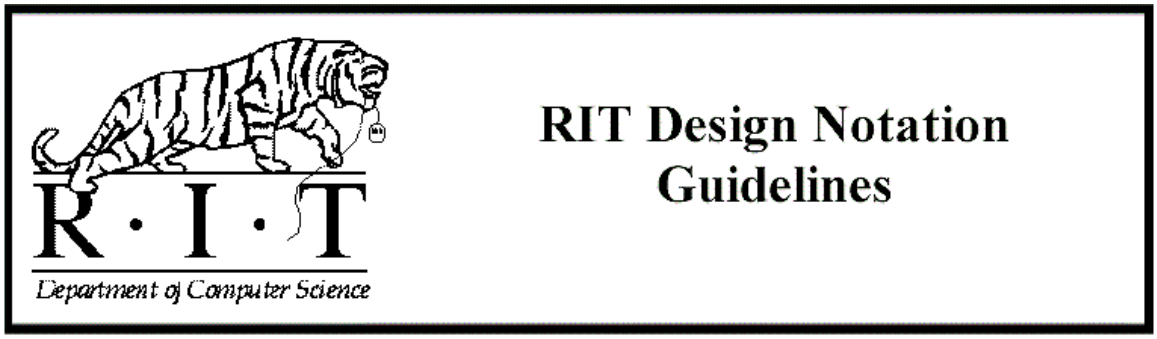

Introduction

A software design is useless if it cannot be clearly, concisely, and correctly described to the programmers writing the code. In the construction industry, architects use blueprints to describe their design to the contractors responsible for constructing the building. Software architects need a similar form of "software blueprints", that is, a standard way of clearly describing the system to be built. Here we introduce the Unified Modeling Language (UML), a graphical modeling language used to visually express the design of a software system. UML provides a way for a software architect to represent a design in a standard format so that a team of programmers can correctly implement the system.

Prior to the development of UML there were many different and incompatible techniques that software architects used to express their designs. Since there was no one universally accepted method, it was difficult for software architects to share their designs with each other, and more importantly with the programming teams responsible for implementing the software. In 1994 Grady Booch, James Rumbaugh and Ivar Jacobson (this group of Computer Scientists is often referred to as the "three amigos") started work on UML. One of the goals of UML was to provide a unified way of modeling any large system, not just software, using object-oriented design techniques. In order for UML to be widely used it was important that the language be available to everyone. Therefore, the resulting language is a nonproprietary industrial standard and open to all.

There are several aspects of a system that need to be described in a design. For example, the functional aspects of a system describe the static structure and the dynamic interactions between components in the system, whereas non-functional aspects include items such as timing requirements, reliability, or deployment strategies. In order to provide a way to describe all of the relevant aspects of a software system, UML provides five different views that document the various aspects of a system. A **view** is an abstraction consisting of a number of diagrams that highlight a particular aspect of the system. The five views provided by UML are summarized in Table 1 below:

View	*Description*
Use-Case	Describes the functionality that the system should deliver as perceived by external actors (users). Used to document the requirements and specifications of a system.
Logical	Illustrates how the functionality of the system will be implemented in terms of the system's static structure and dynamic behavior.
Component	Shows the organization of the code components.
Concurrency	Describes the concurrency a system, addressing the problems with communication and synchronization that are present in a concurrent system.
Deployment	Illustrates the deployment of the system into physical architecture with computers and devices called nodes.

Table 1 UML Views

In addition to the five views of Table 1, UML defines nine different **diagram types** that describe specific aspects of the system. Since a single diagram cannot possibly capture all the information required to describe a system, each UML view will consist of several diagrams that describe various aspects of the system. The names of the nine types of UML diagrams, a brief explanation of each type, and the view in which the diagram is typically used is given in Table 2.

Name	*Description*	*Views*
Use-case	Captures a typical interaction between a user and a computing system. Useful when defining the user's view of the system.	Use-case
Class	Describes the classes that make up a system and the various kinds of static relationships that exist among them.	Logical
Object	A variant of the class diagram except that an object diagram shows a number of instances of classes, instead of the actual classes.	Logical
State	Describes all the possible states that a particular object can get into and how the object's state changes as a result of messages sent to the object.	Logical, Concurrency
Sequence	Describes how a group of objects collaborate in some behavior concentrating on the messages sent to elicit that behavior.	Logical, Concurrency
Collaboration	Describes how a group of objects collaborate in some behavior concentrating on the static connections	Logical, Concurrency

	behavior concentrating on the static connections between the objects.	Concurrency
Activity	An activity diagram shows a sequential flow of activities performed in an operation.	Logical, Concurrency
Component	Describes the physical structure of the code in terms of code components.	Concurrency, Component
Deployment	Shows the physical architecture of the software and hardware components that make up a system.	Concurrency, Deployment

Table 2 UML Diagrams

UML is a powerful tool that has many features and can be used to express very complicated designs. At this point we are only interested in the logical view (the static structure and dynamic behavior) of a system. Therefore we will make extensive use of class, object, state and sequence diagrams when expressing our designs.

Unlike Java, UML does not have rigid rules regarding what must or must not be included in a diagram and it allows a software architect to determine how much detail to include in the final diagram. When writing up your designs keep in mind what you are trying to illustrate and who will be using your diagrams. A programmer will require very detailed information regarding the state and behavior of an object, whereas a system analyst may only require a general description of the classes that make up the system. Your diagram should provide enough information to illustrate your design, but not so much detail that a reader gets lost. The structure of the UML should be based on the needs of the individuals reading the document.

Class Diagrams

A class diagram in UML is used to describe the static structure of a system in terms of its classes and the relationships among those classes. Class diagrams are the most common way to describe the design of an object-oriented system, and you will find yourself using them all the time. Class diagrams, like UML, are very expressive and provide ways to describe even the subtlest aspects of a class. In order to avoid becoming lost in the details, we will only describe the more commonly used features of class diagrams. As you gain more experience you will almost certainly want to read more about the advanced features of class diagrams, as they can be very useful. There are many excellent references on UML, some of which are listed at the end of this document.

It may not be possible, or even desirable, to use a single class diagram to describe a complete system. It is better to concentrate on key areas of the design and then document these ideas using different class diagrams. Keeping the diagrams simple and using them to convey key concepts of a design can be much more effective than using the "shotgun" approach of describing everything in as compact a space as possible.

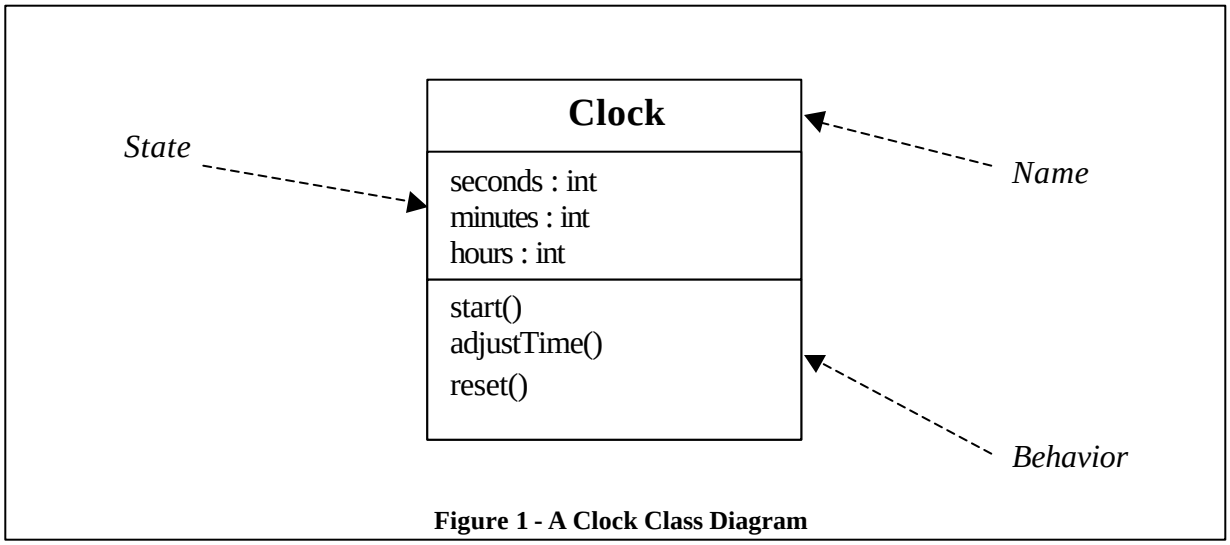

Figure 1 - A Clock Class Diagram

A class, in a class diagram, is drawn as a rectangle that can be divided into three compartments, as shown in Figure 1. The name of the class appears in bold text centered in the compartment at the top of the rectangle. The compartments that describe the state and behavior of the class are optional, as shown in Figure 2a. Type information for the methods that make up the behavior of the class is optional, however, the names of routines that take parameters must be followed by an open and closed parentheses (even if you choose not to put anything inside of them). Parameter and return types for behaviors may be specified using the colon notation shown in Figures 2b and 2c.

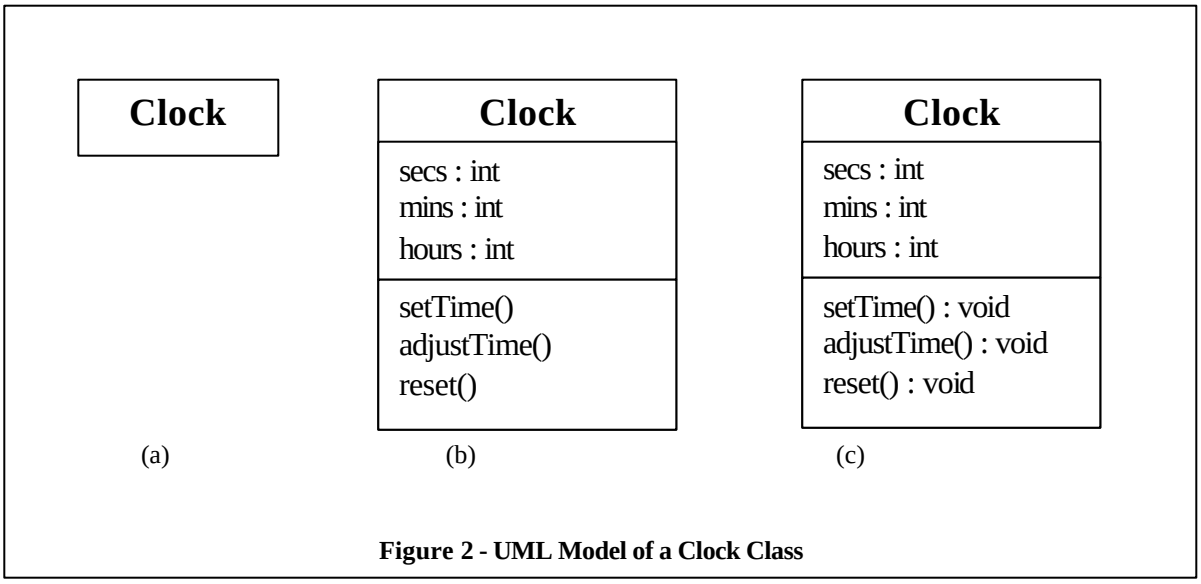

Figure 2 - UML Model of a Clock Class

When drawing a class diagram you want to include only as much information as a reader needs to understand your design. You do not want to overwhelm a reader with trivial

detail when it is not required. For example, most class diagrams do not contain obvious behaviors, such as accessors or mutators, so that the non-obvious behaviors in the class are easier to recognize. Similarly we may choose to omit such relatively unimportant details as `void` return values, as shown in Figure 2b and 2c. Classes whose behaviors are well known, such as classes provided in a system library, will either be drawn as a single rectangle with no compartments as in Figure 2a, or omitted from the diagram altogether.

Describing the classes that are present in a software system does not provide enough information for a programmer to understand how these classes work together in a program. For example, if you were asked to describe a family it would not be enough to say, "A family consists of parents and children." Additional information must be provided to describe the nature of the relationship between parents and children. To completely specify the static structure of a system, the classes and the nature of the relationships between the classes must be defined.

A relationship exists between two classes if one class "knows" about the other. In most object-oriented programming languages, a relationship exists between two classes if an instance of one class invokes a method or accesses the state of an instance of the other class. Knowing that a relationship exists between two classes does not provide any information about the *nature* of the relationship. For example, the relationship that exists between the parents in a family is considerably different than the relationship that exists between the parents and the children. UML defines several different types of relationships that can be used to describe the way in which two or more classes are related in a class diagram. Here we use three different types of relationships: associations, dependencies, and generalizations.

An **association** is a relationship that ties two or more classes together, and represents a structural relationship between instances of a class. An association indicates that objects of one class are connected to objects of another. This connection is permanent and makes up part of the state of one of the associated classes. From a programming perspective two classes are associated if the state of one class contains a reference to an instance of the other class.

In a home heating system, for example, relationships exist between the thermostat, heater, and users of the system. The relationship that exists between the thermostat and heater makes up the structure of the system. At anytime while the heating system is in existence, the thermostat will know about the heater. This type of relationship, that represents a permanent structural relationship between two classes, would be classified as an association. The relationship between the thermostat and heater is clearly different than the relationship that exists between a user and the thermostat. A user is an important part of the system and "uses" the thermostat to adjust the temperature in a room however, a user does not make up part of the structure of the system nor is the system always associated with a user. The heating system in a room continues to function whether or not we are in the room.

For another example of association, consider the relationship that exists between instances of the `Engine` and `Car` classes in an automotive simulation program. The relationship that exists between instances of the `Engine` and `Car` classes form part of the structure of a `Car`. As long a `Car` exists, the `Car` will know about, or be in a relationship with the `Engine`. This is not true about the relationship between the `Car` and `Driver` classes. At night when we are sleeping and our van is in the garage, our van can still function as a car. On the other hand should someone break into our garage and remove the engine from our van, it will no longer be capable of functioning as a car. Therefore, although relationships exist between these classes, there is an association between the `Engine` and `Car` classes, but not between the `Driver` and `Car` classes.

In UML a solid line is drawn between two classes to represent an association. The UML class diagram in Figure 3 specifies that the relationship between the `Car` and `Engine` classes in the automotive simulation program is an association.

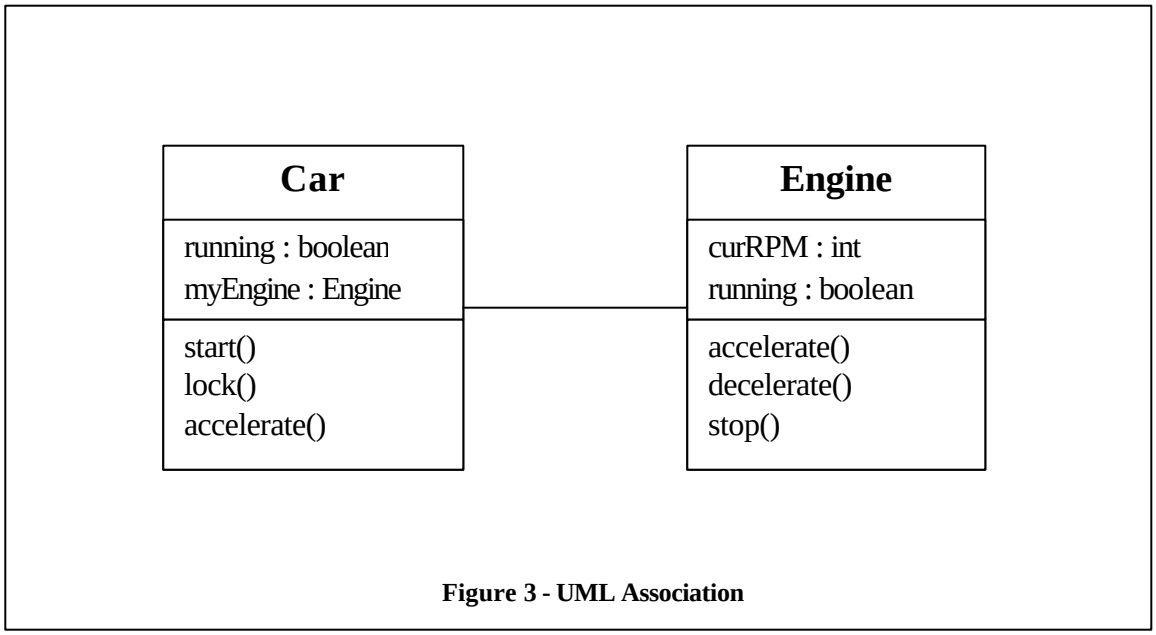

Figure 3 - UML Association

The association between the `Car` and `Engine` of Figure 3 goes in both directions - the `Car` knows about the `Engine` and the `Engine` knows about the `Car`. Associations, however, are not always bi-directional. Consider the relationship between the `Engine` and `GasPedal` classes. This relationship can be described as an association because it is permanent and is part of the structure of a `Car`. Unlike the association between the `Car` and `Engine` classes, the association between the `GasPedal` and the `Engine` does not go in both directions. The `GasPedal` knows about the `Engine` since it invokes the `accelerate()` method of the `Engine`, however, the `Engine` does not know about the `GasPedal` since it never invokes a method on that class.

Navigability information can be included in a UML class diagram to clarify the nature of the relationship between two classes. As shown in Figure 4, an arrow is added to the

solid line that represents an association to indicate the *direction* of a relationship. In the diagram of Figure 4 the arrow indicates that the `GasPedal` knows about the `Engine`, but the `Engine` does not know about the `GasPedal`.

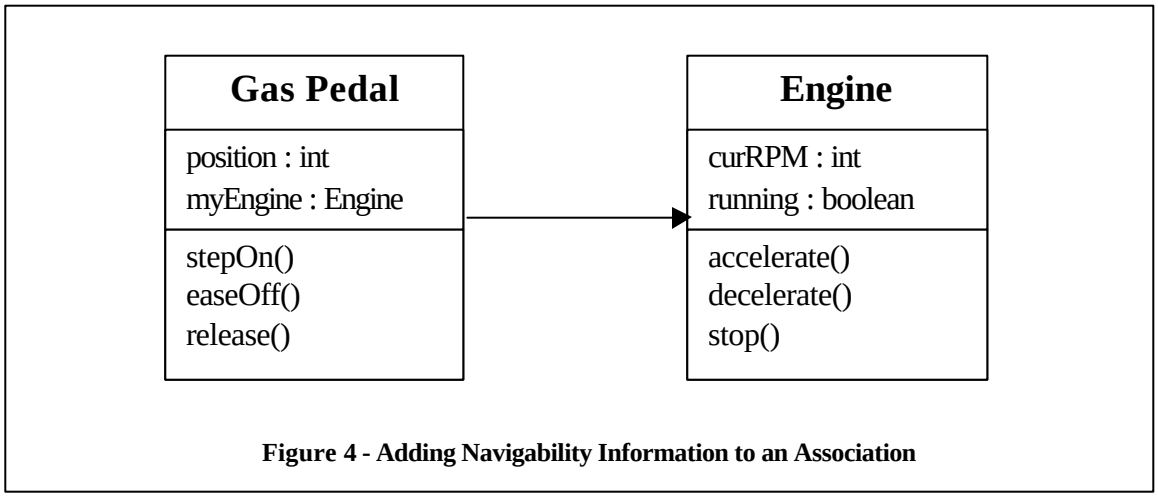

Figure 4 - Adding Navigability Information to an Association

In addition to navigability, **multiplicity** can be used to describe the nature of a relationship between classes. Consider the relationship between the "Automated Teller Machine" (`ATM`) and `Bank` classes in an electronic banking system. This relationship is an association since the relationship is structural and permanent (i.e., the `ATM` always needs to know about the `Bank`). Furthermore, the association is one way since the `ATM` knows about (i.e., invokes methods on) the `Bank`, but the `Bank` does invoke methods on the `ATM`. However, there are likely to be several instances of the `ATM` class, whereas there will only be one instance of the `Bank` class. This last bit of information can be included in a UML class diagram by adding multiplicity information to the association.

Multiplicity information has been added to the UML diagram in Figure 5. The '*' indicates that there may be zero, one, or more instances of the `ATM` class. The '1' next to the `Bank` class indicates there is exactly one bank in the system.

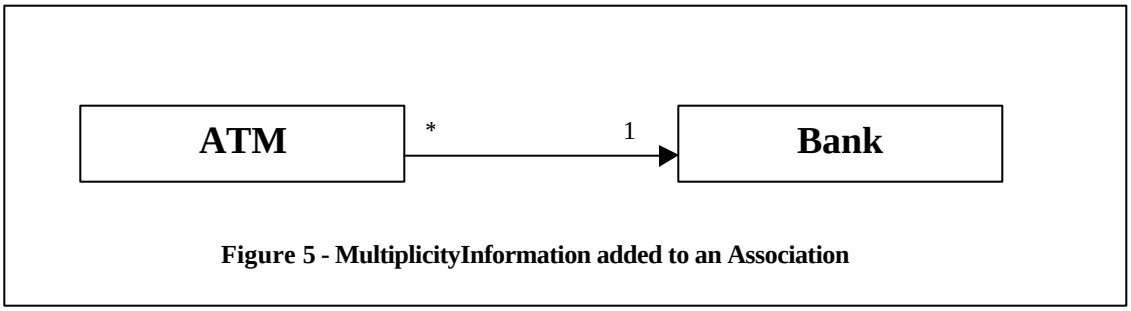

Figure 5 - MultiplicityInformation added to an Association

Let's use association, navigability, and multiplicity to model an inventory system. Our design consists of two classes: `Inventory` and `Item`. The `Inventory` class is responsible for storing items and providing methods to add, delete, locate, and modify items in the collection. The `Item` class captures the state associated with the items

owned by each department. The UML class diagram, shown in figure 6, models this design.

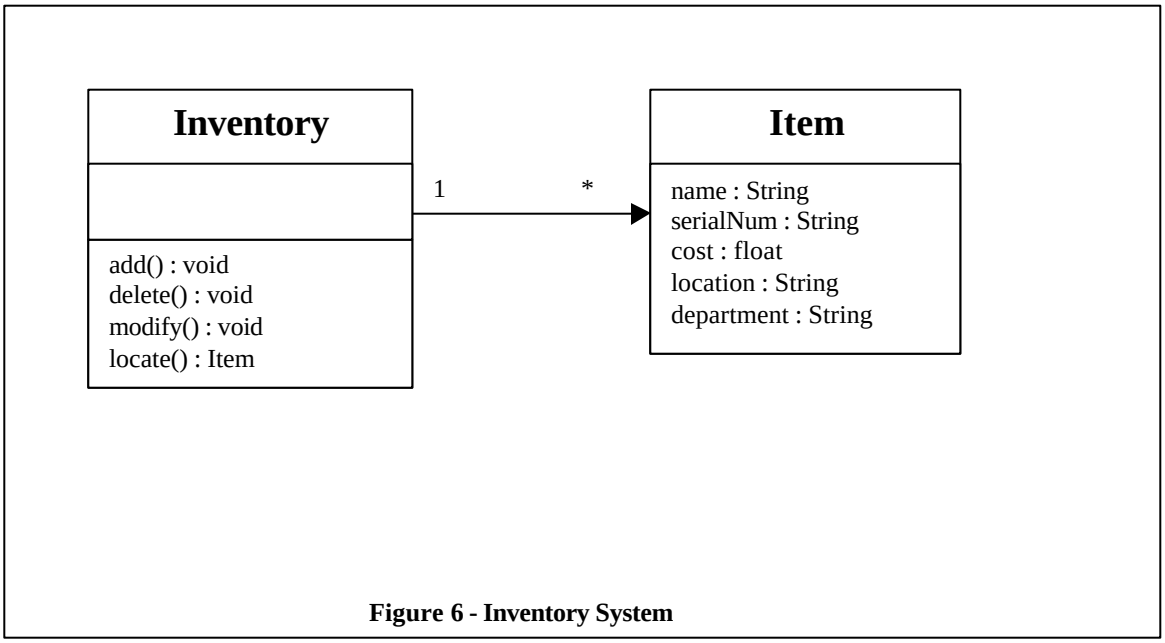

Figure 6 - Inventory System

The class diagram in Figure 6 includes a description of the two classes, `Inventory` and `Item`, which make up the inventory system. Since the behaviors that are associated with the `Item` class consist exclusively of accessors and mutators, the type of methods that you would expect to find in a class such as this, they have been omitted from the diagram. Furthermore the type of data structure used to implement the `Inventory` class will not affect the design of the system so it has been omitted to make the diagram easier to understand.

The relationship between the `Inventory` and `Item` classes is drawn as an association because it describes the structure of the system (i.e., the `Inventory` consists of `Items`). In this case the `Inventory` will not invoke methods of the `Item` class, but clearly the `Inventory` must know about the items. The navigation information specifies that the `Inventory` knows about the `Item`s, but that the `Item`s do not know about the `Inventory`. Finally, the multiplicity information specifies that a single `Inventory` will hold zero, one, or more items and that the system consists of exactly one inventory object.

The second type of relationship that we will discuss is a **dependency**. A dependency is a using relationship that specifies that a change in one class may affect another class that uses it. A dependency is inherently a one-way relationship where one class is dependent on the other. Returning to the automotive simulation program, clearly a relationship exists between the `Car` and `Driver` classes, but this relationship should not be

classified as an association because it does not constitute part of the structure of the system. It would be more descriptive to indicate that the `Driver` uses the `Car` since should the `Car` class change (perhaps the car no longer has an automatic transmission) the `Driver` may need to change in order to use the `Car`.

The difference between an association and a dependency can be made a little clearer by looking at the way these relationships are implemented in a program. Associations are usually implemented as part of the state of one of the classes. For example, in the code that implements the state of the `GasPedal` class, you would expect to find a variable that refers to the `Engine` object that the `GasPedal` controls. As long as the `Car` is in existence, a `GasPedal` object will always be associated with an `Engine`. The state variable provides a mechanism whereby the `GasPedal` can access the `Engine`.

A dependency typically takes the form of a local variable, parameter, or return value in the methods that implement the behavior of the object. These types of variables are often referred to as *automatic* since they are created and destroyed as needed. If the variable is in scope, in other words exists, a class has a way to access a class it depends on. So at some point in the lifetime of the object it may know about an instance of a class with which it is related and at other times it will not. This reflects the transient, or non-permanent, nature of a dependency. In the automotive simulation program you would expect that the `Driver` has a method named `driveCar()` that takes as a parameter a reference to the `Car`, which is to be driven. The `Car` is clearly not part of the state of the `Driver`. Furthermore a `Driver` only knows about a specific `Car` when they are physically driving it.

In a UML class diagram a dependency is drawn as a dashed line as shown in Figure 7. The arrow on the line points to the independent element. In Figure 7 the arrow captures the fact that should the `Car` change, the `Driver` may need to change, but if the `Driver` changes, the `Car` will not be affected.

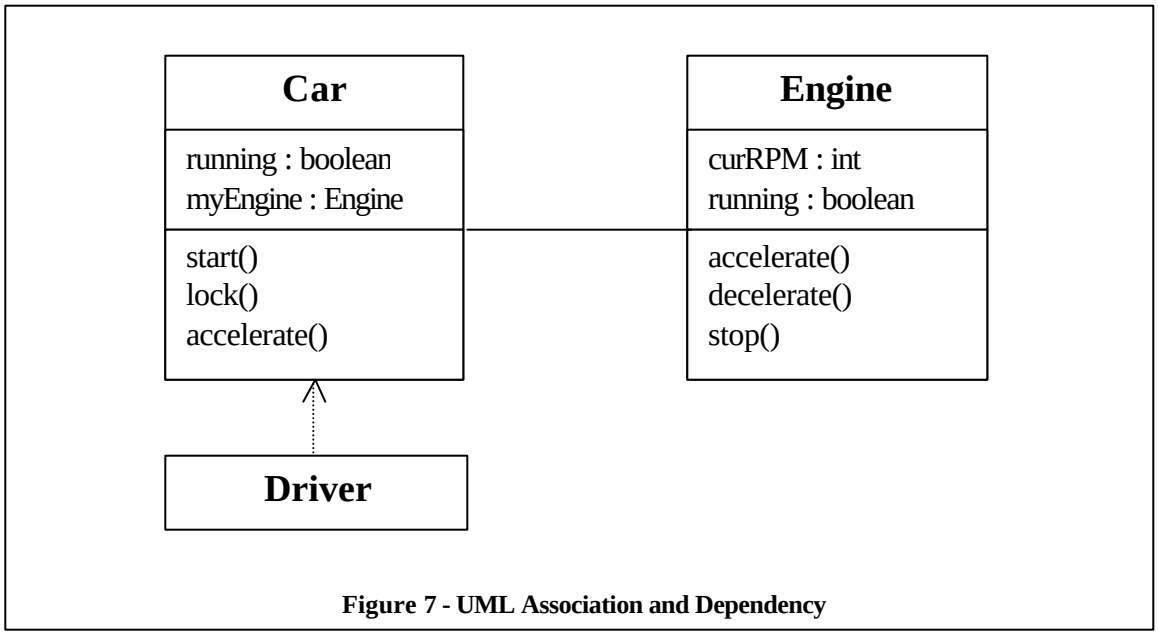

Figure 7 - UML Association and Dependency

Generalization, or inheritance, is the third type of relationship that will be used in this text. In an inheritance relationship a subclass is a specialized form of the superclass. If you look at the relationship from the superclass' perspective, you could say that the superclass is a generalization of its subclasses. This generalization relationship is denoted by a triangle connecting a subclass to its parent class. The triangle is connected to and points at the parent class as shown in Figure 8.

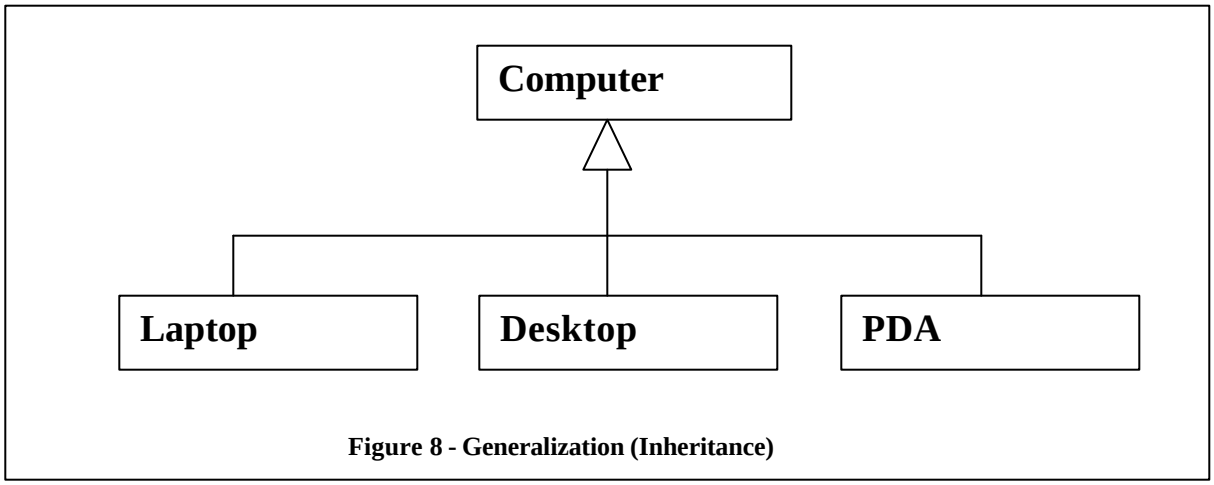

Figure 8 - Generalization (Inheritance)

Both associations and generalizations (inheritance) can be illustrated in a single diagram as shown in Figure 9. This diagram describes the relationships between a processor, disk controller, and disk drive. Here the `CPU` is associated with the class `Controller` that describes all controllers. `SCSIController`, a sub-class of `Controller`, is the class that is associated with `DiskDrive`. In a class diagram, associations should only be

shown at the highest possible level. For example, in Figure 9 the association is between the `CPU` and `Controller` (the superclass) and not between the `CPU` and the `SCSIController` (the subclass). This indicates that the `CPU` is designed to work with any `Controller` not just a `SCSIController`.

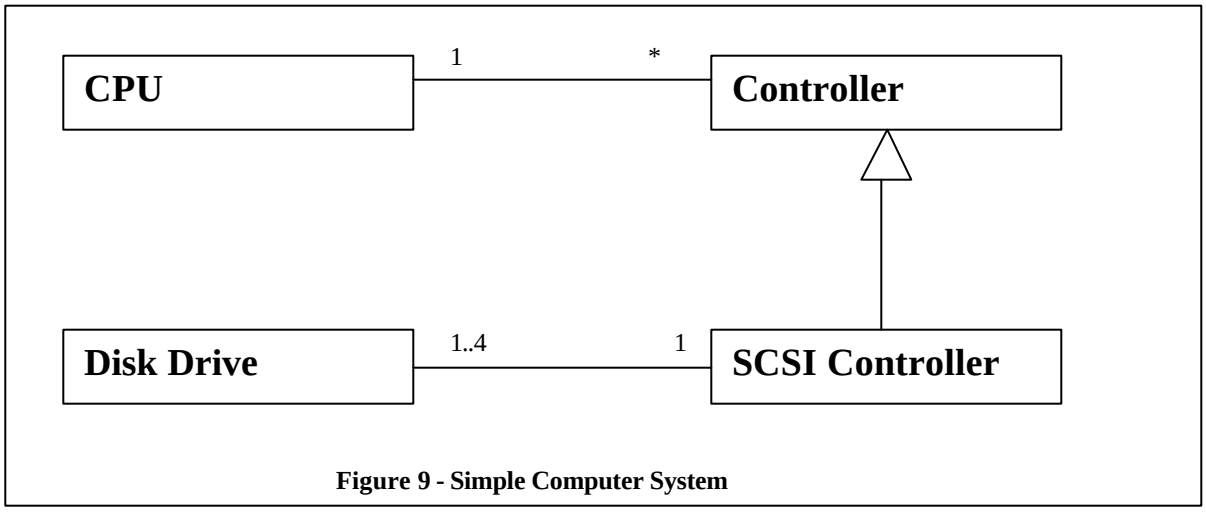

Figure 9 - Simple Computer System

Example Class Diagrams

The best way to master the basics of class diagrams is to use them to model simple systems. This section presents four different UML class diagrams along with a description of the system that each diagram models. In order to improve your understanding of class diagrams, take a few minutes to review each UML class diagram before reading the description of the diagram. Write down what you believe is the description of the system being modeled. Then read the description of the system that follows and reconcile your description with the one given in this document.

The first class diagram that models two classes in a system used by a veterinarian to track patients (i.e., pets) is given in Figure 10 below:

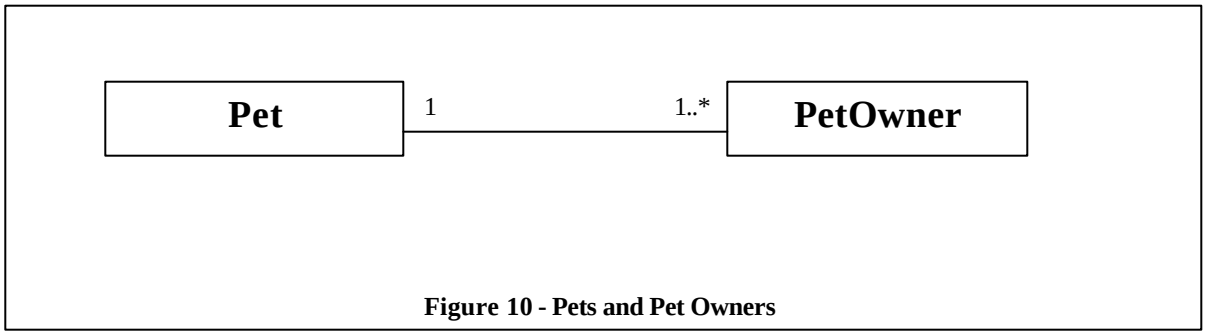

Figure 10 - Pets and Pet Owners

`Pet` and `PetOwner` are two of the classes in the veterinarian animal tracking software system. The class `Pet` includes all the state that the system maintains for a single animal, and the class `PetOwner` contains the state associated with the owner of a `Pet`.

In the UML class diagram of Figure 10, the state and behavior for these classes have been omitted since the only thing of interest here is the relationship between the two classes. The state of every `Pet` object contains a reference to its owner. Since this relationship provides structural information and instances of the `Pet` class will always contain a reference to a `PetOwner`, the relationship is modeled as an association.

The absence of the navigability information in the drawing indicates that the association is bi-directional. In other words, a `Pet` knows about its owner and a `PetOwner` knows about its pet. Finally, the multiplicity information shows that every `Pet` has exactly one owner, and every `PetOwner` has one or more `Pet`s. If the multiplicity information was omitted, nothing could be said about the number of `PetOwner`s associated with a `Pet`, and the number of `Pet`s associated with a `PetOwner`.

The next class diagram that will be discussed is shown in Figure 11 below:

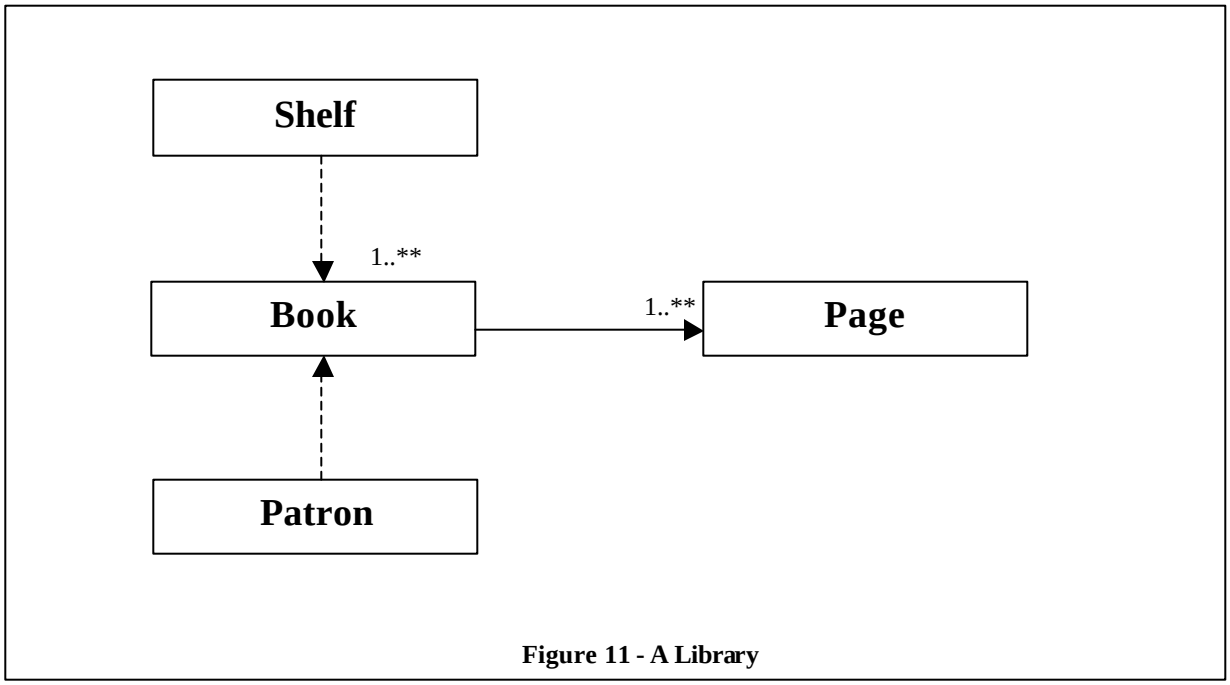

Figure 11 - A Library

The class diagram in Figure 11 describes the relationships that exists between books, pages of a book, shelves, and the patrons of a typical library. This diagram indicates that a Book contains one or more pages. This relationship has been modelled as an association because the pages are actually part of the book. If we rip the pages out of the book, that book is no longer a book. Note, however, that whether the pages are in the book or not, they are still pages. This is the reason why the solid line that specifies the association between the `Book` and `Page` classes has an arrow that points to the `Page` class. This indicates that the `Book` class is associated with the `Page` class and not the other way arround. Compare this to the relationship that exists between the `Shelf` and `Book` classes. Clearly a book it not part of a shelf, and a shelf is not part of a book, however, should the properties of a book change (becomes taller, heavier, etc.) the shelf

may have to change to accommodate the book. This means that in this relationship the `Shelf` class is dependent upon the `Book` class. The arrow specifies the independent class (i.e., the class that does not have to change).

The UML class diagram, in Figure 12 models the types of accounts provided by a typical bank.

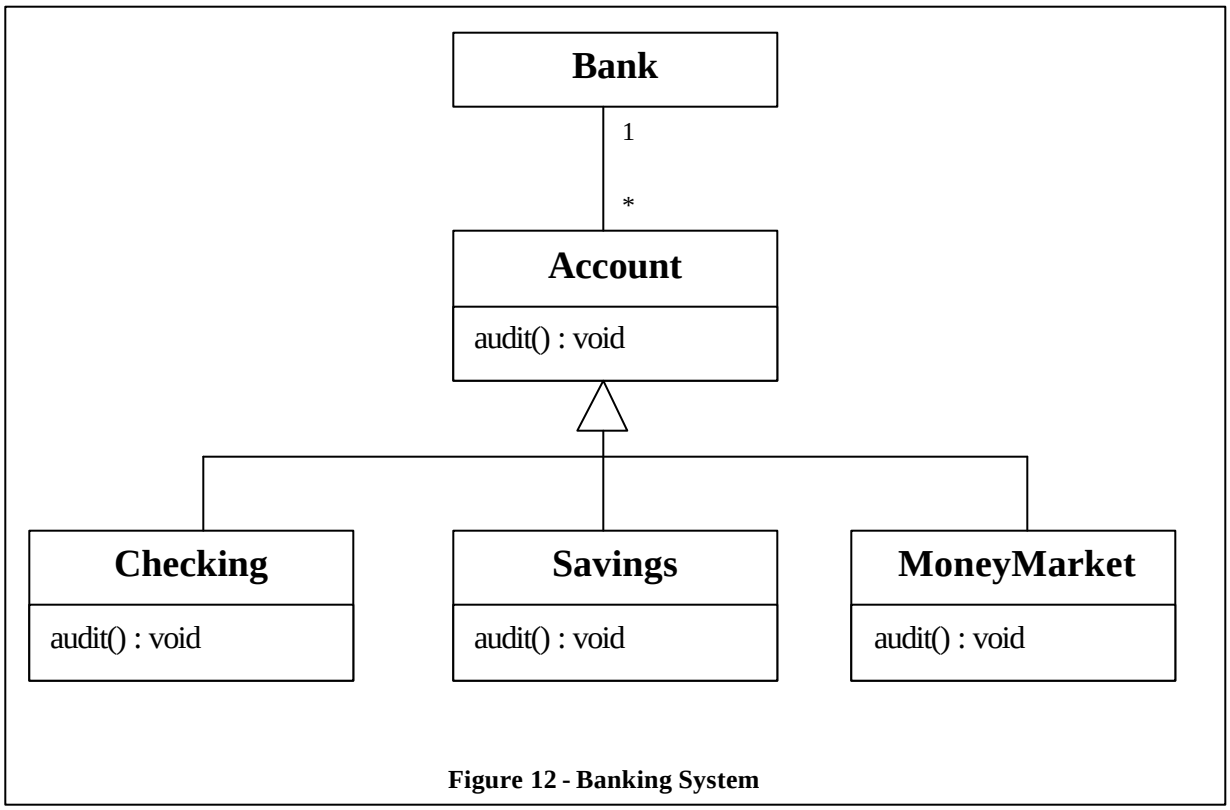

Figure 12 - Banking System

In this system a bank is associated with, or has, one or more accounts. The lack of navigability information indicates that the association goes in both directions. Clearly the bank knows about its accounts, but the accounts also know about the bank. This means that an account can utilize the services provided by its bank, perhaps obtaining the current interest rate set by the Federal Reserve Bank. An account is a generalization of three classes: Checking, Saving and Money Market. The `audit()` method is defined in the `Account` class and is a behavior that all of its subclasses will have.

The class diagram in Figure 13 models a simple home heating system.

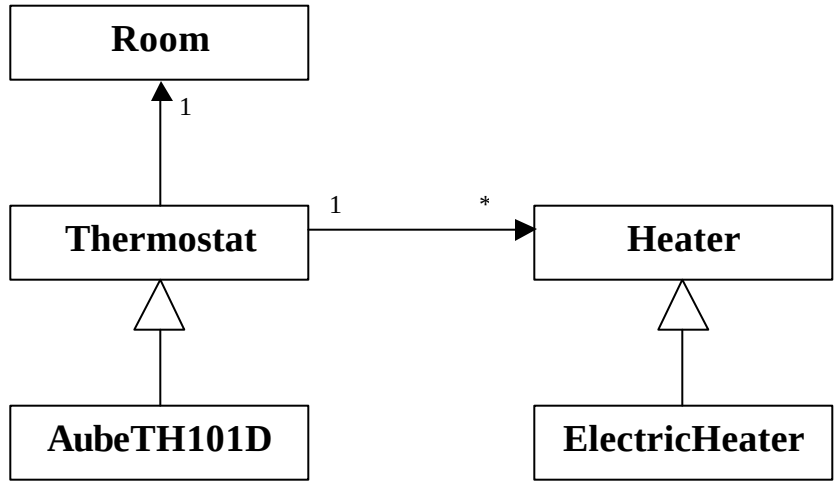

Figure 13 - Home Heating System

UML class diagrams are an excellent tool for capturing the static aspects of a system, namely the classes in the system and the relationships between classes. What we have not talked about yet is how we capture the dynamic aspects of a system. Using a class diagram it is not possible to document that in the home heating system of Figure 13 the thermostat queries the room for the current temperature and turns on the heaters if necessary. A class diagram can show that a thermostat knows about a room and knows about the heaters, but it cannot say anything about what specific interactions the thermostat has with these items. In the next section we introduce UML sequence diagrams that are used to describe the dynamic behavior of an object-oriented system.

Sequence Diagrams

Sequence diagrams describe how groups of objects dynamically collaborate with each other. Typically, a single sequence diagram describes a single behavior. It diagrams a number of objects and the messages passed betweens these objects. Sequence diagrams provide a way to describe the dynamic behavior of a system.

To illustrate some of the basic features of sequence diagrams, here we will model the behavior of a printing system. The system consists of a number of printers, each of which has different resources; the size and type of paper it is currently holding, or the ability to print in color. Each printer is serviced by a print queue that holds the jobs for this printer. Information about the printer resources, and the location of the queues that

service the printers is maintained by the printer registry. The class diagram for this system is given in Figure 14.

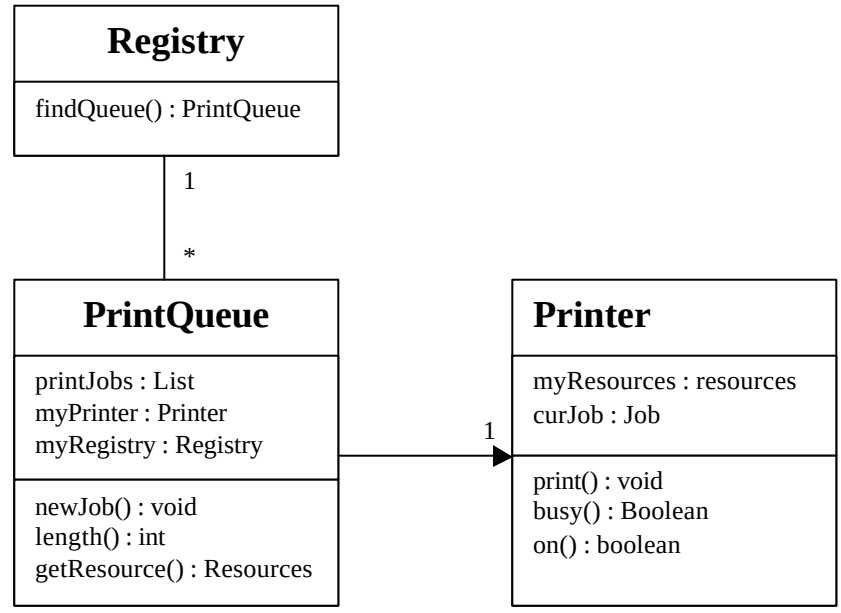

Figure 14 - Printing System

Although the class diagram in Figure 14 explains the static structure of the printing system, it does not explain how the objects collaborate to achieve a specific behavior. For example, a question that a programmer needs to ask when implementing this system is "How does a job find a printer with the set of resources it requires?" The answer is that when a print job needs to be sent to a printer, a message is sent to the registry that contains the job to be printed and a list of the required resources. The registry then searches each of its print queues looking for the first printer that has the resources required to print this job. If such a printer exists, the registry sends a `newJob()` message to the print queue, which in turn will send a `print()` message to the correct `Printer`. This process is described in the sequence diagram given in Figure 15.

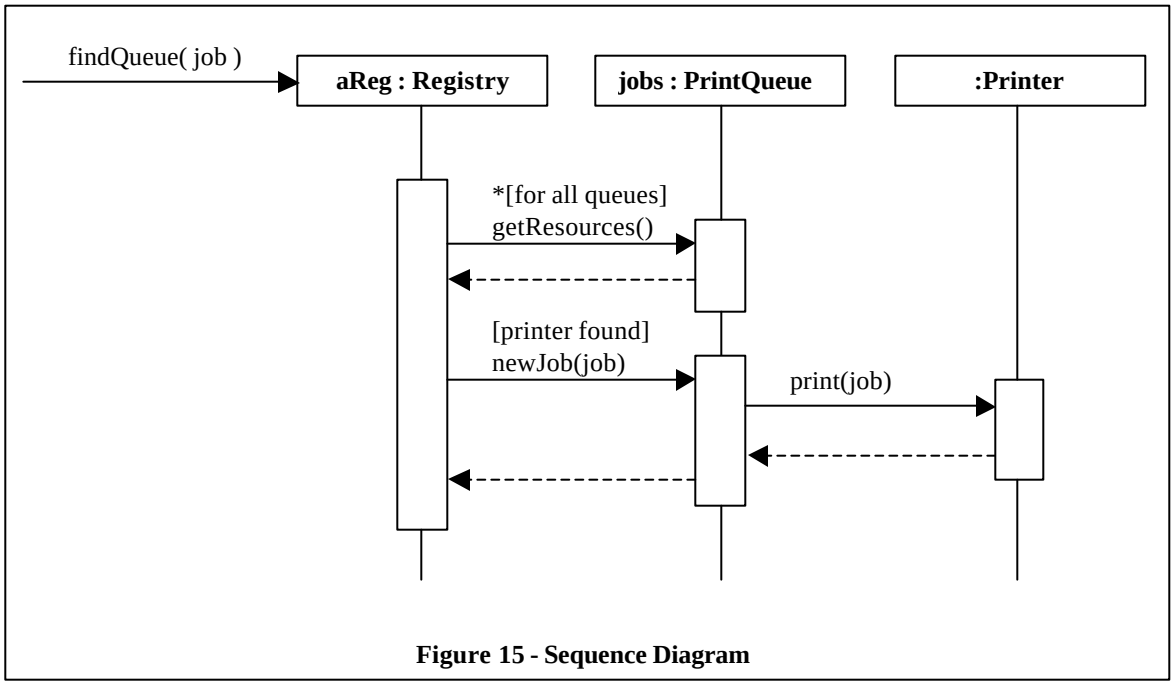

Figure 15 - Sequence Diagram

Time flows from top to bottom in a sequence diagram and the vertical lines that run down the diagram denote the lifetime of an object. Since a sequence diagram illustrates objects, as opposed to classes, a slightly different labelling convention is used. In a sequence diagram each line is labelled with the name of the object, followed by the name of the class from which the object is instantiated. A colon is used to separate the name of the object from the name of the class. It is not necessary to supply names for all of the objects, however, the class name must always be given. When labelling a line using only a class name, the class name must be preceeded by a colon.

A vertical rectangle shows that an object is active; that is, handling a request. The object can send requests to other objects, which is indicated by a horizontal arrow. An object can send itself requests, this is indicated with an arrow that points back to the object. The dashed horizontal arrow indicates a return from a message, not a new message. In Figure 15 the `PrintQueue` sends the `print(job)` message to a `Printer` which is followed by a return. Note that after handling the message the `Printer` object is no longer active and therefore the vertical rectangle ends.

Two forms of control information can be placed in a sequence diagram: condition and iteration. The condition marker indicates when a message is sent. The condition appears in square brackets above the message: for example, `[printer found]`. The message is only sent if the condition is true. The iteration marker indicates that a message is sent many times to multiple receiver objects. The basis of the iteration appears within square brackets immediately preceded by an asterisk ('*'). In Figure 15 the control information `*[for all queues]` indicates that the `Registry` will send a `getResources()`

message to each of the `PrintQueue` objects that it is managing. The condition marker `[printer found]` specifies that the print job will be sent to a printer only if a printer with the correct resources is located.

A sequence diagram illustrates the sequencing of events in an object-oriented system. It does not show the algorithms that are involved, only the order in which messages are sent. Each of the classes that appear in a sequence diagram should be described in a separate class diagram. Note that if a sequence diagram indicates that an object sends a message to another object, then in the corresponding class diagram there must be a relationship, either an association or dependency, between those two classes. You should include a separate sequence diagram for each of the major behaviors in the system being modelled. What is a "major" behavior? It depends on your audience and what information you are trying to convey in your drawing, but as a guideline you should include those behaviors that are central to an understanding of the software being developed. For example, in our home heating system, it would be essential to understand the behaviors associated with obtaining the room temperature and activating the heaters. However, the behaviors associated with testing the thermostat or putting it into "Vacation Mode" would be less important. You must be very careful not to hide the central aspects of the the design by including a great deal of unnecessary information.

Summary

In this document we introducted the Unified Modeling Language (UML), which is a graphical modeling language that can be used to visually express the design of a software system. UML provides a way for a software architect to represent a design in a standard format so that a team of programmers can correctly implement the system. UML is a powerful tool that has many features and can be used to express very complicated designs. In this document we only described class and sequence diagrams.

A class diagram in UML is used to describe the static structure of a system in terms of its classes and the relationships among those classes. A relationship exists between two classes if one class "knows" about the other. In this document we described three different types of relationships: associations, dependencies, and generalizations. A sequence diagram describes how groups of objects dynamically collaborate with each other. Sequence diagrams provide a way to describe the dynamic behavior of a system.

When constructing UML diagrams remember that there are no rigid rules regarding what must or must not be included in a diagram. It is up to you to decide how much detail to include in your diagrams. When documenting your designs using UML keep in mind what you are trying to illustrate and who will be using your diagrams. When in doubt ask you instructor for additional guidance.

The features of UML introduced in this document are summarized in Figure 16.

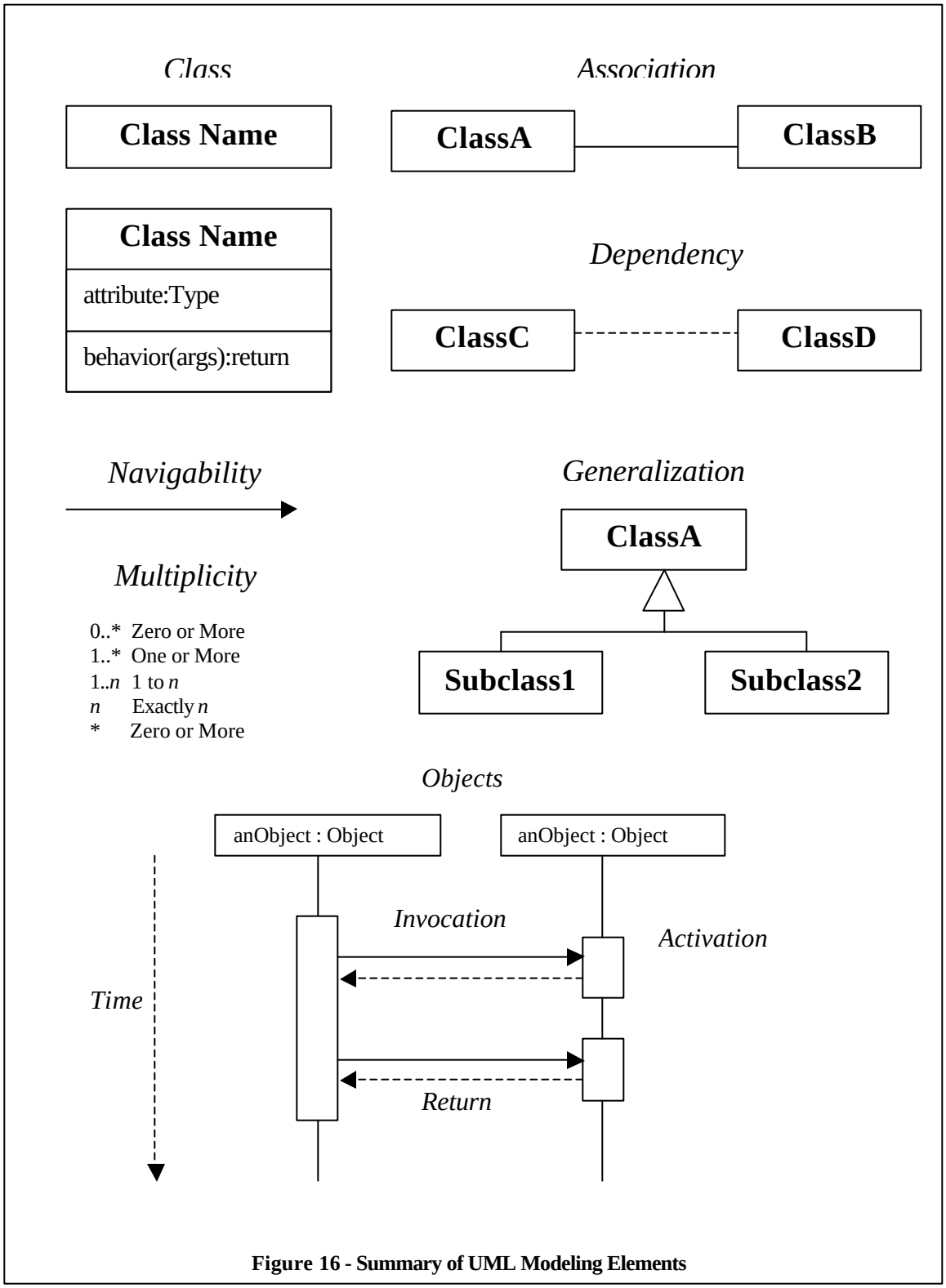

Figure 16 - Summary of UML Modeling Elements

References

Books

UML Distilled: A Brief Guide to the Standard Object Modeling Language, second edition, by Fowler and Scott, Addison-Wesley, @2000.

The Unified Modeling Language User Guide, by Booch, Rumbaugh, and Jacobson, Addison-Wesley, @1999.

The Unified Modeling Language Reference Manual, by Rumbaugh, Jacobson, and Booch, Addison-Wesley, @1999.

The Essence of Object-Oriented Programming with Java and UML, first edition, Wampler, Addison-Wesley, @2002.

Java Design: Objects, UML, and Process, first edition, by Knoernschild, Addison-Wesley, @2002.

UML Explained, first edition, by Scott, Addison-Wesley, @2001.

UML in a Nutshell, by Sinan Si Alhir, O'Reilly and Associates, @1998.

Web

Unified Modeling Language Resource Center, http://www.rational.com/uml

UML Resource Page, http://www.omg.org/uml

UML Home Page, http://www.uml.org

UML Design Center, http://www.sdmagazine.com/uml