**What is UML?** (Thanks to Julie Adams)

**UML** stands for Unified Modelling Language.

**UML** is a modelling language used to express software designs.

**UML facilitates:**
- Specifying, visualizing, understanding, and documenting problems.
- Capturing, communicating, and leveraging knowledge in problem solving.
- Specifying, visualizing, constructing, and documenting solutions.

**UML captures knowledge** (semantics) about a problem and expresses knowledge (syntax) about the problem.

**UML defines:**
- A notation in graphical models, or the syntax of the language.
- A meta-model is a diagram that defines the notation.

**UML** is based upon the object-oriented paradigm.

**UML** can be applied to different types of systems, domains, methods, and/or processes.

**UML** enables software teams to capture, communicate, and leverage strategic, tactical, and operational knowledge to improve software quality, reduce costs, and reduce the product cycle time.

**UML** was originally developed by Rational Software Corporation in cooperation with Grady Booch, James Rumbaugh and Ivar Jacobson.

**Goals of UML**

**UML** is intended to be a ready-to-use visual modelling language that is simple and extensible.

**UML** is intended to be implementation independent (programming language or hardware).

**UML** is independent of the development process.

**UML** is intended to address recurring architectural complexity problems.

**UML** is scalable.

**UML** is intended to be general purpose and powerful while being simple to use.

**UML** is intended to communicate information about a problem.

- Natural language is too imprecise.
- Code is too detailed.

**What UML is not**

**UML** is not:
- A visual programming language that communicates an implementation or solution.
- A specification tool or repository.
- A modelling language specification that provides modelling elements, notation, and usage guidelines.
- A process that provides guidance on the order of activities, specifications of tasks, etc.
- It does not enable the process by promoting a use-case-driven, architecture-centric, iterative, and incremental process that is object oriented and component based.

---

**The Development Process**

**Stages of producing an application**

- Conceptualization
- Problem
- Problem-solving
- Solution
- Realization

**Process stages**

- Inception
- Elaboration
- Construction
- Transition

---

**The Inception Phase**

During the Inception phase, the team develops the business rationale and the project scope.

- How much the project will cost?
- How much money will be made from the project?
- How big is the project?

---

**The Elaboration Phase**

During the elaboration phase, the team collects more detailed requirements, does a high-level analysis and design for the baseline architecture, and creates a construction plan.

- Determine what it is you will actually build.
- Determine how you will actually build the project.
- Assess the project risks.

**The elaboration phase is completed when:**

- The team feels comfortable providing a development schedule.
- The team feels that all the significant risks have been identified and the major risks are understood.

**The elaboration phase usually takes about a fifth of the total project development time.**

---

**Project Risks**

**Types of project development risks:**

- The first risk is to ensure the team completely and accurately understands what should be developed. This is called requirements risks.
- The second risk involves the technology the team will use to develop the product.
- What hardware, programming languages, etc. This is called technology risks.

**Risks:**

- The third risks involves the level of experience and expertise your team requires to complete the project.
- Finally, the political risks must be assessed. Such risks may affect the teams ability to complete the project.
Requirements Risks

UML can be used to mitigate the requirements risks.

Use cases can be used to document the expected interaction between the user and the product.

Use cases usually correspond to the requirements.

Use cases are defined for most, if not all functions the product will perform.

It is very difficult to define all use cases.

UML can be used to develop a conceptual domain model.

The domain model defines the environmental constraints, and user constraints for the product.

An understanding of the domain model is useful for developing the use cases and the requirements.

Another UML tool used to mitigate the requirements risk are interaction diagrams.

Interaction diagrams represent how the various roles interact in the product.

This may be human-human, human-machine, machine-machine.

The UML tools we will discuss can provide the development team with the capability to record domain expertise that is critical for mitigating requirements risks.

Technological Risks

The technological risks usually involve incorporating different types of technology into the overall product.

This could be as simple as ensuring that the product runs on Windows 2000 or as complicated as predicting how the pieces will work with hardware that has not yet been developed.

One of the best ways to mitigate technological risks is to prototype the system components and try them out.

Another way to mitigate the technological risks is to design the product such that components can be replaced if the desired components do not work.

(Flexibility)

The UML use cases are one tool that can be reviewed to determine if there are any technological risks.

Skills Risks

Team members who do not have the proper training or not enough training are risks to the project.

Mitigating skills risks means ensuring that the team has access to training and mentors.

Most companies want to skimp on paying for training, the result is that the project takes longer to complete.

In the end, the cost to develop the project is the same or more than the training costs would have been.

Political Risks

The political risks for each project are different.

The risks may be that the project is “high stakes” meaning that there is a lot of pressure to create the best product, as cheaply as possible, in the shortest time possible.

The risks may be that the project is “low stakes” meaning that the team may work for a period of time, and then the project is cancelled.

Another example of a political risk is “stepping on someone else’s turf”. This means that the project is viewed as taking work away from another group or is an overlap of another groups work.

Political problems are not fun!!

The Construction Phase

During the construction phase, the product code is actually developed and tested.

The construction phase is a series of iterations during which code is developed, tested, and integrated.

The development schedule determines what features will be constructed during a particular construction phase iteration.

The first step is to determine which use cases will be implemented during each iteration (development of the release schedule).

The product users should divide the use cases according to business value.

The developers should divide the use cases according to development risks.

High risk items should be scheduled for early iterations.

The iteration duration should be determined.

It is possible that each iteration will not be the same length.

In general, an iteration should be long enough to allow the developers to complete a handful of use cases.

The order of use cases in certain iterations may require the order of use case implementation to be changed.

At the completion of each iteration, the released code should permit a user demonstration and systems test should be performed to verify requirements.

At the end of the construction phase, the UML documents that have been created can be used to help document the system but should not be the sole documentation.

The Transition Phase

The transition phase involves all the activities that need to be left at the end of the development cycle.

Alpha and Beta testing.

Performance tuning (optimisation).

User and support personnel training.

Alpha testing usually involves on-site testing of the complete system to ensure it works properly.

During Beta testing a subset of the customer base uses the completed system for a period of time before the product is made available to all customers.

Usually still debugging the system.

It is important to leave the system optimisation until the end because:

Optimisation usually reduces the clarity of the system.

This complicates the development process and can increase the development time.

There should be no new code development during the transition phase.

It is possible that the developers will be correcting bugs.
Use Cases

Use Cases describe the functionality or behavior of a system. Each use case describes a different capability that the system provides it's clients.

A use case contains:
- An actor or actors that represent the role the user plays while interacting with the system.
- A scenario that describes a sequence of steps detailing an interaction between the actor and the system.
  - It describes one feature that is provided by the system.

A use case is a set of scenarios that have a common user goal (function).

Actors

The actor in a use case represents the users of a system.
- The users can be humans, and/or other systems.
- It is unusual to have actors that are not human.
- The system actors may play multiple roles in a system.
  - For instance, a Sales manager maybe both a supervisor and a salesman.

A single actor may carry out many use cases.

Identifying Actors

Some questions to help identify the system actors are:
- Who is interested in a particular requirement?
- Where in the organization is the system used?
- Who will benefit from the system use?
- Who will supply the system with this information, use this information, and remove this information?
- Who will support and maintain the system?
- Does the system use an external resource?
  - Does one person play several different roles?
  - Do several people play the same role?
  - Does the system interact with a legacy system?

Use Cases

Not all use cases will be obvious.

Some methods for identifying use cases are:
- Reviewing focus group session results.
- Observing how users interact and use similar systems or their current system.
- Reviewing the marketing requirements document.
- Interacting with the customer.

Some questions to ask that can help identify use cases are:
- What are the tasks of each actor?
- Will any actor create, store, change, remove, or read information in the system?
- What use cases will create, store, change, remove, or read this information?
- Will any actor need to inform the system about sudden, external changes?
- Does any actor need to be informed about certain occurrences in the system?
- What use cases will support and maintain the system?
- Can all functional requirements be performed by the use cases?

Use Case Development Dangers

It is very easy to attempt to define all the use cases and their alternatives in a system.
- Eventually you will reach the point of diminishing returns.
  - Strive for the 80/20 rule.
- A good rule of thumb is:
  - A use case typically represents a major piece of functionality that is complete from beginning to end. A use case must deliver something of value to an actor.

Use Case Flow

A use case essentially documents the flow of events that are needed to accomplish the required behavior.

The flow of events should include:
- When and how the use case starts and ends.
- What interaction the use case has with the actors.
- What data is needed by the use case.
- The normal sequence of events for the use case.
- The description of any alternate or exceptional flow.

Use Case Outline

X. Use Case Name
X.1 Precondition
X.2 Main Flow
X.3 Subflows (if applicable)
X.4 Alternative flows

Simple Use Case

Jim needs to check his email to see if he has mail from his mother. He opens his mail program on the computer and browses through the messages. He finds a message from his mother.

Simple Use Case

1. Check mail
   1.1 The preconditions assume that the computer is up and running.
   1.2 Main Flow
      a. Open mail program.
      b. Browse through the new mail messages.
      c. Find an email from mother.

Simple Use Case

1.4 Alternative: Mail program is not active.
   At step 1, the mail program is not open.
   a. Allow the user to start the mail program.
   a.1 The user starts the mail program.
   a.2. The user enters the appropriate password.
Student Course Use Case
We are developing a system that will allow students to review the courses that they have registered for. When the students log onto the system they need to provide their student id number, a password, and the quarter they wish to review. Once this information is properly entered, the student is presented with a listing of courses for the quarter.

Professor’s Course Use Case
Another user of our system is the professor.
When entering the system the professor needs to provide a user name and password.
The professor needs to indicate to the system what courses he will teach during a particular quarter.
The professor also needs to be able to get a list of the students registered for his courses (course roster).

Use Case Relationships
A relationship indicates the communication between an actor and a use case.
The relationship is called an association relationship.
The actor may communicate to the use case (system).
The use case (system) may communicate to the actor.
Communication may flow in both directions.

Use Case - Include Association
It is possible that a system may have multiple use cases that contain the same behavior. In this case, the identical behavior is defined as it’s own use case and the multiple use cases refer to the similar behavior use case.
This is called an include relationship.
The association is labelled with <<include>>.
Defining a separate use case for the common behavior implies that one does not have to copy the description of that behavior to every use case that uses it.

Use Case - Generalize Association
If use case A is defined and a use case B does just a bit more then the use case A, then use case A can be generalized to capture the alternative scenario represented in use case B.
This is called a use case generalization association.
The generalization use case is associated with another use case.
Most likely the generalized use case is not associated with an actor.

Use Case - Extension Points
Another relationship involves extending a use case to add behavior to the base use case but with specific “extension points” are declared.
This is called an extend relationship.
The association is labelled with <<extend>>.
When the extend relationship is used, the additional behavior may only be added at the extension points.
An extension use case may also include “extension points” itself.

When to use include, generalization, and extend:
If the same steps are repeated in two or more separate use cases then use the include relationship to avoid the repetition.
If there is a variation on the normal behavior then use the generalization relationship.
If there is a variation on the normal behavior but there is a need to use a more controlled form, then use the extend relationship.
Remember to declare the extension points in the base use case.

Use Case Diagrams
The use case diagram shows:
The actors.
The use cases.
The relationships between the actors and use cases as well as use cases and use cases (extends, generalization, extends).

Classes
A class represents a definition containing specific data (variables) and behaviors (functions or methods in Java).
From the class definition, objects are created (instantiated).
Each class has a unique name.
Classes may have variables (attributes).
Classes have functions (operations or behaviors)

Class Diagrams
A class diagram describes the classes in the system and the types of static relationships that exist between instances of the classes.
The diagram shows the attributes and operations for a class.
The diagram will also show the object constraints.
Finally, the diagram will show the relationships between the classes.

Class Attributes
Each class object will likely have class attributes.
Attributes represent the data for a particular class.
An attribute has a name, data type, and possibly a defaultValue.
The Class Diagram definition contains attributes.
The Class object will list the names of attributes and their data types.
If necessary, the attribute may also have a default value.
The attributes shown in a class diagram do not indicate if the individual attribute is optional or mandatory.

Let’s look at a class definition for the Students in a SIS like system.
The attributes are:
user Name, password, major
**Class Operations**

UML uses the term class operation to refer to the behaviors or functions a class implements.

The class diagram definitions also contain the public functions for a class.

Functions that simply manipulate the class attributes (accessor/mutators) and do not interact with external objects (private) are not usually shown in the class diagrams.

A typical operation definition will look like:

visibility name (parameter-type-list): return-type

Visibility indicates if the function is public, private, or protected.

Public = ‘+’, private = ‘-’, and protected = ‘#’

Currently we will not worry about visibility.

Name is the function name.

Parameter-type-list represents the types of the data inputs for the function.

Return-type is the type of the returned value.

A query operation is a function that simply returns a value from the object.

We called these accessor or get() functions in Java.

A modifier operation is a function that changes a value or values in the object.

We called these mutator or set() functions in Java.

**Associations**

An association represents the conceptual relationship between the class instances.

Associations relate two or more classes where the relationships have common characteristics or features (attributes or operations)

An association does not represent data flow between two classes.

Data may flow in either direction.

There is simply a link between one class and the objects in another class.

Associations represent a permanent link between objects.

**Association Ends**

Each association has two ends, each attached to a single class.

Each end may be named, this name is called a role name.

The association itself can be named but we will not use this capability.

**Role Names**

The role name represents the purpose or capacity wherein one class associates with another.

A role name is typically a noun.

Role names are placed on the association near the class that it modifies.

The class where the attribute or operation is defined.

Most associations will have only one role name.

---

**Multiplicity**

An association end will also have multiplicity.

Multiplicity indicates how many objects of the class are associated with a single object of the other class.

In other words, the number of objects that are linked to one other object.

Each end of an association has multiplicity.

Multiplicity indicates the lower and upper bounds for the connected objects.

1 means exactly one object.

* means 0 .. = (infinity)

1 .. * means 1 to infinity.

0 .. 1 means zero or 1 object.

**Reflexive Associations**

It is possible that multiple objects belonging to the same class may have to communicate with one another.

In this situation, the association comes out of and goes into the same class.

A good example is a course class that requires information about course pre-requisites where each pre-requisite is stored as a course object.

**Finding Associations**

An association exists if a class requires an attribute with a data type of another class definition.

An association exists if there is communication between two classes.

Message passing.

**Generalizations**

A generalization link is used when the model contains super classes and sub-classes.

Generalization represents inheritance in object-oriented programming.

**Constraints**

Constraints can be described in UML.

While UML captures many constraints in the Class Diagram, there will be some that cannot be captured using associations, attributes, operations, or generalizations.

In this situation, additional constraints may be shown in the diagram.

The text of the constraint is stored inside of braces {}.

**Design by Contract**

Design by Contract relies on assertions to help debug programs during development.

**Assertions**

An assertion is a Boolean statement that should always be true.

The assertion will only be false when there is a bug in the program.

Assertions are usually only checked while debugging programs not during execution.

**Assertion Types**

Design by Contract uses three types of assertions.

Pre-conditions, post-conditions, and invariants.
**Assertion Types - pre-condition**
A pre-condition is a statement that represents the state the program should be in before an operation is executed.

A pre-condition specifies the function A is responsible for checking that the program state is correct before calling function B where function B requires a particular program state.

Pre-conditions apply only to operations.

**Assertion Types - post-condition**
A post-condition is a statement that checks the state of the program after an operation is completed.

Post-conditions apply to operations.

An exception occurs when the pre-conditions are meet, an operation is called, but the post-conditions can not be meet.

**Assertion Types - invariant**
An invariant is an assertion about a class and applies to all instances (objects) of the class.

An invariant specifies a constraint that must always hold true for the associated element.

An invariant may become false during function execution, but the invariant should be restored to true by the time any other object can interact with the receiver.

**Interaction Diagrams**
Interaction Diagrams allow the designer to show how groups of objects collaborate to obtain some behavior.

Interaction Diagrams will show the behavior for a single use case including the objects, and the message passing between the objects.

The two types of interaction diagrams we will talk about are sequence diagrams and collaboration diagrams.

**Sequence Diagrams**
A sequence diagram displays the object interactions arranged in a time sequence.

The diagram shows the objects and classes required for the scenario with the sequence of messages exchanged between the objects.

Sequence diagrams are composed of:
Class roles representing the roles that objects play in the use case.
Lifelines representing the existence of an object over a period of time.
Activations representing the time during which an object is performing an operation.
Messages represent the communication between objects.

Sequence diagrams are read and developed from left to right.

Usually the first item on the left is the actor for the scenario.
I will expect that the first item in your sequence diagram is the actor for the scenario.

This is then followed by the objects in the sequence that they will be accessed.
A lifeline shows the object's life during the interaction (scenario).
Lifelines are shown as a line displayed vertically from the bottom of each object.
Messages show the operations that are executed to complete a use case.
Messages are represented as a horizontal line with an arrow between the lifelines of objects.

The order of message calls is shown from top to bottom with the first call being on the top and the last call being on the bottom.
Each message should be labeled with the name of the operation.
A self-call occurs when an object sends a message to an operation defined in itself.

There may be operations that require certain information exist before the operation exists, this is referred to as a condition.

In this case the operation is only executed if the condition is meet.
Conditions are show by using [ ].

A return shows that an operation has completed and returns to the calling operation.

The return is shown on the diagram as a dashed line.

Usually returns are only shown for clarity, not for every message.

An asynchronous message allows the caller to continue with it's own processing while the called operation executes.

This is used when threads have been implemented.
Asynchronous messages are represented with half-arrowheads on the message link.
Accessed via the link specification.

UML also allows the signification of deleting an object.
A large X is displayed and indicates that the object will self-destruct.

Sometimes it is necessary to provide textual descriptions for the diagram.
UML standard is to place such messages on the far left of the diagram using the note feature.
The note feature can be used to place your name in your Rose files.

**Collaboration Diagrams**
Collaboration diagrams are an alternative to Sequence diagrams.
Collaboration diagrams describe the interactions among classes and associations.
The interactions are shown as the exchange of messages between classes through their associations.

Collaboration Diagrams are composed of:
Class roles representing the roles objects may play within the interaction.
Association roles representing the roles that links may play within the interaction.
Message flows representing messages sent between objects via links.
The links transport or implement the delivery of the messages.
The sequence of steps from the use case are indicated by numbering the messages.

This aspect makes it more difficult to visualize the message calling sequence than when a sequence diagram is used.

Rational Rose will automatically create the Collaboration Diagram from an existing Sequence Diagram.
Open the sequence diagram, select from the rose menu structure: browse -> create collaboration diagram
Sequence vs. Collaboration
Sequence diagrams should be used when trying to view the scenario in a time-ordered basis.
Collaboration diagrams should be used for get “the big picture” for a scenario.
Collaboration diagrams tend to be used during the design phase when attempting to specify the implementation of the relationships.

CRC Cards
A technique called Class-Responsibility-Collaboration (CRC) cards can be used to represent classes as responsibilities on index cards.
A responsibility is a high level description of the class’ purpose.
Responsibilities may be either attributes or a set of behaviors (in the future operations)
The collaboration represents the other classes required to fulfill the responsibility.

<table>
<thead>
<tr>
<th>Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check if items in stock</td>
</tr>
<tr>
<td>Determine price</td>
</tr>
<tr>
<td>Check for valid payment</td>
</tr>
<tr>
<td>Dispatch to delivery address</td>
</tr>
</tbody>
</table>

The key is to list all responsibilities on one card

Stereotypes
Stereotypes provide the capability to create a new kind of modeling element.
They can be used to classify or mark modeling elements.
A type of stereotype could be an interface
An interface is another name for an abstract class.
Stereotypes are displayed on the diagram using guillemets, e.g., << stereotype name >>
Stereotypes can be applied to classes or relationships in a class diagram.
Some typical stereotypes are:
<<actor>>
<<extends>>
<<include>>
<<generalization>>

Object Diagrams
An Object Diagram describes the static structure of a system at a particular time.
This diagram shows a particular situation while the class diagram shows all possible solutions.
Object Diagrams show instances of classes.
Object diagrams are essentially collaboration diagrams without messages.
Object Diagrams are composed of:
Objects represent particular entities or instances of classes.
Each object has a name and a class type.
Variables are usually shown with their current data.
Links represent the particular relationships between objects.

Class Scope
UML allows one to show which attributes and operations have class scope.
In C++ this would correspond to static members.
The only static members allowed is the main method as stated in the coding standard.
It is indicated by underlining the attribute or operation

Multiple & Dynamic Classification
Classification represents the relationship between an object and it’s data type.
Single classification means that an object belongs to a single data type that may inherit from super types.
Multiple classification means that an object may be described by several types that are not necessarily connected by inheritance.
Multiple classification permits multiple data types for an object without defining a specific data type.
Need to make sure which combinations are permitted.
A discriminator can be used to label a generalization line.

Multiple & Dynamic Classification
Dynamic classification permits objects to change type within the subtyping structure

Multiple & Dynamic Classification

Derived Associations & Attributes
A derived association or attribute is calculated from other information on the class diagram.
Derived values indicate a constraint between values.
An association or attribute that is derived has a ‘/’ in front of it in the class diagram.
Abstract Classes
An Abstract class is a class definition that as operation declarations but no function bodies or fields.
When creating an abstract class in UML
Include the condition '{abstract}' with the class name.
All abstract operations should be shown in italics.

Relationships:
A realization indicates that one class implements behavior specified by another class.
A realizing class must conform to the abstract class but does not have to use inheritance.
A dependency indicates that if the abstract class changes then the subclass may also have to change.
Dependencies should be kept to a minimum.

Reference Objects
A reference object uses a reference or pointer to refer to an object.
All objects that reference the same object.
You will only have one instance of a reference object.
Associations usually represent reference objects in UML.

Value Objects
A Value Object has multiple objects representing the same object.
Essentially there are multiple copies of the object.
Attributes usually represent value objects in UML.

Frozen
The frozen constraint indicates that the value of an attribute or association end may not change during the lifetime of the source object.
A constant variable.
Frozen is not read-only
The constraint is shown with the '{frozen}' and read-only is shown with '{read-only}'.

Classification vs. Generalization
It is common to hear inheritance referred to as an “is a” relationship.
Assuming that every sub-class has an “is a” relationship can lead to inappropriate sub-classing and confusing responsibilities.
Generalizations are transitive.
Classification is not transitive.
A classification followed by a generalization can be combined but not in the other direction.

Templates
Template classes allow us to define a single class that represents an entire range of related classes.
For instance, if we define a stack class we can create a template implementation that will create stacks of any data type.
When showing template classes in UML a template parameter is added to the class definition.
The template parameter is a place holder for the type parameter.
To create a template class in UML create a class instance and then select the class type to be parameterizedClass.
The operations defined for the class should also included the type parameter.

Visibility
Visibility refers to the ability of other classes to access a particular class’ elements.
Public elements can be accessed by any other class.
Private elements can only be used by the owning class.
C++ also includes protected elements but as the code standard indicates we will not use this visibility level.
Every attribute and operation will have an associated visibility.
All attributes are to be defined as private.
Operations may be private or public.
The main function is the only operation that is allowed to be protected.
The class diagrams should indicate if each element is private or public.
Public elements are preceded by a +.
Private elements are preceded by a -.
And protected elements are preceded by a #.

C++ also allows a class or a function to be defined as a friend of another class.
This means that the other class can directly access the private elements or functions in the original class.