Introduction to Unified Modeling Language (UML)

A computer program is typically a large, complex system composed of many different components. During the object-oriented system analysis and process, programmers must understand the requirements for the new computer program and determine the objects that should exist in the new program. An object is a software entity that represents something in the real world that must be incorporated into the computer program, such as a customer, an inventory item, or an invoice. During the analysis and design process, programmers work to clarify the software requirements and build a model of the new computer program composed of these objects. It is important to create a good model for the new program that can be used to communicate the complexities of the new program to other members of the software development team, users, and managers to ensure the development of a good business solution.

Unified Modeling Language (UML) is a very popular modeling tool that was designed to be non-technical. UML is a unification of modeling languages and methodologies originally created by several individual object-oriented software development experts, including Grady Booch (the Booch Method), James Rumbaugh (Object Modeling Technique), and Ivar Jacobson (Object-Oriented Software Engineering). UML provides a standard way of specifying, analyzing, designing, and documenting object-oriented software. It can be used to communicate the complexities of computer programs in a precise and succinct way to both technical and non-technical stakeholders. The more complex a program is, the more important it becomes to break the program into simpler, easier to understand components using a good modeling technique. UML was approved as the standard modeling language for object-oriented systems by the Object Management Group in November, 1997. Since it is a standard, UML has its own specialized shapes and conventions.

UML 2.x defines fourteen types of diagrams divided into two different categories:

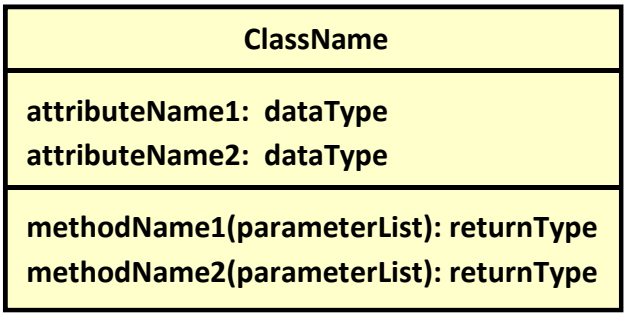
- Structure diagrams - used to document the architecture of the software
    - Class diagrams
    - Component diagrams
    - Composite structure diagrams
    - Deployment diagrams
    - Object diagrams
    - Package diagrams
    - Profile diagrams
- Behavior diagrams - used to describe the functionality of the software
    - Use Case diagrams
    - Activity diagrams
    - UML State Machine diagrams
    - Interaction diagrams - a subcategory used to model the flow of control and the flow of data
        - Communication diagrams
        - Interaction Overview diagrams
        - Sequence diagrams
        - Timing diagrams

In CSI 117, we will focus on only two types of UML diagrams: Class diagrams and Use Case diagrams. Class diagrams are emphasized throughout the course, since they are used extensively to analyze and design object-oriented computer programs.

Class Diagrams

A class diagram describes the static structure of a computer program, showing the different classes that make up the program and how the classes relate to one another. This type of diagram is more useful to communicate among the members of the software development team than it is to communicate with users and managers. The diagram shows which classes "know" about which classes (i.e., which classes call methods in other classes, but not specifically which methods are called), and which classes are "part of" another class.

A class defines the blueprint from which individual objects are created. A class in a program has a name, and it consists of attributes (data members) and methods (modules representing processes or behaviors). In UML, classes are represented by rectangles divided into 3 sections. The top section contains the name of the class. The middle section shows the class attributes. The bottom section shows the class methods with their parameter lists (and their return types if the method is a function).



Represents a class in a UML class diagram

Visibility Modifiers

Visibility modifiers (aka access specifiers) are shown in a UML class diagram immediately in front of attributes and methods to specify how other classes in the program can gain access to them. There are 3 visibility modifiers used in UML class diagrams:

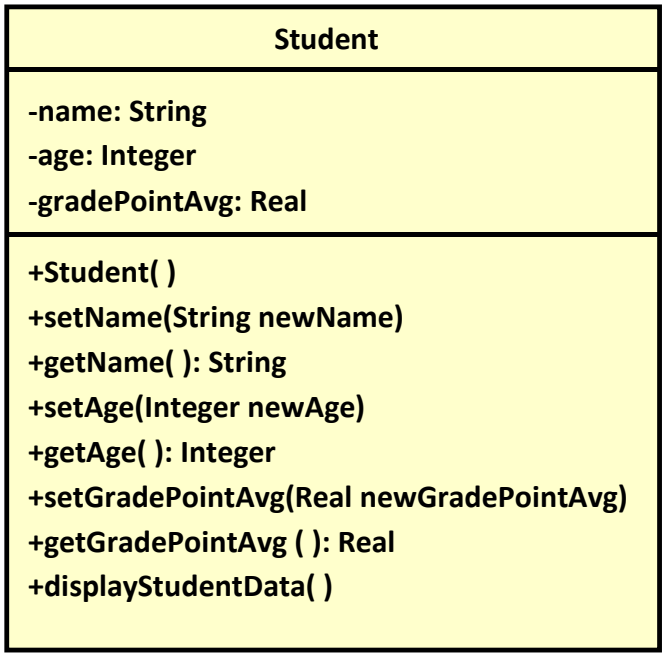
Symbol	Meaning
-	private: cannot be accessed by any method that is not part of the object's class; data members (attributes) are usually private
+	public: can be accessed by other programs and methods outside the class; methods are usually public
#	protected: used with inheritance; can be directly accessed by methods in a child class, but cannot be accessed by methods in an other class

Attributes

In a UML class diagram, attributes are shown using the visibility modifier, followed by the attribute name, followed by a colon (**:**), followed by the attribute's data type. For example:
-studentAge: Integer
-employeeName: String
#itemPrice: Real

Methods

In a UML class diagram, methods are shown using the visibility modifier, followed by the method name, followed by a left parenthesis (**(**), followed by a parameter list, followed by a right parenthesis (**)**). A parameter list consists of a declaration for each parameter (the data type of the parameter followed by its name), separated by commas (**,**). If the method is a function, the right parenthesis is followed by a colon (**:**) and the data type of the value returned by the method. For example:
+setStudentAge(Integer newStudentAge)
+getStudentAge( ): Integer
#displayInventoryData( )
-calculateTotalPrice(Real salesTaxRate): Real



An example showing the name, attributes, and methods of a single class

Class Relationships

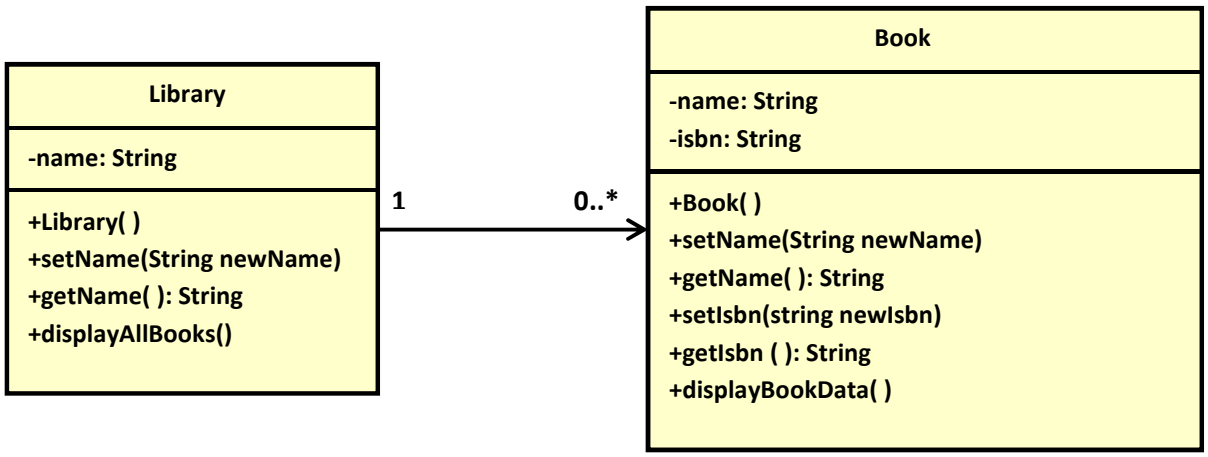
Classes can relate to one another in different ways. Relationships in class diagrams include Associations, Aggregations, Compositions, and Generalizations. The specific notation used to connect classes on a class diagram indicate the type of relationship between the classes. Numeric values are also shown on the relationship to indicate multiplicity.

Multiplicity (Cardinality)

Place multiplicity notations near the ends of a relationship. These notations indicate the number of instances (objects) of one class that is linked to one instance (object) of the other class. It provides a way to express constraints on the number of objects participating in a relationship. For example, the program may have the constraint that one Library object can have one or more Books, but each Book object can belong to only one Library. Multiplicity may be specified as shown in the table below.

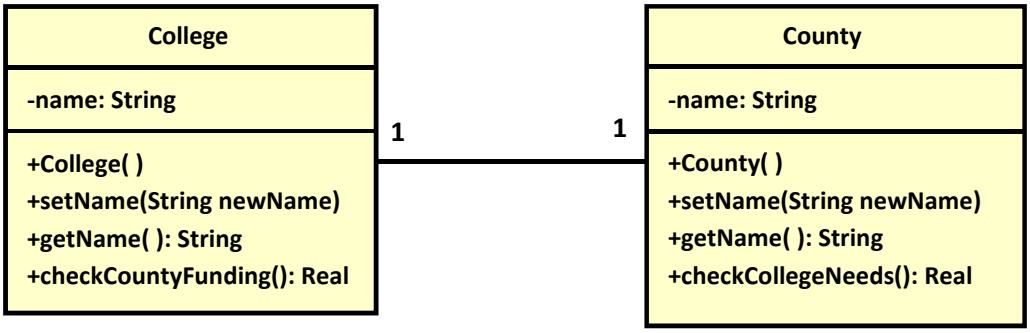
Multiplicity Notation	Meaning
	If no symbols or numbers are shown, 1 is assumed
1	Exactly one
*****	Zero or more
0..*	Zero or more
1..*	One or more
0..1	Zero or one
3..7	Specified range (in this case, 3, 4, 5, 6, 7)

The multiplicity notations shown in the diagram below specify that the program contains one Library object for every Book object, and for each Library object in the program there can be zero to many books.

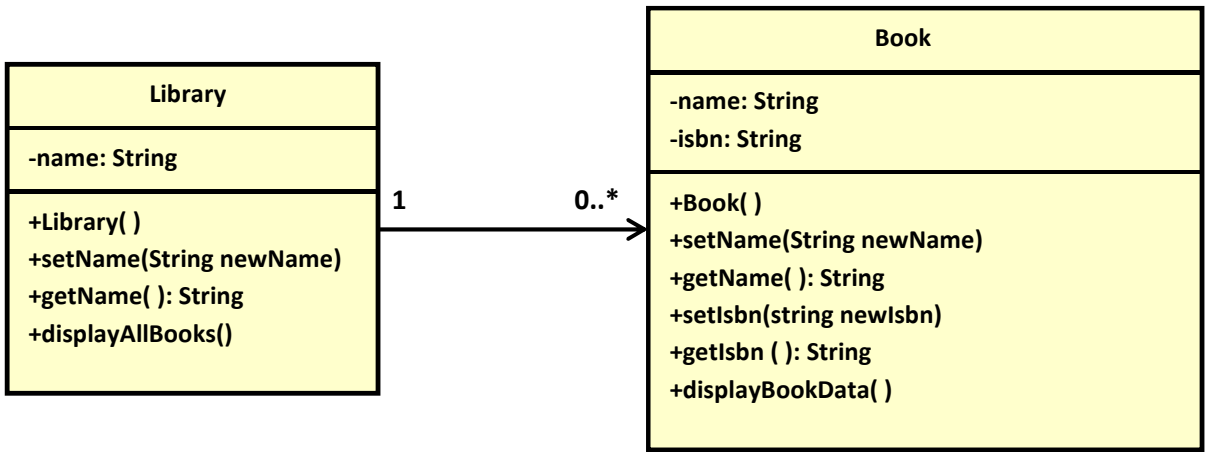


Association Relationship

An association represents a relationship (connection or link) between classes that indicates how instances (objects) of one class can communicate with instances (objects) of the other class. An association can be bidirectional, indicated by a solid line with no arrows, meaning that objects of each class know about objects of the other class and methods in each class make calls to methods in the other class. Each end of the association also has a multiplicity value, which specifies how many objects on this side of the association are associated with one object on the other side. The diagram below shows a bidirectional association. The diagram indicates that methods in the College class call methods in the County class, and methods in the County class call methods in the College class. The multiplicity values indicate that for each College object in the program, there is only one County object, and for each County object in the program, there is only one College object.



Alternatively, an association can be unidirectional, indicated by a solid line with an arrow on one end, meaning that objects of only one class know about objects of the other class and methods in that one class make calls to methods in the other class, but the reverse is not true. Again, each end of the association has a multiplicity value, which specifies how many objects on this side of the association are associated with one object on the other side. The diagram below shows a unidirectional association. The diagram indicates that methods in the Library class call methods in the Book class, but methods in the Book class do not call methods in the Library class. The multiplicity values indicate that for each Book object in the program, there is only one Library object, and for each Library object in the program, there are 0 to many Book objects.

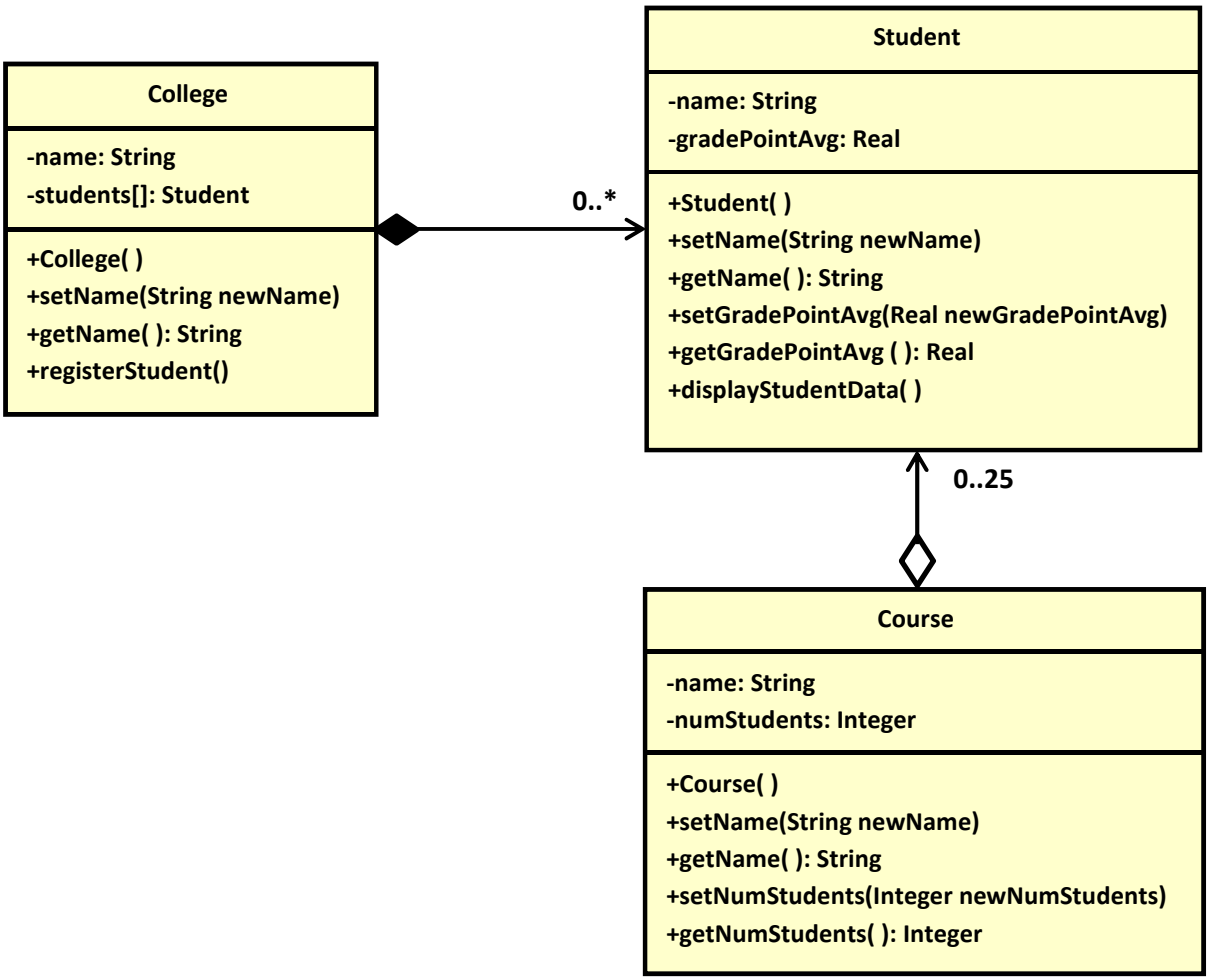


Composition and Aggregation Relationships

Aggregations and Compositions are special types of associations in which the two participating classes do not have equal status, but they have a "whole/part" relationship. In both types of relationships, the class playing the part of the whole always has a multiplicity of 1, and that value is not shown on the diagram.

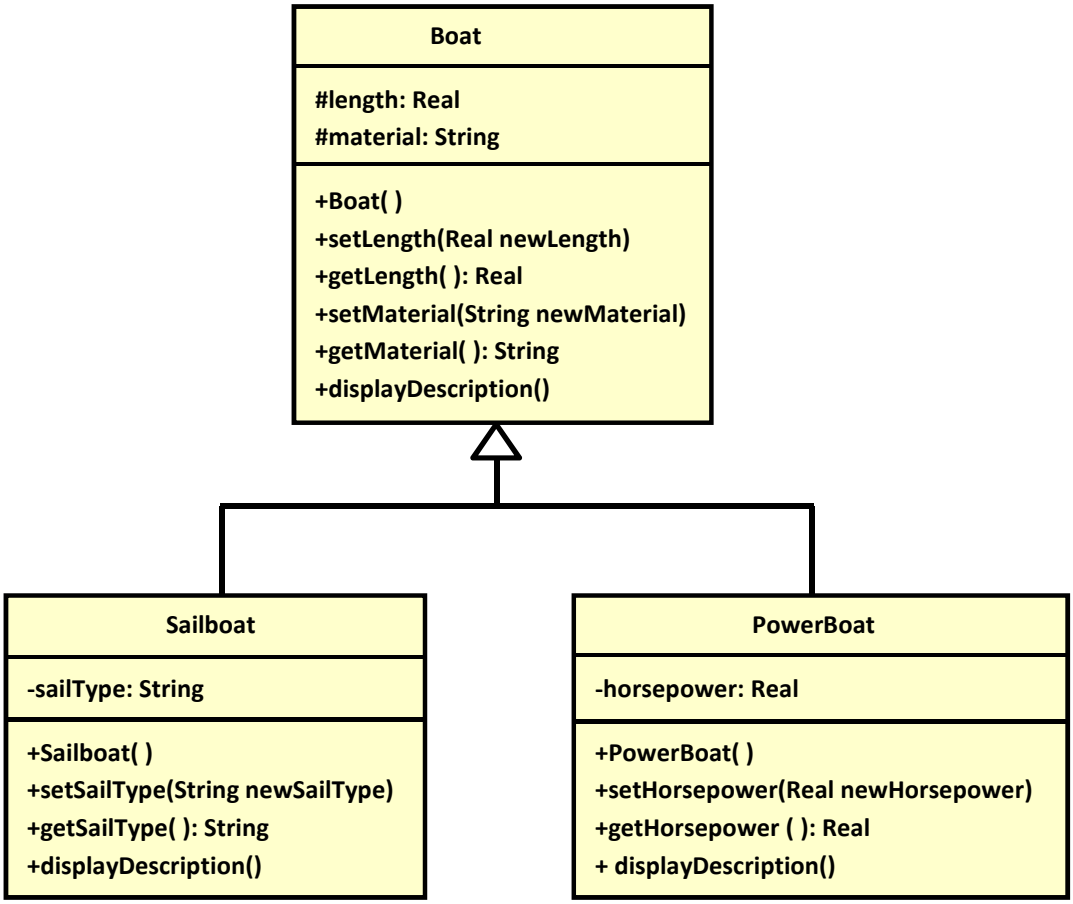
A composition relationship, designated with a filled diamond on the side of the whole class, denotes a strong ownership relationship between the whole and the part. The objects of the part class cannot exist on their own, meaning they exist only inside the whole, so if the whole object is destroyed, all the part objects are destroyed simultaneously. In the diagram below, the College class is composed of Student objects. If the College no longer exists in the program, the Students no longer exist, either.

Aggregation, designated with a hollow diamond on the side of the whole class, is also a whole-part relationship, but it does not have the strong ownership of the composition relationship. This means the part objects can continue to exist after the whole object is destroyed. In the diagram below, the Course class is an aggregation of Student objects, but when the Course class ends, the Student still exists and can take other Courses.



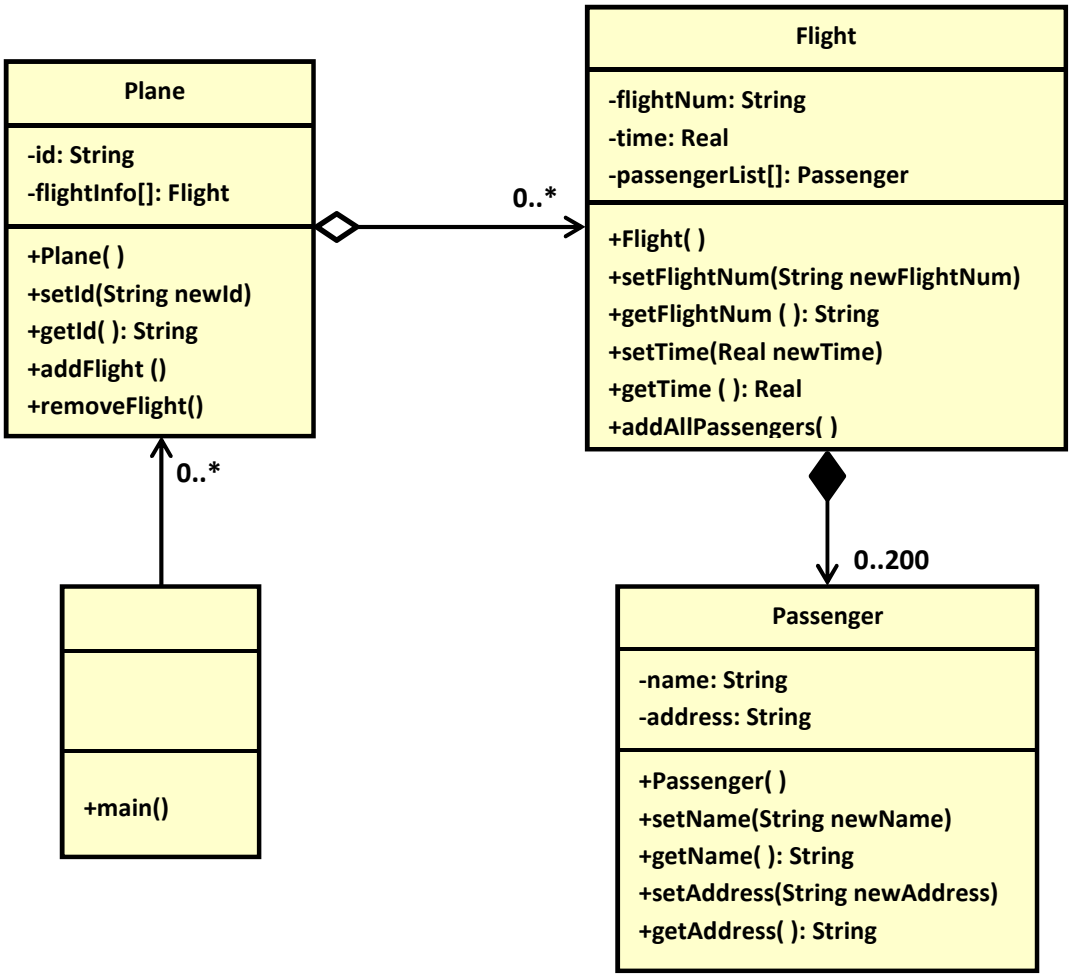
Generalization Relationships

Generalization is another name for inheritance, which is one of the fundamental features of object oriented programming. Generalization is an "is a" relationship, referring to a relationship between two classes where one class is a specialized version of another. In such a relationship, the child class (derived class) inherits the attributes and methods of the parent class (base class). The child class becomes a specialized version of the parent class by changing the attributes and methods it inherits and/or defining additional attributes and methods of its own. For example, the diagram below shows that a Sailboat and a PowerBoat are both specialized types of Boat. In a class diagram, a generalization is represented using a hollow arrow on the side of the parent class. Note that in order for the child class to have the ability to directly access attributes in the parent class, the attributes in the parent class must have protected access, rather than private.



Putting It All Together

An example of a complete class diagram is shown below. Note that a program must always contain a main() method; therefore, a complete class diagram must always show the main() method. The main() method may be part of a class (called the application class), as in a Java program, or it may exist outside of all classes, as in a C++ program, where the application itself is not a class.



Use Case Diagrams

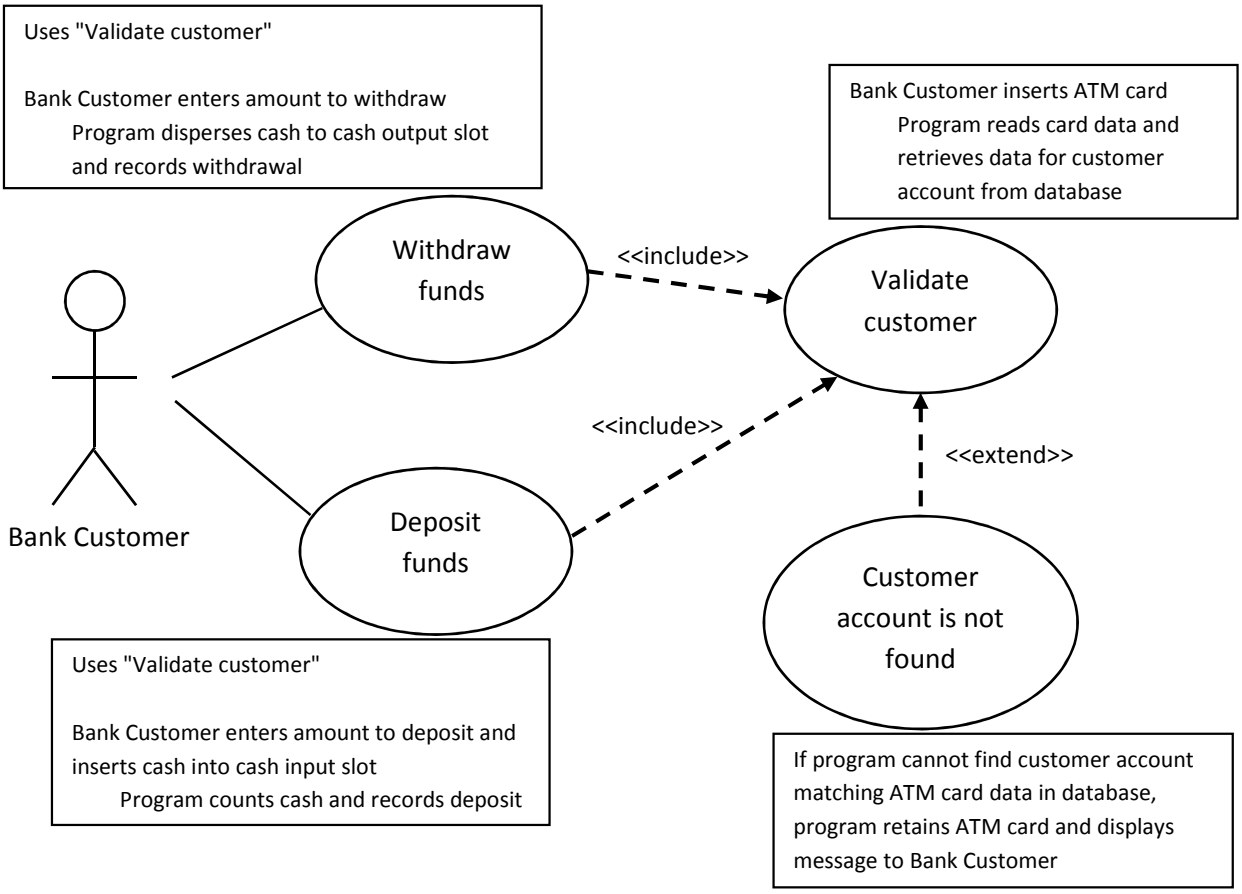
Use case diagrams are used during the analysis process to capture and clarify the requirements of the program. They provide a way for the programmer to communicate with the future users of the program, such as company employees, customers, suppliers, and managers about what the program is supposed to do. A use case diagram models ***what*** the program is supposed to do, but ***not how*** the program will do it. Use case diagrams are used to describe typical interactions between users and the program. Each use case diagram describes a group of actions taken by an actor to reach a specific goal or perform a particular task. Many different use case diagrams are drawn for one program. Therefore, one use case diagram does not depict all the functions of the program. One use case diagram depicts the program functions that are visible to one actor (a user playing one role) when the actor is working with the program to perform a particular task.

A use case diagram is a collection of the following:

- Actor
    - The role played by a person when interacting with the program, or an external entity, such as another computer system or a database, that interacts with the program
    - If the same person assumes different roles and interacts with the system in different ways to perform different tasks, a different use case diagram is drawn for each role/task
    - Represented by a stick figure with a short, descriptive name
- Use case
    - An action performed by an actor to achieve a specific goal or perform a particular task
    - Represented by an ellipse containing a verb in present tense followed by a noun phrase that describes the action
- Interactions and/or communications between the actors and the use cases, represented by lines with no arrows
- Relationships (either <<extend>> or <<include>>) between use cases, represented by dashed lines with an arrow showing the direction of the relationship
- Text linked to each use case that describes the actions the actor performs and the program's responses to those actions

Typically, the actor that initiates the task is shown on the left side of the diagram, the use cases are shown in the center, and any other actors involved are shown on the right side of the diagram. An <<extend>> relationship describes something that is outside the normal course of events, such as an id number that is not found. An <<include>> relationship describes an action that is common to multiple use cases, so it is pulled out into is own use case to avoid repeating text for the same actions multiple times in the diagram.

Here is an example of a use case diagram for a bank customer:



References

An Object-Oriented Approach to Programming Logic and Design, 3rd edition, Joyce Farrell, Course Technology, 2011

"Practical UML : A Hands-On Introduction for Developers", Randy Miller, http://edn.embarcadero.com/article/31863

"Software Design Tutorials – Unified Modeling Language", www.smartdraw.com

"Umbrello UML Modeller Handbook", The Umbrello UML Modeller Authors, c. 2003

UML 2 and the Unified Process, 2nd edition, Jim Arlow and Ila Neustadt, Addison-Wesley, 2005

UML Distilled, 3rd edition, Martin Fowler, Addison-Wesley, 2004

UML Explained, Kendall Scott, Addison-Wesley, 2001

"UML Tutorial", Sparx Systems, http://www.sparxsystems.com/resources/uml2_tutorial/uml2_usecasediagram.html