Thu Jan 6 13:14:27 PST 2005

Using The Unified Modeling Language in CS320

Introduction

This is a set of notes on how we can use the Unified Modeling Language(the UML) to help define programming languages. BNF does a good job for describing syntax. The UML clarifies the concepts and relationships involved in a programming language. Many call this the *static semantics*. The UML can describe the behavior of each part of a program when it is executed or evaluated. So the UML provides a way to define what a language means -- *the semantics*. BNF and UML complement each other. They can be used in most software projects.

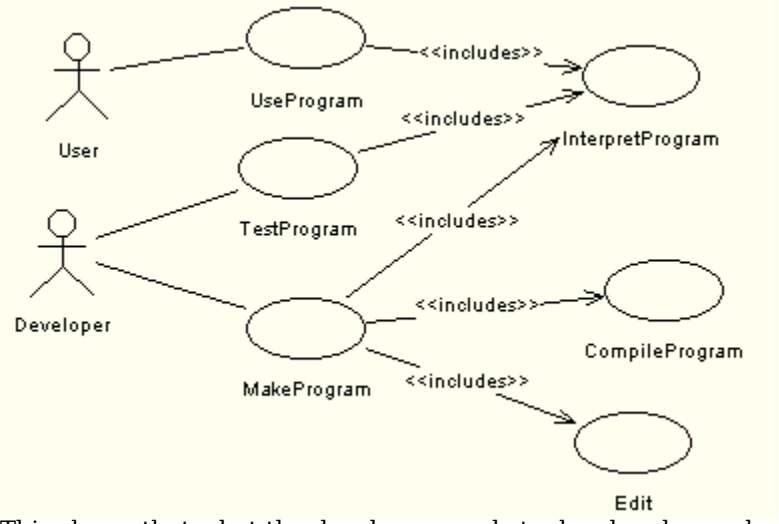
These notes are in three parts. First come the diagrams that tend to be the same for all languages and so don't have to done in your project. Second the diagrams that are most important in the CS320 project. Third are some common patterns needed to describe programming languages in the UML.

These notes do not cover the whole of the UML. They omit Activity diagrams (flow charts in the UML) and Collaboration diagrams, and don't say much about Sequence Diagrams and State Charts. For a more complete overview of the UML see my notes on the UML for CSci202 and my summary .

Part 1: General Properties of Programming Languages

UseCases

The best starting point for a *real* computer project is knowing what the users want. In the UML the user's needs are represented as a collection of *UseCases*. Each UseCases describes some process that gives the user some tangible response. It is a piece of text describing how an actor gets the response -- typically as a successful scenario plus a number of more complex ones. The details of how the actor interacts in the usecase is given in a *scenario* -- a simple text description often developed using a word processor. The users of the software appear as classes of *actors* shown by little stick-figures of people. Each use case is drawn as an oval bubble with a name. In the case of programming language, the diagram is always like this:



This shows that what the developer needs to do when he or she develops software:

- Developing the software means making and testing it.
- Making software can involve editing.
- Making software means using compilers and/or interpreters.

The developer's needs are *not* what the user of a program wants. A user may need to test the software but they do not want to use editors and compilers! They want to use the software.

Usecase diagrams help us think about programs from the user's perspective. It allows to spot common shared UseCases. It lets us plan the order in which we will work on the software. A usecase diagram does not tie us to a particular program design and lets us implement and schedule the various activities in many ways.

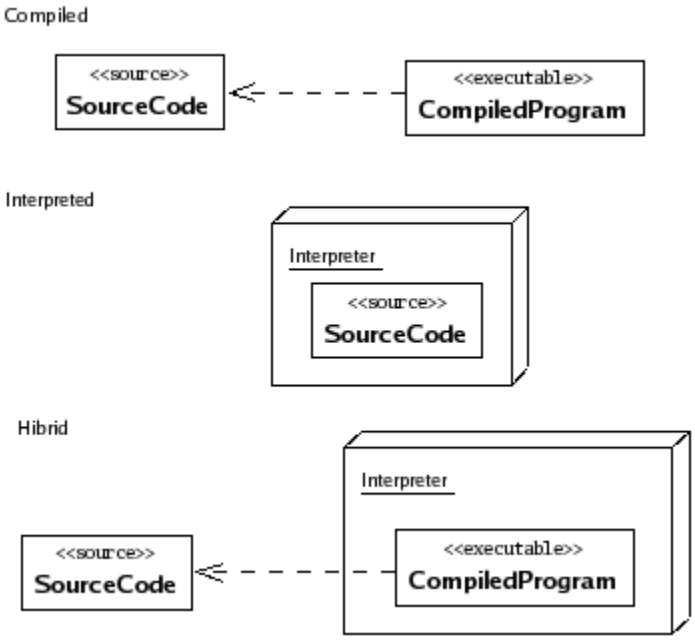
A use case diagram by itself is not enough to design software. The written scenarios fro the use case bubbles above are left as an exercise for the student.

Artifact Diagrams

An *artifact* is the finished products of a software project: a compiler, an editor, a game, an application, a library, a ton of source code, and so on. They are shown a boxes with a stereotype (in *guillemots* <<...>>) that indicates the type of artifact. See table below:

Stereotype	Meaning(UML2)
<<file>>	A physical file in the context of the system developed.
<<script>>	A script file that can be interpreted by a computer system.
<<executable>>	A program file that can be executed on a computer system.
<<library>>	A static or dynamic library file.
<<source>>	A source file that can be compiled into an executable file.
<<document>>	A generic file that is not a source file or executable.

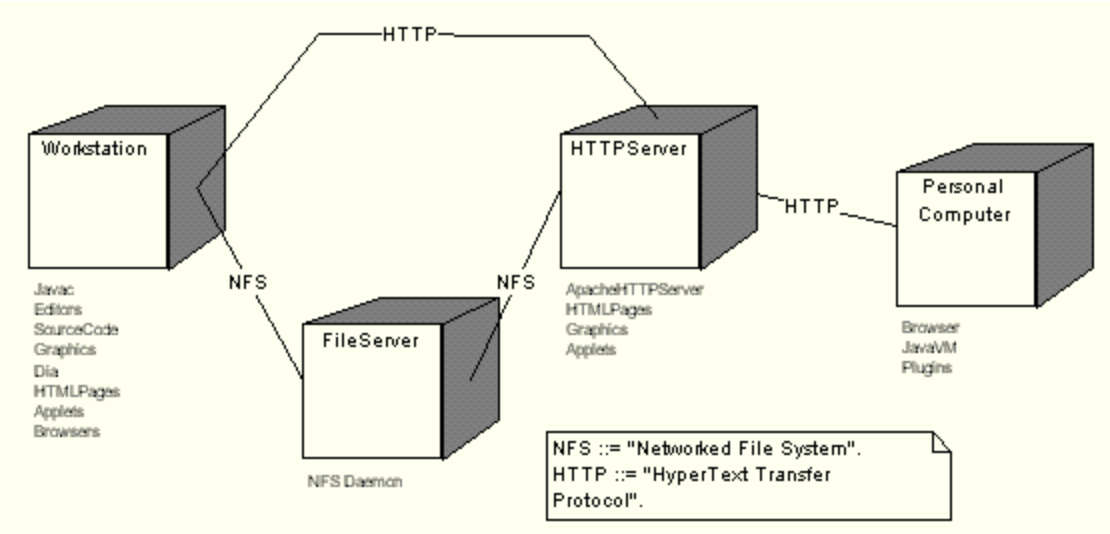
These artifacts *manifest* components which in turn *encapsulate* collections of collaborating classes. In large projects one has to plan how the the software is broken into *components* and *artifacts*. One needs to worry about where code is executed. You need to document this information and also show how the components (and artifacts) depend on each other. The UML also has a special way to show that an artifact is executed by a piece of software. This needs the cubical symbol for an execution environment from UML Deployment diagrams. In programming languages there are essentially three ways a language is implemented: compiled, interpreted, and hybrid (See chapter 1 of Sebesta's book). Here are the three diagrams:



Deployment Diagrams

Deployment diagrams are the UML way to document hardware. They are vital for Internet projects. They show how pieces of hardware are connected and the *execution environments* that support and execute programs. They define the network of machines that implement the software. They can also document where the software artifacts are placed in this network.

There are lots of specialized symbols you can use for CPUs, Devices, Pages, Scripts, etc. The example following uses the UML standard Processor node. They model the Deployment of components in your CS320 lab work:



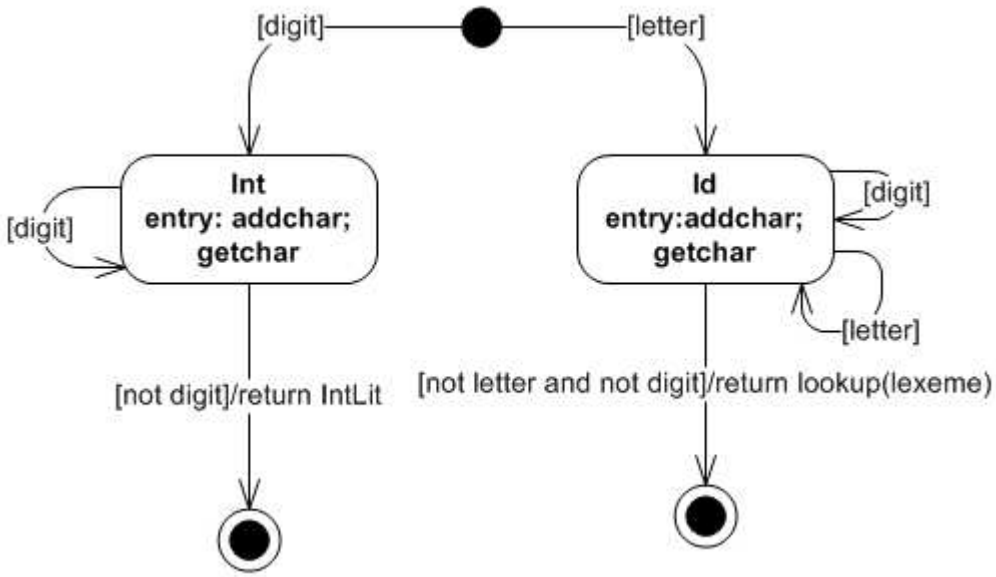
The above UseCase, Component, and Deployment diagrams will describe nearly all programming languages in CSci320. You shouldn't need to draw them in your CSci320 project.

The UML has other kinds of diagrams: Activity, Sequence, Collaboration, and State diagrams that we may need in class. They may help you in other courses in our degree.
However, they will not be needed in projects in CS320. For more on these topics see another of my handouts: A Quick Guide to the UML .

Part 2: Expressing the Particular Properties of Programming Languages

Modeling Syntax

The UML is not designed to describe the syntax of languages. But there is one diagram in the UML -- the state Machine diagram -- that is ideal for defining lexical structures. The CSCI320 book (Sebesta) has a simple lexer in Figure 4.1 as a Finite State Machine. Here is the same machine expressed in the UML



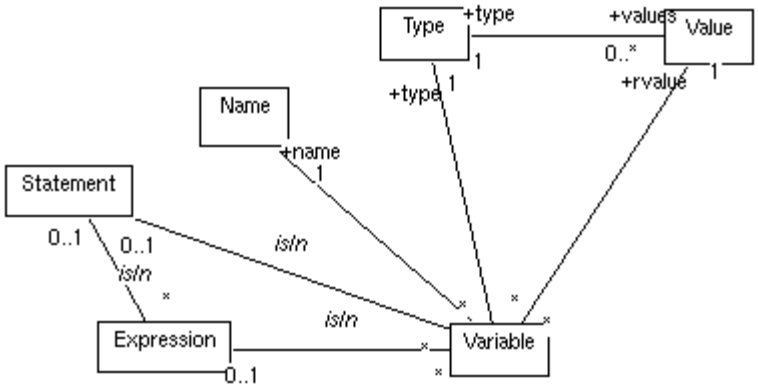
The more higher-level syntax is best expressed using Extended Bachus-Naur Form.

Modeling Concepts

Languages differ in their ideas. The UML *Class Diagram* is good at showing how ideas (concepts) fit together. They have concepts in boxes with lines connecting them between them.

If we wanted to model an HTML document we would need ideas like this: HTML Document, Head,Title, Body, Anchor, URL, etc. To model most programming languages we would need concepts(boxes) like: Statement, Expression, Variable, Type, etc. .

Here is a UML model for the concepts like Type, Variable, Value, and so on plus the associations between them:



The diagram above shows a collection of named boxes - indicating sets, classes or types of object. The boxes have lines connecting them called *links*. The UML associations are called *bindings* in a programming language.

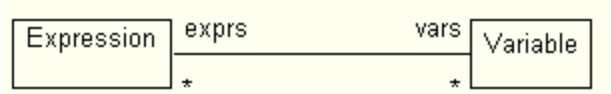
Associations are often marked with *roles* and *multiplicities*.
The names placed at one end of a link are called *roles*. If a *role* is omitted then it is assumed to be the name of the box with the first letter in lower case. So some of the roles in the above diagram can be left out -- which?

The numbers and asterisks put next to links are called multiplicities. They show how many objects participate in a relationship. For example, each Variable has precisely one Name, Value, and Type. However many Variables can have the same Value and any number of Variables in a program can have particular Type. An asterisk(*) indicates an unknown number. The form *n* .. *m* means between *n* and *m* inclusive. The 0..1 format indicates options. An 1..2 means 1 or 2. And 28..31 would describe the multiplicity of days in a month. Notice that Statements can have many Expressions and many Variables in them. And an Expression can have many Variables in it. If you omit a multiplicity then it is assumed to be '*'. Note this well! No multiplicity does not mean 1. It means `any number`. But so does "*" and "0..*".

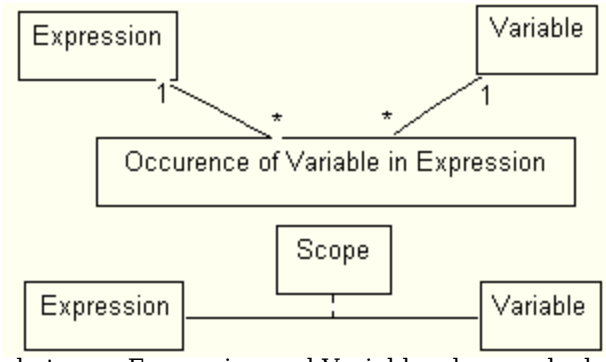
It is wise to do diagrams like this in pencil or on an erasable board at first. A simple diagramming tool also will let you sketch ideas and then edit them. A drawing like this is changed many times in a project as we learn more about how the objects fit together. The associations and classes define an architecture that will develop and then become a stable core of all the software in a given domain.

It is nearly always worth investigating many-to-many relations like the one between Expression and Variable above.

- A Variable can occur in many Expressions.
- An Expression can have many Variables.

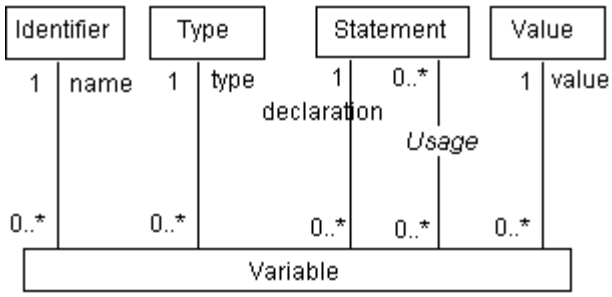


A very common technique in programming languages is to separate the occurrence of a variable from the variable itself. The variable is responsible for tracking it's name, type, and value, plus all of its occurrences. An alternative is to invent a class that is responsible for the rules associating Variables and Expressions -- the Scopes:



We will return to the relationship between Expression and Variable when we look at physical modeling below.

Often a complex relationship is embodied in a class of objects. Each is responsible for connecting the objects taking part in the relationship. Each object is as a "node" linking several other objects. For example in a modern programming language a Variable will have a name, a declaration, a number of uses, a data type and a value:



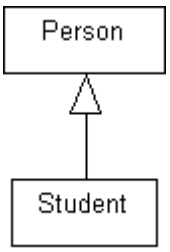
Notice, in the above figure, that unlike a mathematical digraph, two boxes can have two or more parallel links.

Special Relationships Between Classes

The UML provides notations for some special relationships between classes/concepts:
generalization, abstraction, composition, aggregation, dependency, ...
Only use these if you are certain that they apply. If you have any doubts use a general link and a comment.

Generalization

Generalization is a relationship that is naturally expressed with a sentence like this: "Every ____ is also a _____". It classifies objects by how they behave. Here is the simple diagram that states that `Every Student is also a Person`:



Generalization is used only when you know that all students are a kind of person -- they have all the same properties and operations. It says that everything that a Person can do can also be done by a Student -- but in the Students own special way.

So, any operations listed for a Person are automatically ok for a Student as well. If an operation (member function) is listed for a Person and `also` for a Student then this means that People and Students can both do the operation, and that, they do it `differently`. If an operation appears in Student alone then it does not effect what People can do.

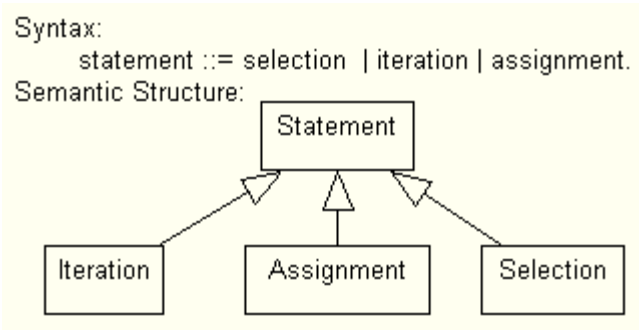
Similarly with attributes: Students have all the same attributes as People and Students can have extra attributes that People don't have. If an attribute (data member or field) is in both Student and People then Student has two copies: one as a Person and one as a Student. Code in the Student class then refers to the Student's attributes.

We say Student is the *subtype* and the Person is the *super*-type. A subtype or generalization is a class that shares all the properties of its parent (super-type, base class). Subtypes may extend or override their *inherited* properties.

The diagram is correct if and only if (1) a Student shares the same data (attributes) as a Person, (2) any operation that works on a Person also works on a Student, and (3) every Student automatically and permanently becomes a Person when the Student is created. Further this arrow should only be used when every object in the new class is automatically an object of the class it generalizes. Logically this means that anything that can be done to or by a Person can also be done to or by a Student... however a Student may sometimes react with different behavior than the more general Person does.

The diagram does not allow a Person to become a Student. *Generalization is static*. Some people are Students at the start of their life and others are not. In a program this would mean that the class `Student` would extend or inherit all the properties from `Person`. Generalization is most common in formal domains like programming languages, mathematics, geometry, application programmer interfaces (APIs), and legislation. In a domain where an object can start out as a Person and then become a Student, or where an object can start out as a Student and later cease to be a Student we would want some kind of dynamic connection between Person and Student. The UML does provide a standard way to document this but in CS320 please treat dynamic relationships as associations.

In programming languages for example there are many types of statements. They are all statements, but each behaves slightly differently. So we can say that we have a general Statement and Specialized ones. Here is a diagram of a Language with three special kinds of Statement: Iteration, Selection, and Assignment. The BNF + UML might be:

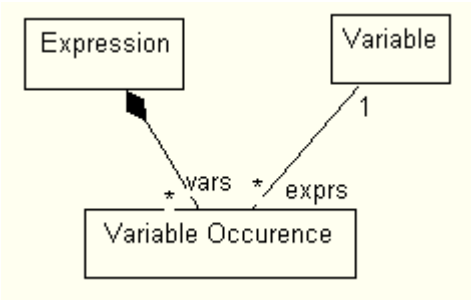


Abstraction

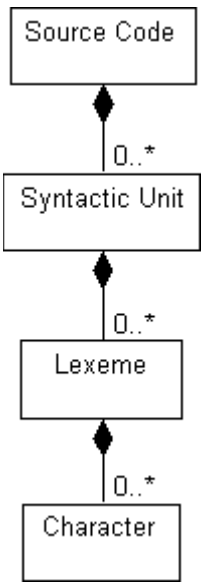
There is a similar relationship to generalization when one class implements the properties of another class. It is drawn with a similar arrowhead but with a dashed line. We will cover the UML symbols for abstractions when we cover Abstract Data Types later in this course.

Composition

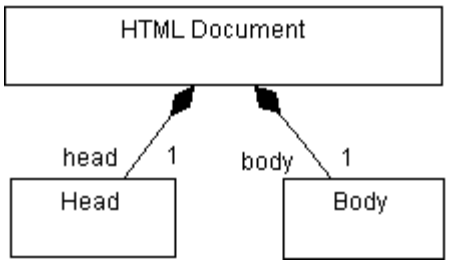
In C++ we can declare a *struct* to be a collection of components. A piece of text is made up of words. A car has four wheels. In Mathematics we talk about something being a "triple (A,B,C) where A is a .... and B is a .....". These are valid examples of *composition*. It is indicated by a black diamond at one end of the link. (◆——) We use the dark diamond to indicate that the objects in one class possesses or is made up of the components. For example an Expression contains many occurrences of a Variable:



In programming languages the composition symbol occurs mainly when we talk about pieces of code being inside other pieces. For example one of the tasks of a compiler is to take a stream of characters and organize them into lexemes and syntactic units:

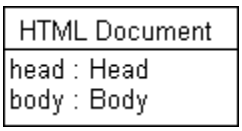


An HTML Document, for example, has two parts, its Head and its Body:



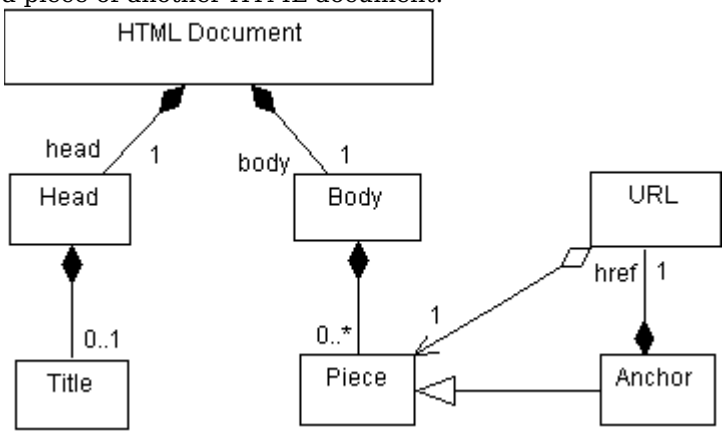
Attributes

Instead of using composition (◆—) you can list the parts of an object inside the box:

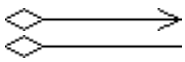


Aggregation

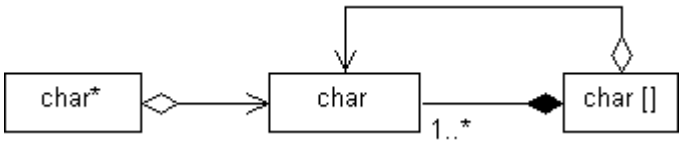
We can also show that a class has parts that have an independent existence. They are not physically a part of the object, but are "owned" by it in some other way. Examples are pointers, references, and addresses in programming languages. This kind of relationship is shown by an open or empty diamond. For example, in the HTML a link between two documents is made by an anchor tag that contains a Universal Resource Locater (URL). The URL refers or points to a piece of another HTML document:



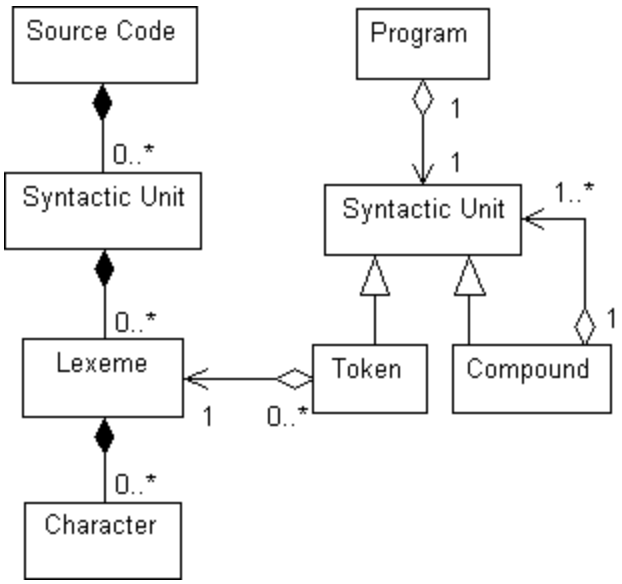
One end of an Aggregation has a diamond. The other end of the *Aggregation* link indicates whether the association can be navigated in both directions or only one:


Can follow link in one direction
Can follow link in either direction

Aggregation is often used to show a pointer in a data structure. For example in C the data type of 'char*' consists of pointers to a character, but a character array(char[]) is made of a characters. Each array name *also* represents the address of first of these characters. So, not only does an array *contain* many characters it also *refers* to one of them.

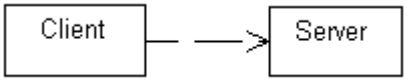


Aggregation and composition can be used to show complex internal data structures. For example, a compiler will construct an internal data structure corresponding to the structure of the input source code.



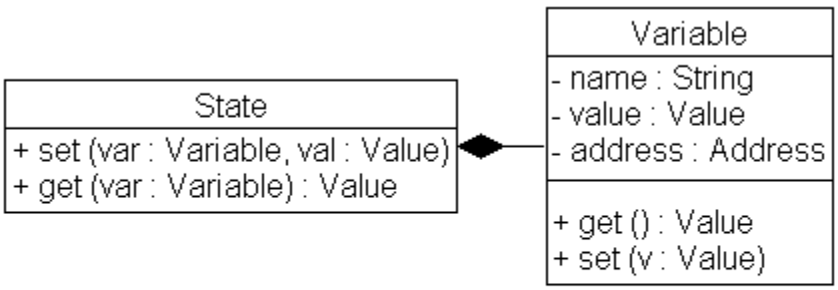
Dependency

Sometimes we wish to show that one object or class depends on another object or class. Perhaps one object is a client for a service (operation) provided by another object. Perhaps a change in one object forces a change in a dependent object. Perhaps the definition or specification of one class refers to the another class. In the UML this is shown by a dashed arrow connecting the dependent object to the object it relies on. For example a client relies or depends on its server.



Attributes and Operations

In the later stages of object oriented analysis and design we need more detail on our objects. We can add the *attributes* (data)and *operations* (behavior) of objects in a single box like this:



The State of a running program in a simple language is a collection of Variables each with a name, address, and value.

As in C++, operations and attributes can have the same name in different classes. The double colon (::) symbol is used to distinguish the different versions:
Variable::set(value:Value)

Programming Language Semantics

In CS320 we use the the UML to express the semantics of programming languages by *adding operations* to the boxes that describe the structure and ideas of the language. Typically we use an *execute* operation on statements and an *evaluate* operation on expressions. Typically the evaluation of expression will depend on the values of the variables in the expression. This means we would need a new class that associates variables and values together. Since this models the state of a virtual machine we would call it State. States would have two operations

State::set(Variable, Value) : State

State::get(Variable) : Value

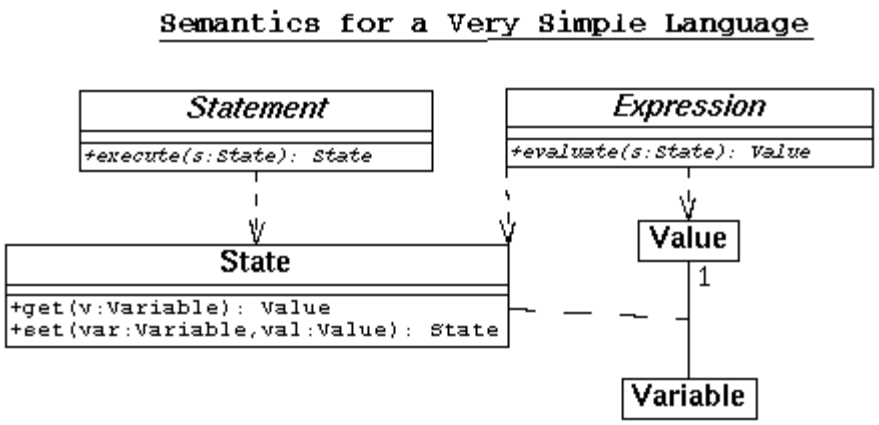
Given State and Value the evaluate operation would be like this:

Expression::evaluate(State) : Value

Executing a statement is mode complex because it will change the state of the virtual machine. So, the signature of the execute operation should be:

Statement::execute(State) : State

Putting all these ideas together we get a generic model for the semantics of a very simple programing language with Statements, Expressions, Variables, and Values:



Note. Your mileage may vary. The semantics of real languages is a lot more complex than this.

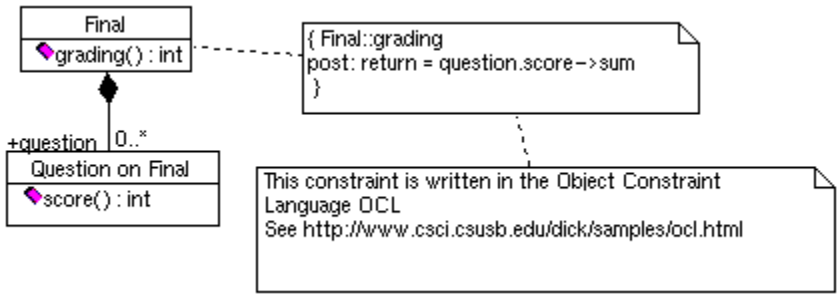
Constraints

We often need to show that a property always holds in a valid program. For example that each variable is declared before it is used. Constraints can be used to do this. The UML has several ways of doing this including Comment boxes and a special syntax on links and inside boxes:

```
{ constraint }
```

You usually write these in a mixture of English and mathematics. There is also a special Object Constraint Language (OCL).

For example, the operation of grading a Final implies adding up the scores on the Questions on the Final. Here is a picture of the class:

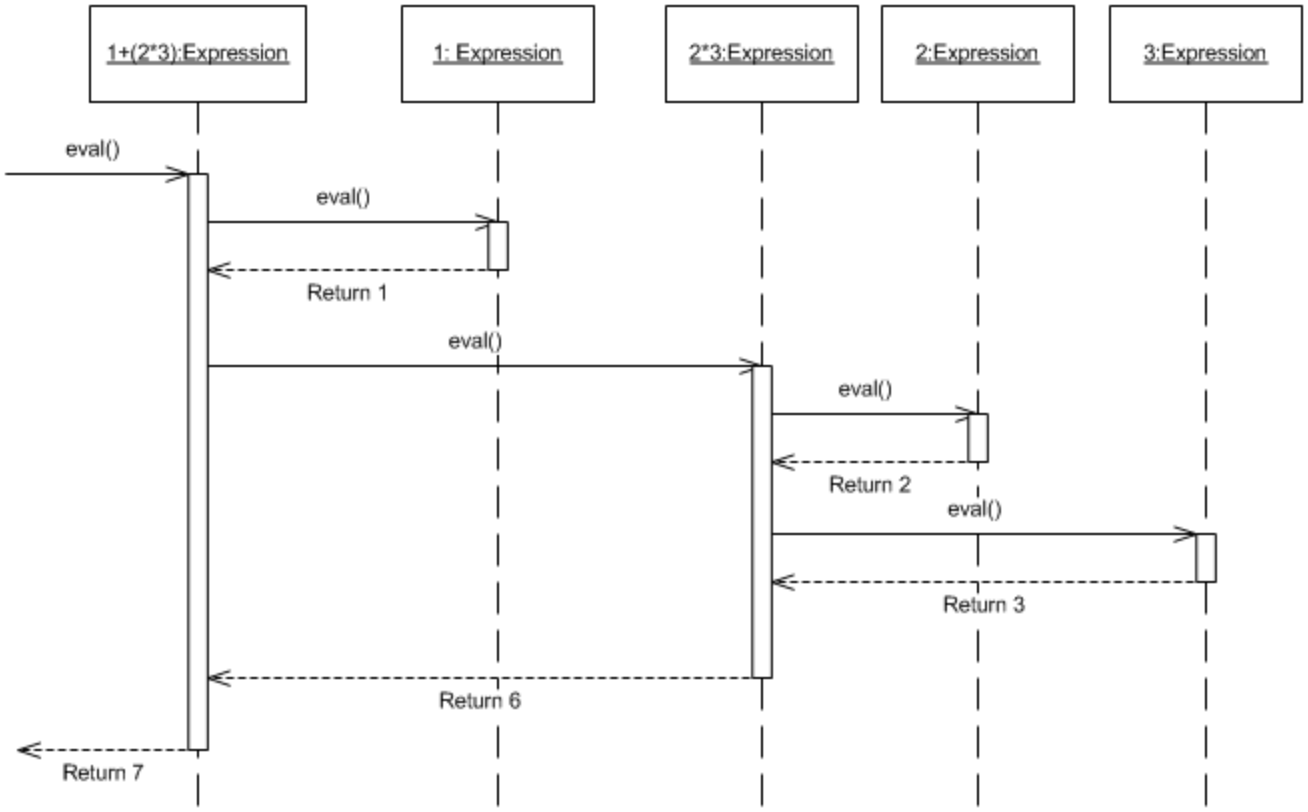


It would be a mistake to show addition (an operation) as an association or special class of objects.

In a CASE tool you can often attach constraints to things in your diagram without cluttering the diagram itself. For example, in Rational Rose, each thing in the diagram has a *specification* attached to it to record details and constraints. They are accessed by double clicking the image. The documentation field in these specifications can include constraints. The Operations Tab has a list of operations. These also have specifications that can be opened up and edited. Constraints can be recorded in the pre-condition and post_condition tabs.

Operational Semantics

Class diagrams with Constraints (above) indicate the structure of a running program. To show what happens as it runs we can use UML Interaction diagrams. The sequence diagram is probably easiest to understand. A Collaboration diagram is better for complicated interactions. For example, suppose that we want to explain how an expression like "1+2*3" is executed, then we would draw the following sequence diagram:

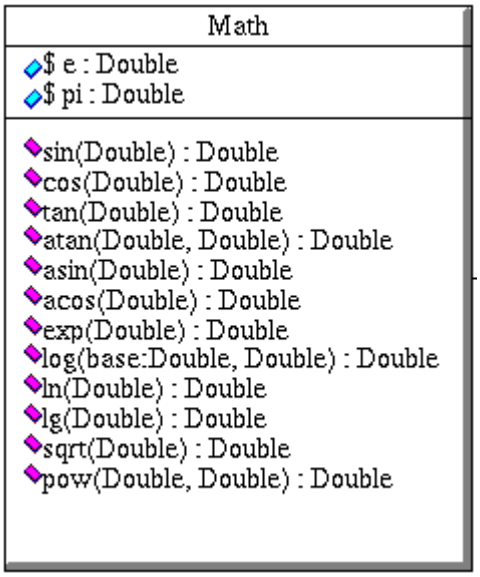


The above shows the objects that represent the formula: three constants (1,2,3) and two operations (2*3, 1+(2*3)). A request to evaluate the expression appears at top left and then each call appears in order and connects the calling object to the called objects. It also shows the values returned as each expression is evaluated.

Part 3: Common Patterns that appear in Programming Languages

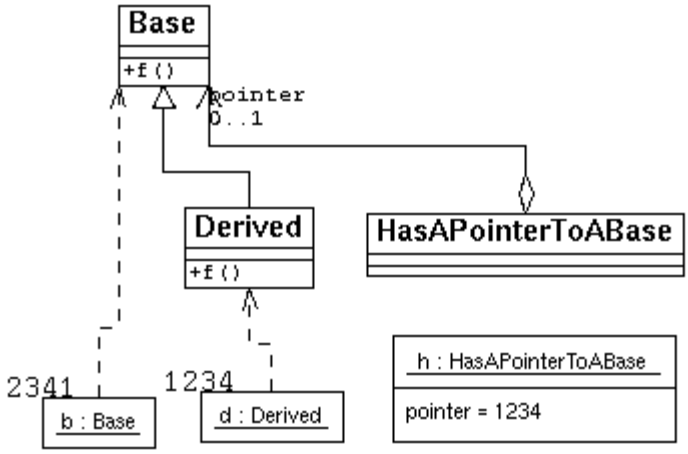
Class Utilities

Some times we want to model a function library (for example <cmath> in C++) or a package of constants (Ada has a standard package of constants used in the sciences). The UML models collections of data and functions that are not attached to objects (`free` functions and `static` data) as Class Utilities. This looks like a normal class with a shadow. You can find an example (based on Java 1.0) below:



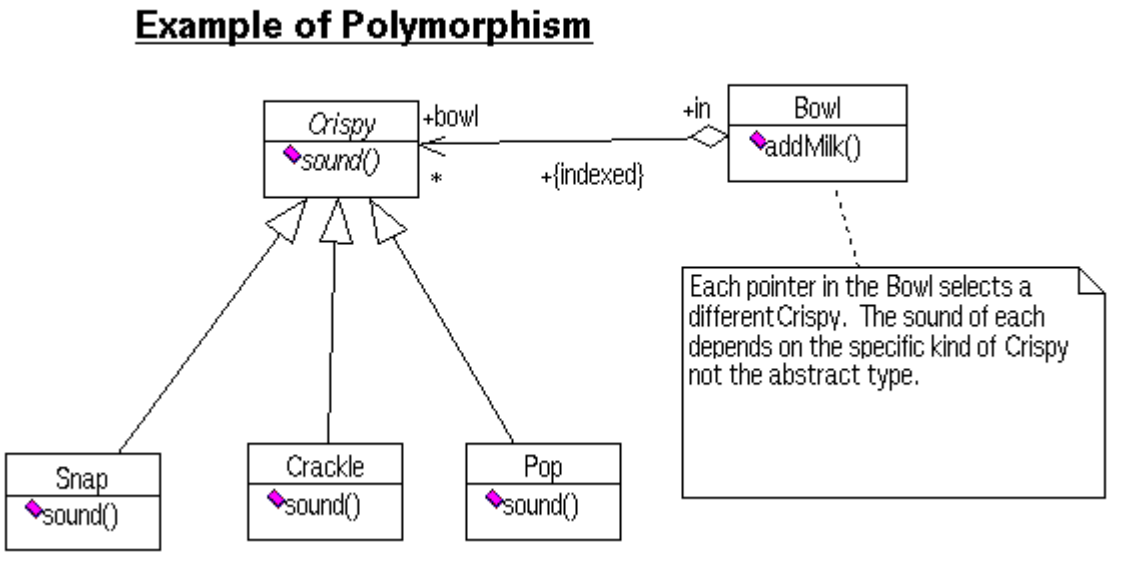
Polymorphism

In the UML it is assumed that when a operation is called and there are several alternative classes that have the operation defined then the object to which the operation is applied always determines the operation that is executed. For example if there is a Base class with a function f() and a Derived class that overrides f() then an object in the Derived class will execute Derived's f, and an object in the Base class executes the Base f. This is true even if the access is via a pointer to the object and if the variable holding the pointer is declared to be of the Base type.




```
d.f() and b.f() do different things.
h.pointer can be 2341 or 1234 and so point at b or d.
If h.pointer is 1234 then h.pointer.f() is d.f().
If it is 2341 the h.pointer.f() is b.f().
In C++ we must write
      'virtual f()' in B and 'h.pointer->f()'.
```

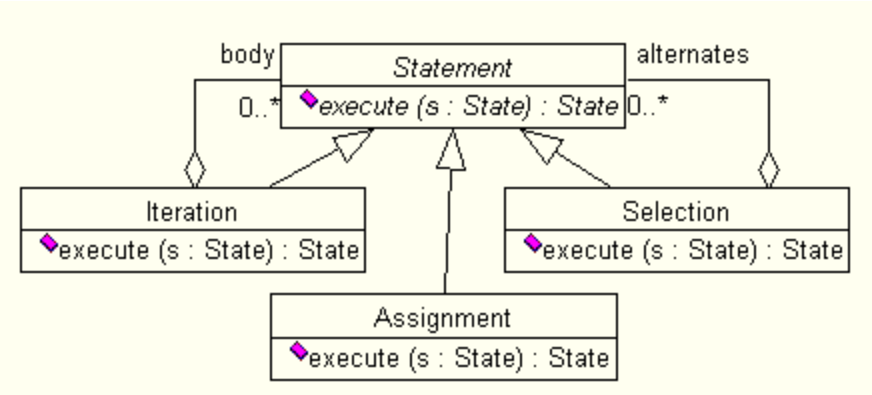
Briefly: the UML models all functions as `virtual`. Here is a nice example involving a bowl of Rice Crispies that all make different sounds:



Here is the C++ code, and a Java application.

Polymorphism is helpful whenever specialized classes of objects behave differently to the general ones. If we have objects that share a number of common operations, but also have their own specific versions then we can show the common properties as a generalization of the special cases -- and show (and specify) operations that are special.

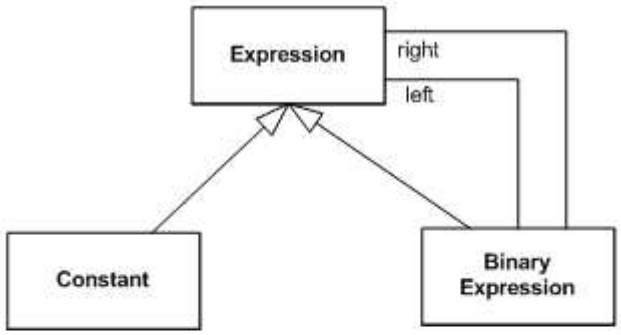
Polymorphism is a powerful tool for simplifying the semantics of a programming language. It is very common to have an abstract Statement that has an abstract *execute* operation. The different types of statement that specialize the general statement (Loop, Selection, Assignment, etc.) each define their own implementation of the *execute* operation. Similarly with Expressions and evaluation. By definition all can be evaluated but different kinds of Expressions implement the evaluation operator in different ways.

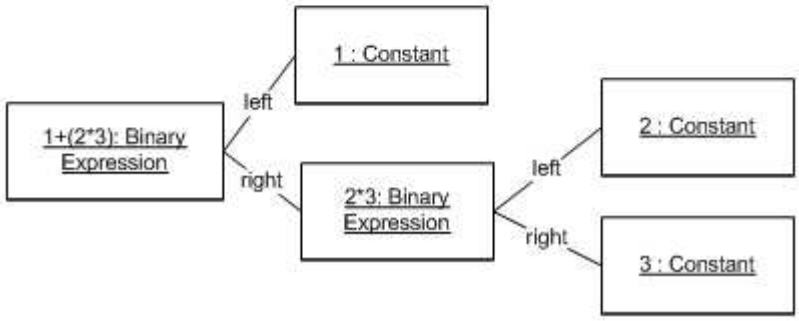


The Composite Pattern

In describing programming languages we often have a situation where an object can be made of other objects from the same class. In fact there was an example in the previous diagram because Selections and Iterations refer to their components. A similar example is when, a statement may be a compound statement, and compound statements contain a number of statements. Similarly an expression often has an operator and a number of operands, and each operand can be any expression. These are all examples that can use the Composite Pattern.

Similar thing happen in other domains as well. For example in modeling a factory we might find that a product is made of parts and these parts are themselves products of the factory. The Composite pattern handles these situations well. For example, in most programming languages an expression can include constants and binary expressions. A binary expressions has two parts. Here is a suitable class diagram and an object diagram illustrating the data structure.





Review Questions

1. What is the meaning of a box in a UML class diagram?
2. How is an association between two classes shown in a UML diagram?
3. How do you show that there is: 1 or 2 items, 0 or 1 items, 0 or more items, and 1 or more items in a UML diagram.
4. Two classes have a many-to-many relationship. How is this shown as a link in UML? What are the two ways of making another class represent the relationship.
5. How are a *role* and an *association* connected in UML?
6. Name the 5 special links made between classes in UML. Sketch the graphic form of each link.
7. State all of the conditions needed to be sure that the *generalization* relation holds between two classes.
8. When should you use the composition relationship in a UML diagram?

9. How does the idea of composition relate to the mathematical term "n-tuple"?
10. What makes you use *composition* when drawing a UML diagram of a data structure? When do you use *aggregation* in a diagram of a data structure?
11. What is the alternative way of showing *composition* in a UML diagram that does not involve a link?
12. What is the effect of putting an arrow head on an *aggregation*?
13. What does an arrow with a dotted line indicate in UML?
14. In a UML box with three compartments what is the meaning of:

    a) the top one, b) the middle one, and c) the bottom one.
15. How do you express the following function specifications in C++ as operations in UML:

    ```
    int a();
    int b(int);
    void c(int,int);
    ```
16. True or false:

    a. Generalization is dynamic.
    b. It does less harm to use composite when it does not apply than vice versa.
    c. Destroying a composition doesn't destroy its parts.
    d. Destroying an aggregation destroys the things it refers to.
17. What is a component and what is an artifact?
18. What does a solid node box show in a deployment diagram?
19. Draw a deployment diagram that shows that (1) the client PC executes the JVM(Java Virtual Machine) and the JVM executes a Java Byte Code applet. (2) The Applet is stored on the Server and depends on the Java Source code that is also stored on the Server.
20. Draw a diagram with three classes: A, B, and C. B and C are special kinds of A. A is a generalization of B and C. The operation fun() is defined for all three classes. However C::fun() is inherited from A::fun() and B::fun() overrides A::fun().
21. Draw an object diagram of an expression like 1*2+3*4 assuming the same structure as above.
22. Draw a sequence diagram showing how 7*6 is evaluated, step by step.