Connect the two with the dashed line and label the line with a <<become>> stereotype.
You put the condition inside a pair of square brackets, and the condition precedes the message-label. The important thing is to coordinate the conditions with the numbering.

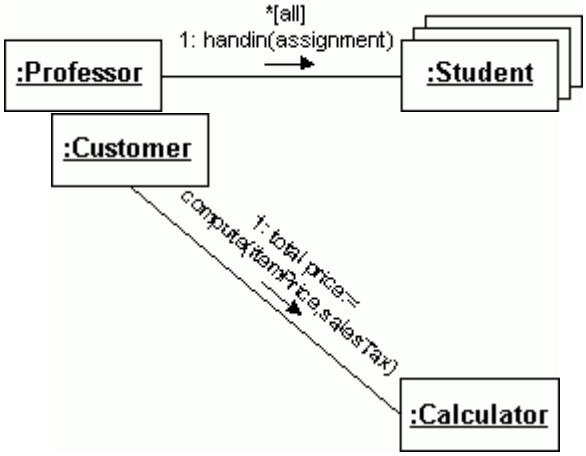
To get closer to this concepts please refer to the Collaboration Diagram - Example.

Also you can present **object creation**.

In object's creation process you add a <<create>> stereotype to the message that creates the object.

In the collaboration diagram, you can represent **multiple objects** and **returned results**.

The representation of multiple objects is a stack of rectangles extending "backward". You add a bracketed condition preceded by an asterisk to indicate that the message goes to all objects.
Returned results are written as expression that has a name of the returned value on the left, followed by ***:=***, followed by the name of the operation, and a list of possible attributes that operation can accept, to produce the result. The right side of the expression is called a ***message-signature***.



A professor asks a group of students to hand in an assignment.

A message can be a request for an object to perform a calculation and return a value.

In some interactions, a specific object controls the flow. This **active object** can send messages to passive objects and interact with other active objects. Also you might run into an object sending a message only after several other (possibly nonconsecutive) messages have been sent. That is, the object must **synchronize** its message with a set of other messages.

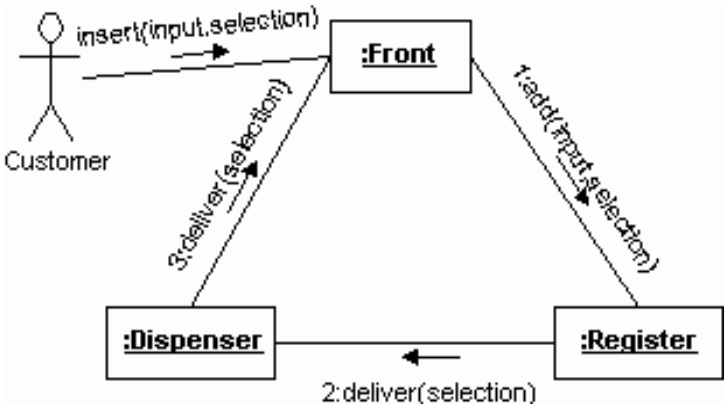
The collaboration diagram represents an active object the same way as other, except that its border is thick and bold. A comma separates one list-item from another, and the list ends with a slash.

2. Collaboration Diagram - Example

Self-service machine again will be target for this example. In previous example (See, Sequence Diagram - Example) the sequence diagram for this system was built. This is good time to try the ways of converting sequences into collaboration diagrams.

The best-case scenario consists of several steps:

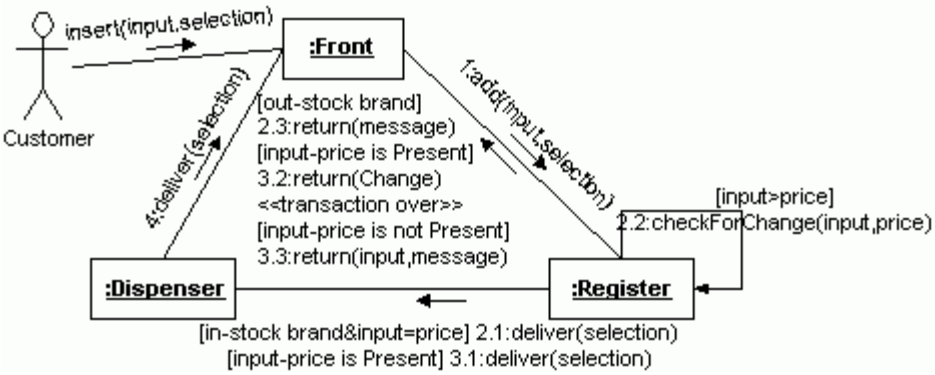
1. The customer inserts money in the machine and makes selection of one or more product(s) present in the machine.
2. When the register gets the money (in this case customer puts the correct amount of money, and there is allways a product in the selected brand), the selected product is delivered to the dispenser.
3. The dispenser delivers the product to the front of the machine, and the customer gets that product.



But when we examined this machine and its work in detail, new scenarios were introduced: incorrect-amount-of-money and out-of-product scenario. While building the sequence diagrams, conditions were introduced to display the work of the machine. Here conditions will be used too.

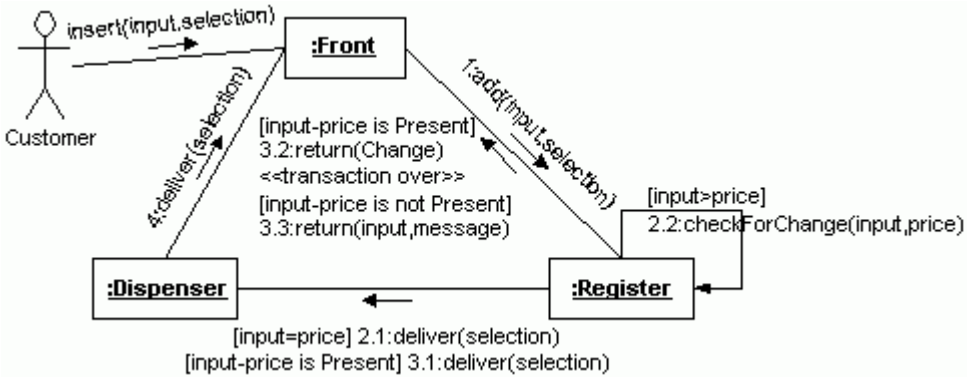
You put the condition inside a pair of square brackets, and the condition precedes the message-label. The important thing is to coordinate the conditions with the numbering. When you're modeling *if*-statement there are two possible conditions. Messages for these conditions have a same number, and you add a decimal point and another number. This is called **nesting**.

In out-of-product scenario, the machine has to display an out of product message, and prompt the customer for another product or return the money back. If customer makes another selection this process must be repeated. Also a correct-amount-of-money scenario may be introduced. Collaboration diagram for this case is shown on the picture:



As you can see, from this picture two possible scenarios may happen. *Out-of-product* and *Product-in* are that two possibilities. If we assume that this happened after message *1:add(input,selection)*, the number for this messages will be ***2***, and we add ***1*** for the Product-in and ***2*** for the out-of-product scenarios, after the decimal point.

What happens when the machine doesn't have the correct change? The machine has to display an out of change message, return the money, and prompt the customer for the correct change. In effect, the transaction is over. When the machine has the correct change, it returns the change to the customer and delivers selected product.
Two branches that are produced from nesting process are numbered with ***2.1*** and ***2.2***. And three possibilities in condition-message 3 will be:***3.1, 3.2*** and ***3.3***. Collaboration diagram is shown on the following picture:



That’s all for this example, and for this day. All you have to do now is answering the questions given in Questions & Answers and Workshop part from this day. Next day will introduce to you the last three UML diagrams.

DAY 4

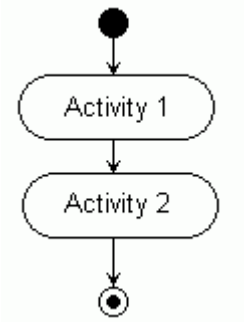
D4.1 Activity Diagrams

If you've ever taken an introductory course in programming, you've probably encountered the ***flowchart***. The flowchart shows a sequence of steps, processes, decision points and branches. Novice programmers are encouraged to use flowcharts to conceptualize problems and derive solutions.

The UML activity diagram is much like the flowcharts of old. It shows steps (called, appropriately enough, ***activities***) as well as decision points and branches. It's useful for showing what happens in a business process or an operation. You'll wind it an integral part of system analysis.

First and foremost, an activity diagram is designed to be a simplified look at what happens during an operation or a process. It's an extension of the state diagram. The state diagram shows the states of an object and represents activities as arrows connecting the states. The activity diagram highlights the activities.

Each activity is represented by a rounded rectangle - narrower and more oval-shaped than the state icon. The processing within an activity goes to completion and then an automatic transmission to the next activity occurs. An arrow represents the transition from one activity to the next. Also an activity diagram has a starting point represented by filled-in circle, and endpoint represented by a bull's eye.



Transition from one activity to another in the Activity Diagram

In next lesson will be explained ways of creating an activity diagram, rules between activities, and at the end an example for this diagram will be given to you.

The activity diagram is an extension of the state diagram. State diagrams highlight states and represent activities as arrows between states. Activity diagrams put the spotlight on the activities. Each activity is represented as a rounded rectangle, more oval in appearance than the state icon. The activity diagram uses the same symbols as the state diagram for the startpoint and the endpoint.

1. Building an Activity Diagram

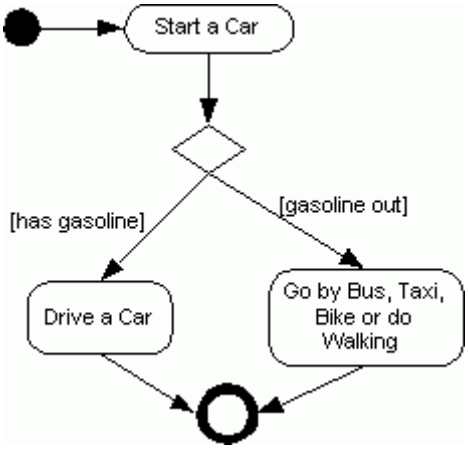
This lesson explains ways of representing decisions, concurrent paths, signals and swimlanes in the activity diagrams.

Decisions

Decision point can be represented in two ways, it's all to you to make decision what way to use in your activity diagrams.

One way is to show the possible paths coming directly out of an activity. The other is to have the activity transition to a small diamond and have the possible paths flow out of the diamond. Either way, you indicate the condition with a bracketed condition statement near the appropriate path.

Imagine that you have to go to the work. You get your car, put the key in the ignition, and there are two possible situations: your car will start or it will not start its engine. These possible cases will produce two other activities: drive a car, or go by a bus, taxi, bike, or go walking. This scenario is shown on this picture (make attention of two ways showing a decision):

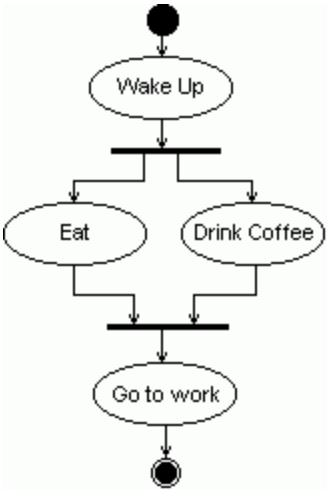


Activity diagram showing two ways of decision

Concurrent paths (fork, join)

When you modeling activities, very frequently you'll have a occasion to separate a transition into two separate paths that run at the same time (concurrently), and the come together.

Split is represented by a solid bold line perpendicular to the transition and shows the paths coming out of the line. To represent the merge, show the paths pointing at another solid bold line.



Activity diagram representing a transition split into two paths that run concurrently and then come together

Signals

During a sequence of activities, it's possible to send a signal. When received, the signal causes an activity to take a place.

The symbol for sending a symbol is a convex pentagon, and the symbol for receiving a signal is a concave polygon.

Sending and receiving a signal

Swimlanes

The activity diagram adds the dimension of visualizing roles. To do that, you separate the diagram into parallel segments called ***swimlanes***. Each swimlane shows the name of a role at the top, and represents the activities of each role. Transitions can take place from one swimlane to another.

It's possible to combine the activity diagram with the symbols from other diagrams and thus produce a ***hybrid diagram***.

Following lesson will give you a written example for an activity diagram.

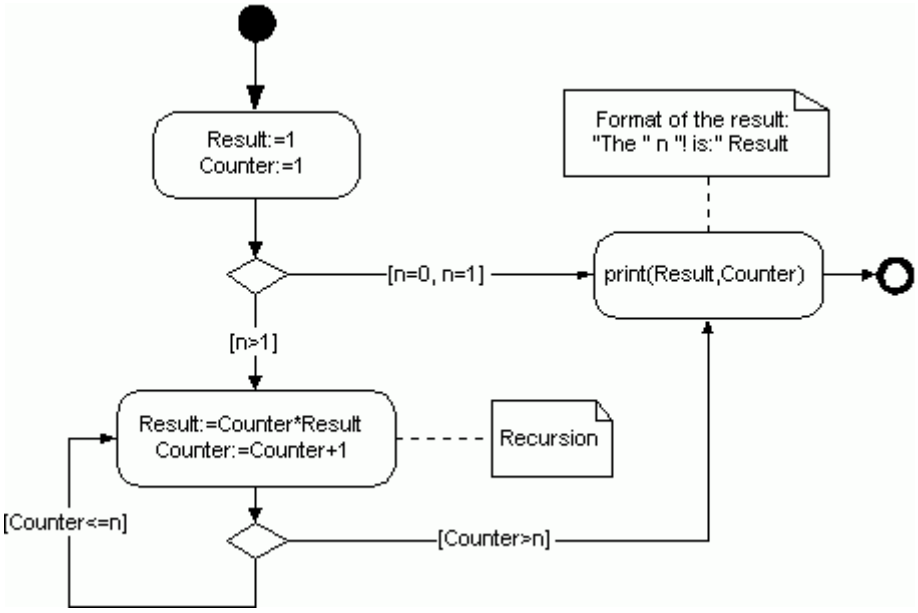
2. Activity Diagram - Example

This example will deal with mathematics. Sometimes (when calculating combinations) you'll need to calculate a factorial of n - ***n!***.
The formula for this is *n!=n*(n-1)*(n-2)*...*2*1.*

Let's dive into the problem. From definition of factorials we have 0!=1 and 1!=1. For the rest of the numbers we must use the given formula. In computer programming this problem is written with recursion. Here also when building an activity diagram recursion will be introduced. You might call the operation *computeFact(n)*. You'll need a counter to

keep track of whether or not the operation has reached the *n*th factoriel, a variable that keep a track of your computations, and two more to store the values of 0! and 1!.

The complete activity diagram is shown on next picture:



If you're dealing with computer programming you will conclude that this activity diagram is very similar with a procedure (function) that computes factorials.

D4.2 More About Interfaces and Components

In previous lessons, you learned about diagrams that deal with conceptual entities. In next lessons, you're going to learn about a UML diagram that represents a real-world entity: ***software component***.

But what is a software component?
A software component is a physical part of a system. It resides in a computer, not in the mind of an analyst.

What's the relationship between a component and a class?
Think of a component as the software implementation of a class. The class represents an abstraction of a set of attributes and operations. And one component can be implementation of more than one class.

You model components and their relationships so that:

1. Clients can see the structure in the finished system
2. Developers have a structure to work toward
3. Technical writers who have to provide documentation and help files can understand what they're writing about
4. You're ready for reuse

When you deal with components, you have to deal with their interfaces. Interface is the object's "face" to the outside world, so that other objects can ask the object to execute its operations, which are hidden with encapsulation.

An interface is a set of operations that specifies something about a class's behavior. It is a set of operations the a class represent to other classes. For you as a modeler, this means that the way you represent an interface for a class is the same as the way you represent an interface for a component. As is the case with a class and its interface, the relation between a component and its interface is called realization. See, Interfaces & Realizations in Day 2.

You can replace one component with another if the new component conforms to the same interfaces as the old one. You can reuse a component in another system if the new system can access the reused component trough that component's interfaces.

As you progress in your modeling career, you'll deal with three kinds of components:

1. ***Deployment components***, which form the basis of executable systems (DLL's, executables, ActiveX controls, JavaBeans)
2. ***Work product components***, from which deployment components are created (data files and source code files)
3. ***Execution components***, created as result of a running system

Instead of representing a conceptual entity such as a class or a state, a component diagram represents a real-world item - a software component. Software components reside in computers, not in the minds of analysts.
A component is accessible trough its interface. The relation between a component and its interface is called realization. When one component access the services of another, it uses an import interface. the component that realizes the interface with those services provides an export interface.

D.3.3 Component Diagrams

A component diagram contains components, interfaces and relationships. Also other types of symbols that you've already learned can also appear in a component diagram.

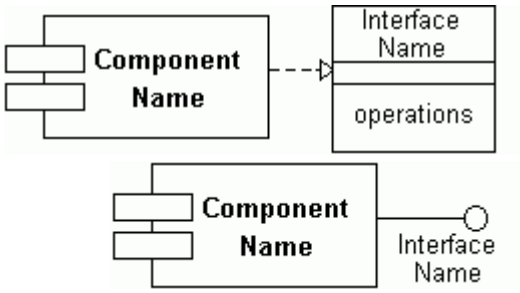
The component diagram's main icon is a rectangle that has two rectangles overlaid on its left side. You put the name of the component inside the icon. The name is string. If the component is a member of a package, you can prefix the component's name with the name of the package.

The component icon

A component and the interfaces are represented in two ways.

One way is to show the interface as a rectangle that contains interface-related information. It's connected to the component by the dotted line and empty triangle that

visualizes realization.
Other way to represent interface is as a small circle connected to the component by a solid line, which represents a realization relationship.



Interface can be represented as a rectangle containing information, connected to the component by a realization arrow

Interface is a small circle, connected to the component with a solid line, which stands for realization

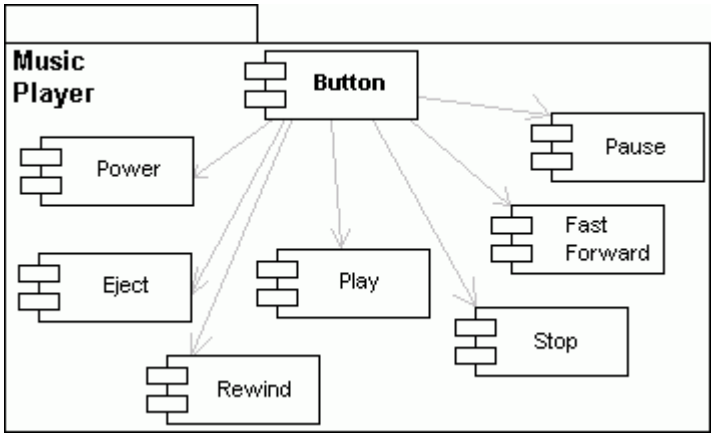
1. Component Diagram - Example

Suppose that we need to build up software for playing music from a CD-ROM Drive. A visual programming language might be used (VisualBasic or Delphi for example). If language supports multimedia controls, than we can use its components an reprogram them if necessary, or we can program new components. One possible graphical design for our player might be:

As you can see this UML Music Player needs these controls:

- play
- stop
- eject
- pause
- fast forward
- rewind
- power

These controls will be realized by ***buttons***, thus we'll have a button performing these controls. If we look at buttons as separate components, we can draw out a component UML diagram. This is shown on the following picture:



The component diagram for the MusicPlayer.

All the components shown on the previous diagram belongs to one global component - Button, but actions they perform are different. We must obtain these actions by programming them.

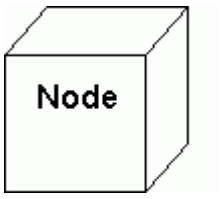
D.4.4 Deployment Diagrams

We reached the end of the UML diagrams. Now it's perfect time to look at the hardware. As you can see, we moved from items that live in analysis, to components that live in computers, to hardware that lives in the real world.

Hardware, of course, is a prime topic in a multi-component system. A solid blueprint for hardware deployment is essential to system design. The UML provides you with symbols for creating a clear picture of how the final hardware setup should look.

The main hardware item is a ***node***, a generic name for any kind of computing resource. Two types of nodes are possible. A ***processor*** is a node that can execute a component, and a ***device*** that can't execute a component. A device typically interfaces in some way with the outside world.

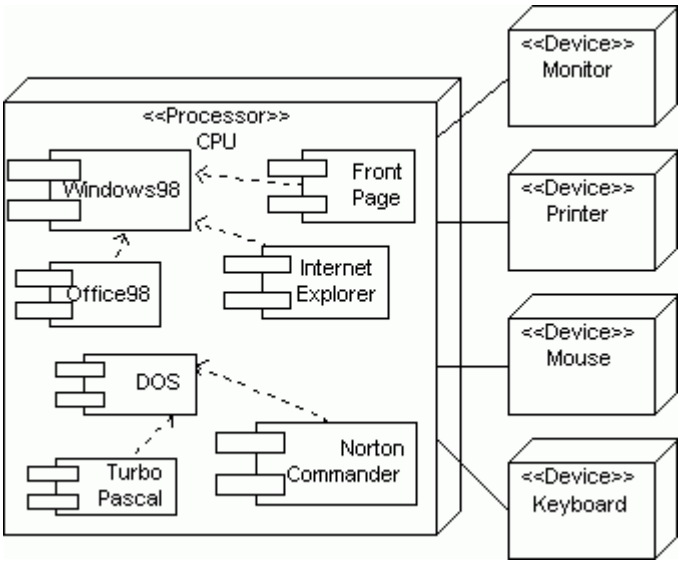
In the UML a cube represent a node. Node has its name, and you can use a stereotype to indicate the type of resource it is. If a node is a part of package, then and the name of package precedes the name of the node. A line joining the two cubes represents a connection between two nodes. You can use a stereotype to provide information about the connection.



The cube is representation of a node in the UML

Also every node deploys some of software components in the system. To indicate deployed components, you show them in dependency relationships with a node.

Let's build a deployment diagram for a home computer system. Computer consists of these hardware elements: CPU, monitor, printer, mouse, keyboard. And installed software components: Windows98, Office98, InternetExplorer4, FrontPage, DOS, NortonCommander, TurboPascal.



The main node is CPU which consists of all the software components with their relationships. Other hardware elements are devices connected to the main processor unit.

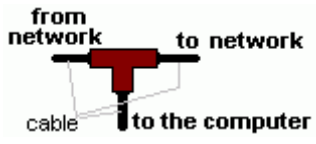
You may extend this diagram by adding a modem and connection to the Internet. Try to redraw the diagram with this extension.

The UML deployment diagram provides a picture of how the physical system will look when it's all put together. A system consists of nodes (represented by a cube) with line connecting them, called connections. There are two types of nodes: a processor (can execute a component) and a device (interface with the real world). Deployment diagrams are useful for modeling networks.

2. Deployment Diagram - Example

This example will describe you the thin Ethernet network. If you're familiar with computer networks this one will be easy to you, but if not then try to understand the following explanation:

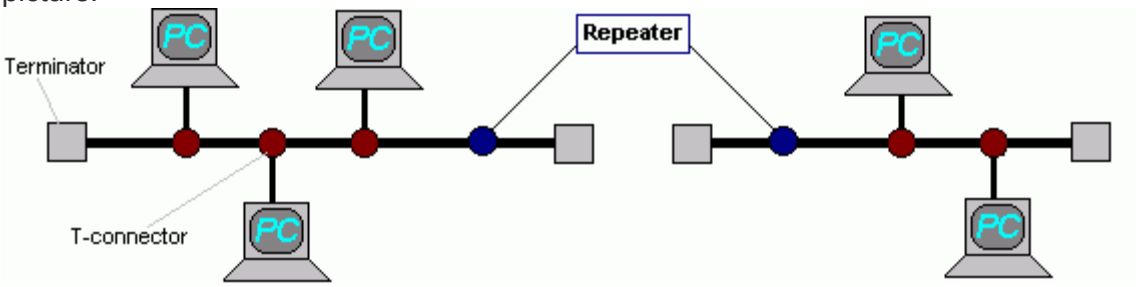
The thin Ethernet is a popular type of network. It is used in local places, for example, rooms or buildings, where connecting cable can be placed. There are some mathematics calculations for the length of the cable for the whole network, and length of cable from computer to computer. The calculation for the deployment diagram is not important.



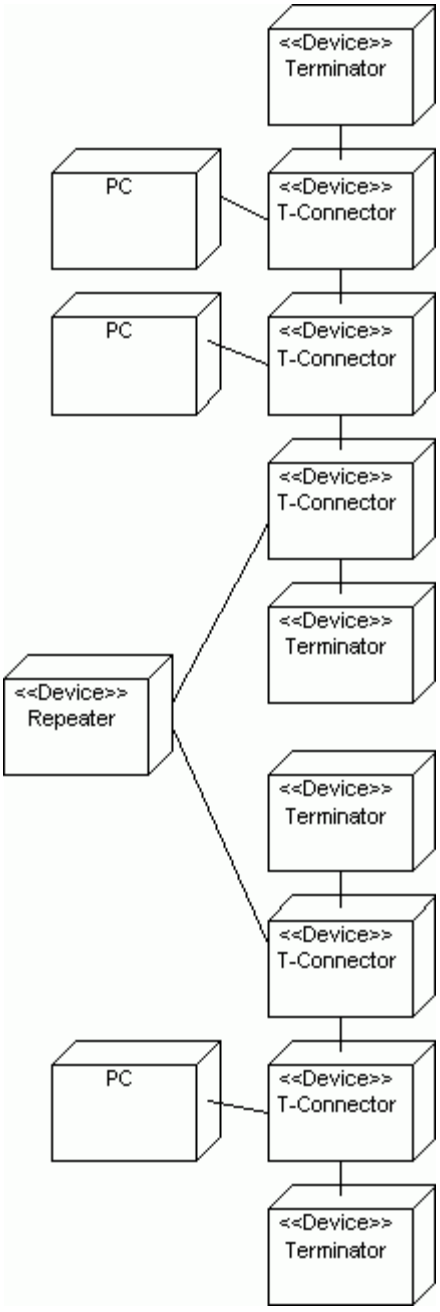
Computers connect to a network cable via connection devices called T-connectors, or sometimes vampire-taps are used. T-connector has three connecting points. Its form is similar with T-letter and that,s why their name came from. Network cable goes from one end-point to the other end-point of the connector, and the third end-point has a cable going to the

computer.

Because length of the network is limited, ending points of the cable must have devices called terminators, which gave infinite resistance to the end of cable. Also one network segment may join another via a repeater. The repeater is a device which amplifies the signal to reduce losing of signal. The thin Ethernet is represented on the following picture:



When we get familiar with the thin Ethernet network concept let's draw the deployment diagram for it. This diagram will look like:



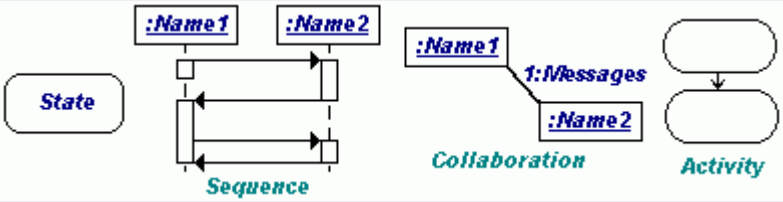
Deployment Diagram for Ethernet network

D.4.5 UML Diagram Set of Symbols

This is the set of important symbols of the UML.

Structural Elements

Behavioral Elements



Relationships

Grouping

Extension

Annotation & Actors