OBJECT-ORIENTED SYSTEM ANALYSIS AND MODELING

CHAPTER ONE

Introduction & Review of principles of Object Orientation

What is Software?

Definition: Software is a set of programs to operate computers and related devices

The nature of software

■ Software is **largely intangible**. Unlike most other engineering artifacts, **you cannot feel the shape of a piece of software**, and **its design can be hard to visualize**. It is therefore **difficult for people to assess its quality or to appreciate the amount of work involved in its development**. This is one of the reasons why people consistently underestimate the amount of time it takes to develop a software system.

■ The mass-production of duplicate pieces of software is trivial. Most other types of engineers are very concerned about the cost of each item in terms of parts and labor to manufacture it. In other words, for tangible objects, the processes following completion of design tend to be the expensive ones. **Software**, on the other hand, **can be duplicated at very little cost by downloading over a network or creating a CD**. Almost all the **cost of software is therefore in its development, not its manufacturing**.

■ The **software industry is labor intensive**. It has become possible to automate many aspects of manufacturing and construction using machinery; therefore, other branches of engineering have been able to produce increasing amounts of product with less labor. However, **it would require truly 'intelligent' machines to fully automate software design or programming.** Attempts to make steps in this direction have so far met with little success.

■ **It is all too easy for an inadequately trained software developer to create a piece of software that is difficult to understand and modify**. A novice programmer can create a complex system that performs some useful function but is highly disorganized in terms of its design. In other areas of engineering, you can create a poor design too, but the flaws will normally be easier to detect since they will not be buried deep within thousands of pages of source code.

■ **Software is physically easy to modify**; however, **because of its complexity it is very difficult to make changes that are correct**. People tend to make changes without fully understanding the software. As a side effect of their modifications, new bugs appear.

Types of software and their differences

There are many different types of software. One of the most important distinctions is between ***custom*** **software,** ***generic*** **software and** ***embedded*** **software.**

Custom Software:

Custom software **is developed to meet the specific needs of a particular customer and tends to be of little use to others** (although in some cases developing custom software might reveal a problem shared by several similar organizations). Much custom software is developed in-house within the same organization that uses it; in other cases, the development is contracted out to consulting companies. Custom software is typically used by only a few people and **its success depends on meeting their needs.**

Examples of custom software include web sites, air-traffic control systems and software for managing the specialized finances of large organizations.

Generic Software:

Generic software, on the other hand**, is designed to be sold on the open market**, **to perform functions that many people need, and to run on general purpose computers**.
Requirements are determined largely by market research. There is a tendency in the business world to attempt to use generic software instead of custom software because it can be far cheaper and more reliable. **The main difficulty is that it might not fully meet the organization's specific needs.**

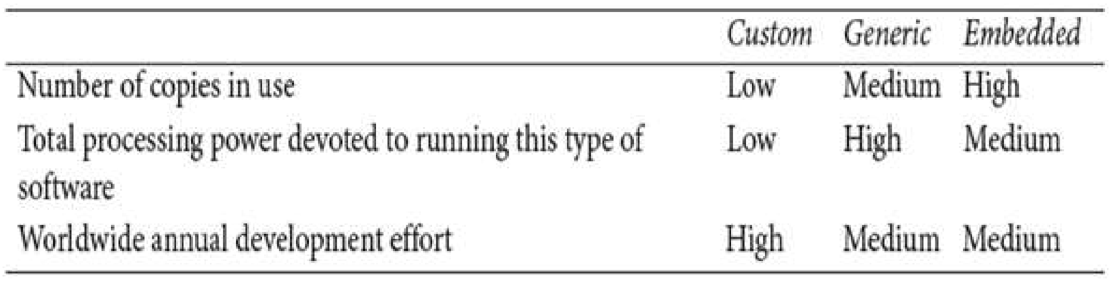
Examples of generic software include word processors, spreadsheets, compilers, web browsers, operating systems, computer games and accounting packages for small businesses.

Embedded Software:

Embedded software runs specific hardware devices which are typically sold on the open market. Such devices include washing machines, DVD players, microwave ovens and automobiles. Unlike generic software, users cannot usually replace embedded software or upgrade it without also replacing the hardware. The open-market nature of the hardware devices means that developing embedded software has similarities to developing generic software;

However, we place it in a different category due to the distinct processes used to develop it. Since embedded systems are finding their way into a vast number of consumer and commercial products, they now account for the bulk of software copies in existence.

Differences among custom, generic and embedded software



	Custom	*Generic*	*Embedded*
Number of copies in use	Low	Medium	High
Total processing power devoted to running this type of software	Low	High	Medium
Worldwide annual development effort	High	Medium	Medium

Note: Another important way to categorize software in general is whether it is *real-time* or *data processing* software.

Real-Time Software:

The most distinctive feature of real-time software is **that it has to react immediately (i.e. in real time) to stimuli from the environment** (e.g. the pushing of buttons by the user, or a signal from a sensor). Much design effort goes into ensuring that this responsiveness is always guaranteed. **Much real-time software is used to operate special-purpose hardware**; in fact almost **all embedded systems operate in real time**. Many aspects of the custom systems that run industrial plants and telephone networks are also real-time.

Generic applications, such as spreadsheets and computer games, have some real-time characteristics, since they must be responsive to their users' inputs. However, these tend to be *soft* real-time characteristics: when timing constraints are not met, such systems merely become sluggish to use. In contrast, most embedded systems have *hard* real-time constraints, and will fail completely if these are not met. Safety is thus a key concern in the design of such systems.

Data Processing Software:

Data processing software is used to run businesses. It performs functions such as recording sales, managing accounts, printing bills etc. The biggest design issues are how to organize the data and provide useful information to the users so they can perform their work effectively. Accuracy and security of the data are important concerns, as is the privacy of the information gathered about people. A key characteristic of traditional data processing tasks is that rather than processing data the moment it is available, it is instead gathered together in batches to be processed at a later time. Some software has both real-time and data processing aspects.

For example, a telephone system has to manage phone calls in real time, but billing for those calls is a data processing activity.

What is software engineering?

Definition: *software engineering* is the process of solving customer's problems by the systematic development and evolution of large, high-quality software systems within cost, time and other constraints.

Other definitions of software engineering

We have presented our definition of software engineering. Here are two other definitions:

■ IEEE: The application of a systematic, disciplined, quantifiable approach to the development, operation, maintenance of software.

■ The Canadian Standards Association: The systematic activities involved in the design, implementation and testing of software to optimize its production and support.

Software Engineering as a branch of the engineering profession

People have talked about software engineering since 1968 when the term was coined at a NATO conference. However, only since the mid-1990s has there been a shift towards recognizing software engineering as a distinct branch of the engineering profession. Some parts of the world, notably Europe and Australia, were somewhat ahead of others in this regard.

In most countries, in order to legally perform consulting or self-employed work where you call yourself an 'engineer', you must be licensed. Similarly, a company that sells engineering services may be required to employ licensed engineers who take formal responsibility for projects, ensuring they are conducted following accepted engineering practices.

Prior to the 1940s, very few jurisdictions required engineers to be licensed. However, various disasters caused by the failure of designs eventually convinced almost all governments to establish licensing requirements.

Licensing agencies have the responsibility to ensure that anyone who calls himself or herself an engineer has sufficient engineering education and experience. To exercise this responsibility, the agencies accredit educational institutions they believe are providing a proper engineering education, and scrutinize the background of those who are applying to be engineers, often requiring them to write exams.

We can characterize the work of engineers as follows:

Engineers *design* artifacts following well-accepted practices, which normally involve the application of science, mathematics and economics. Since engineering has become a licensed profession, adherence to *codes of ethics* and taking *personal responsibility* for work have also become essential characteristics. Some people only include in engineering those design activities that have a potential to impact public safety and well-being; however, since most people who are trained as engineers do not in fact work on such critical projects, most people define engineering in the broader sense.

Historically, engineering has evolved several specialties, most notably Civil, Mechanical, Electrical and Chemical Engineering. *Computer Engineering* evolved in the 1980s to focus on the design of computer systems that involve both hardware and software components. However, most of the practitioners performing what we have defined above to be software engineering have not historically been formally educated as engineers.

Many of the earliest programmers were mathematicians or physicists; then in the 1970s the discipline of *computer science* developed, and educated many of the current generation of software developers. The computer science community recognized the need for a disciplined approach to the creation of large software systems, and developed the software engineering discipline.

In the mid-1990s the first jurisdictions started to recognize software engineering as a distinct branch of engineering. For example, in the United Kingdom those who study software engineering in computer science departments at universities have been able to achieve the status of Chartered Engineer, after a standard period of work experience and passing certain exams. In North America, the State of Texas and the Province of Ontario were among the first jurisdictions to license software engineers (in 1998 and 1999 respectively).

In parallel with the process of licensing software engineers, universities have been establishing academic programs in universities that focus on software engineering, and are clearly distinct from either computer science or computer engineering. Since considerable numbers of these graduates are now entering the workforce, software engineering has become firmly established as a branch of engineering.

Stakeholders in software engineering

■ **Users**: These are the people who will use the software. Their goals usually include doing enjoyable or interesting work, and gaining recognition for the work they have done. Often they will welcome new or improved software, although some might fear it could jeopardize their jobs. Users appreciate software that is easy to learn and use, makes their life easier, helps them achieve more, or allows them to have fun.

■ **Customers** (also known as *clients*): These are the people who make the decisions about ordering and paying for the software. They may or may not be users – the users may work for them. Their goal is either to increase profits or simply to run their business more effectively. Customers appreciate software that helps their organization save or make money, typically by improving the productivity of the users and the organization as a whole. If you are developing custom software, then you know who your customers are; if you are developing generic software, then you often only have *potential* customers in mind.

■ **Software developers**: These are the people who develop and maintain the software, many of whom may be called software engineers. Within the development team there are often specialized roles, including requirements specialists, database specialists, technical writers, configuration management specialists, etc. Development team members normally desire rewarding careers, although some are more motivated by the challenge of solving difficult problems or by being a well-respected 'guru' in a certain area of expertise. Many developers are motivated by the recognition they receive by doing high quality work.

■ **Development managers**: These are the people who run the organization that is developing the software; they often have an educational background in business administration. Their goal is to please the customer or sell the most software, while spending the least money. It is important that they have considerable knowledge about how to manage software projects, but they may not be as intimately familiar with small details of the project as are some of the software developers. For this reason, it is important that software developers keep their managers informed of any problems. In some cases, two, three or even all four of these stakeholder roles may be held by the same person. In the simplest case, if you were privately developing software for your own use, then you would have all four roles.

Software quality

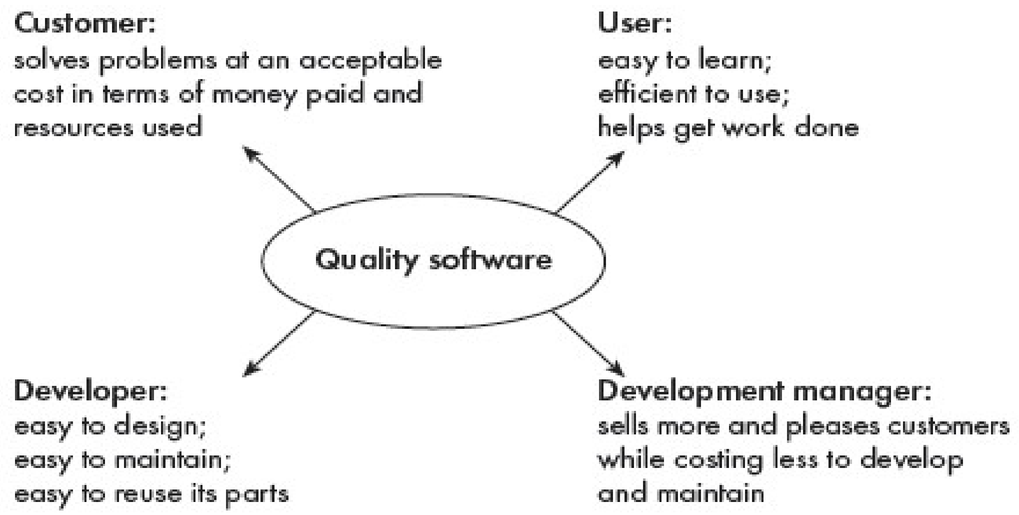
The following are five of the most important attributes of software quality.
.3
■ **Usability:** The higher the usability of software, the easier it is for users to work with it. There are several aspects of usability, including learn ability for novices, efficiency of use for experts, and handling of errors.

■ **Efficiency:** The more efficient software is, the less it uses of CPU-time, memory, disk space, network bandwidth and other resources. This is important to customers in order to reduce their costs of running the software, although with today's powerful computers, CPU-time, memory and disk usage are less of a concern than in years gone by.

■ **Reliability:** Software is more reliable if it has fewer failures. Since software engineers do not deliberately plan for their software to fail, reliability depends on the number and type of mistakes they make. Designers can improve reliability by ensuring the software is easy to implement and change, by testing it thoroughly, and also by ensuring that if failures occur, the system can handle them or can recover easily.

■ **Maintainability:** This is the ease with which you can change the software. The more difficult it is to make a change, the lower the maintainability. Software engineers can design highly maintainable software by anticipating future changes and adding flexibility. Software that is more maintainable can result in reduced costs for both developers and customers.

■ **Reusability:** A software component is reusable if it can be used in several different systems with little or no modification. High reusability can reduce the long-term costs faced by the development team. All of these attributes of quality are important. However, the relative importance of each will vary from stakeholder to stakeholder and from system to system.



What software quality means to different stakeholders

Often, software engineers improve one quality at the expense of another. In other words, they have to consider various *trade-offs*. The following are some examples of this:

■ Improving efficiency may make a design less easy to understand. This can reduce maintainability, which leads to defects that reduce reliability.

■ Achieving high reliability often entails repeatedly checking for errors and adding redundant computations; achieving high efficiency, in contrast, may require removing such checks and redundancy.

■ Improving usability may require adding extra code to provide feedback to the users, which might in turn reduce overall efficiency and maintainability.

Review of the principles of object orientation

What is object orientation?

Object-oriented systems make use of *abstraction* in order to help make software less complex. An abstraction is something that relieves you from having to deal with details. Object-oriented systems combine procedural abstraction with data abstraction. To help you better understand what this means, we will first take a look at these two types of abstraction.

What is Object-Oriented Software Engineering?

Object-Oriented Software Engineering is the process of solving customers' problems by using the **Object-Oriented concepts** and the systematic development and evolution of large, high-quality software systems within cost and time.

Procedural abstraction and the procedural paradigm

From the earliest days of programming, software has been organized around the notion of *procedures* (also in some contexts called *functions* or *routines*). These provide *procedural abstraction*. When using a certain procedure, a programmer does not need to worry about all the details of how it performs its computations; he or she only needs to know how to call it and what it computes. The programmer's view of the system is thus made simpler.

In the so-called procedural paradigm, the entire system is organized into a set of procedures. One 'main' procedure calls several other procedures, which in turn call others.

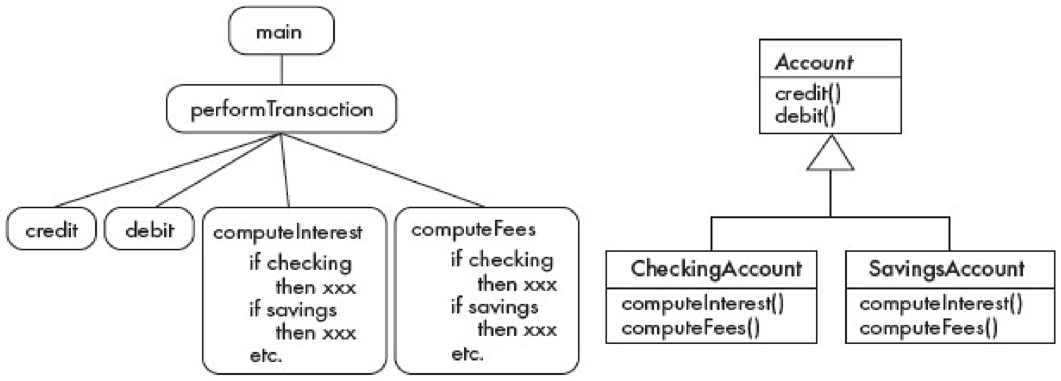
The *procedural paradigm* works very well when the main purpose of programs is to perform calculations with relatively simple data. However, as computers and applications have become more complex, so has the data. Systems written using the procedural paradigm are complex if each procedure works with many types of data, or if each type of data has many different procedures that access and modify it.

The object-oriented paradigm: organizing procedural abstractions in the context of data abstractions

Definition: The *object-oriented paradigm* is an approach to the solution of problems in which all computations are performed in the context of objects. The objects are instances of programming constructs, normally called classes, which are data abstractions and which contain procedural abstractions that operate on the objects.

In the object-oriented paradigm, a running program can be seen as a collection of objects collaborating to perform a given task. Figure 2.1 summarizes the essential difference between the object-oriented and procedural paradigms. In the procedural paradigm (shown on the left), the code is organized into procedures that each manipulate different types of data.

Figure 2.1



Organizing a system according to the procedural paradigm (left) or the object-oriented paradigm (right). The UML notation used in the right-hand diagram will be discussed in more detail later

<u>Concepts that define object orientation</u>

- Object
- Classes
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

<u>Classes and objects</u>

Classes and objects are the aspects of object orientation that people normally think about first. In this section, we will define in more detail what we mean by these two terms.

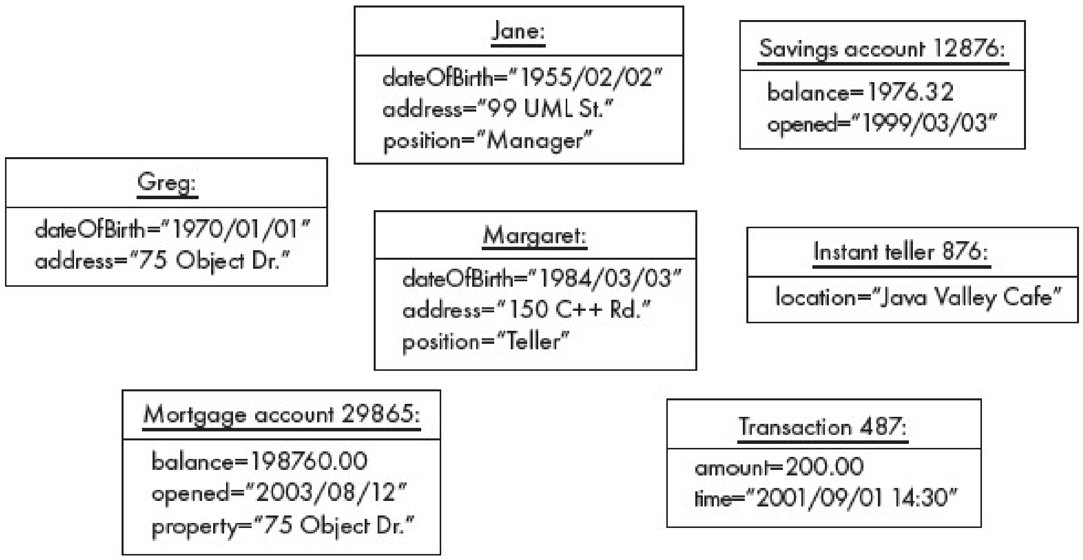
<u>Objects</u>

An object is a chunk of structured data in a running software system. It can represent anything with which you can associate *properties* and *behavior*. Properties characterize the object, describing its current *state*. Behavior is the way an object acts and reacts, possibly changing its state.

The below figure shows some of the objects and their properties that might be important to a banking system. The notation used in represent objects is UML.

<u>The following are some other examples of objects:</u>

■ In a payroll program, there would be objects representing each individual employee.
■ In a university registration program, there would be objects representing each student, each course and each faculty member.
■ In a factory automation system, there might be objects representing each assembly line, each robot, each item being manufactured, and each type of product.



Several objects in a banking application

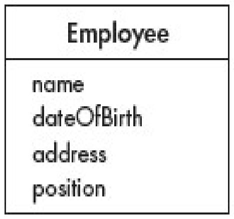
Figure 2.2

<u>Classes</u>

Classes are the units of data abstraction in an object-oriented program. More specifically, a class is a software module that represents and defines a set of similar objects, its *instances*. All the objects with the same properties and behavior are instances of one class.

For example, Figure 2.3 shows how the bank employees Jane and Margaret from Figure 2.2 can be represented as instances of a single class Employee. Class Employee declares that all its instances have a name, a dateOfBirth, an address and a position.

Figure 2.3



<u>Naming</u> A class, representing similar objects from above figure <u>Classes</u>

One of the first challenges in any object-oriented project is to name the classes. Notice that the class names mentioned in the last subsection such as Employee, Hospital and Doctor are *nouns*, have their first letter *capitalized* and are written in the *singular*.

If you want to give a class a name consisting of more than one word, then omit the spaces and capitalize the first letter of each word, For example: PartTimeEmployee.

It is also important to choose names for classes that are neither too general nor too specific. Many words in the English language have more than one meaning, or are used with a broad meaning.

For example, the word 'bus' could mean the physical vehicle, or a particular run along a particular route, as in, 'I will catch the 10:30 bus (but I don't care which vehicle is used)'. You might choose to call the class that represents physical vehicles BusVehicle and the class that represents runs along a route BusRouteRun. Sometimes it is possible to be too specific in naming a class: for example, when filling out a form, you may be asked to specify the 'city' as part of an address. But not everybody lives in a city! Therefore, rather than creating a class called City to store, for example, a person's place of birth, you should perhaps use the more general class name Municipality.

<u>Instance variables</u>

A variable is a place where you can put data. Each class declares a list of variables corresponding to data that will be present in each instance; such variables are called *instance variables*.

Attributes and associations
There are two groups of instance variables, those used to implement attributes, and those used to implement associations. An *attribute* is a simple piece of data used to represent the properties of an object.

For example, each instance of class Employee might have the following attributes:
■ name
■ dateOfBirth
■ socialSecurityNumber
■ telephoneNumber
■ address

An *association* represents the relationship between instances of one class and instances of another. For example, class Employee in a business application might have the following relationships:
■ supervisor (association to class Manager)
■ tasksToDo (association to class Task)

<u>Variables versus objects</u>

One common source of confusion when discussing object-oriented programs is the difference between *variables* and *objects*. These are quite distinct concepts. At any given instant, a variable can refer to a particular object or to no object at all. Variables that refer to objects are therefore often called *references*.

During the execution of a program, a given variable may refer to different objects. Furthermore, an object can be referred to by several different variables at the same time.

The *type* of a variable determines what classes of objects it may contain. Wewill explain the rules regarding this in later sections. Variables can be local variables in methods; these are created when a method runs and are destroyed when a method returns. However, objects temporarily referenced by such variables may last much longer than the lifetime of the method as long as some other variable also references the object.

Abstraction

Abstractions can help reduce some of a system's complexity. *Records* and *structures* were the first data abstractions to be introduced. The idea is to group together the pieces of data that describe some entity, so that programmers can manipulate that data as a unit.

However, even when using data abstraction, programmers still have to write complex code in many different places. Consider, for example, a banking system that is written using the procedural paradigm, but using records representing bank accounts. The software has to manage accounts of different types, such as checking, savings and mortgage accounts (a checking account would be called a cheque account or current account in some countries). Each type of account will have different rules for the computation of fees, interest, etc.

Encapsulation

A class acts as a container to hold its features (variables and methods) and defines an interface that allows only some of them to be seen from outside. Abstraction, modularity and encapsulation each help provide **information hiding**. This arises when software developers using some feature of a programming language or system do not need to know all the details; they only need to know sufficient details to use the feature. The result is that the developers have less confusing detail to understand and will therefore make fewer mistakes. Hence they can work effectively with larger systems.

Methods, operations and polymorphism

Methods

The word '*method*' is used in object-oriented programs where the words '*procedure*', '*function*' or '*routine*' might be used in other programs. Methods are procedural abstractions used to implement the behavior of a class.

Operations

An *operation* is a higher-level procedural abstraction. It is used to discuss and specify a type of behavior, independently of any code that implements that behavior. Several different classes can have methods with the same name that implement the abstract operation in ways suitable to each class. The word 'method' is used because in English it means 'way of performing an operation'. We call an operation *polymorphic*, if the running program decides, every time an operation is called, which of several identically named methods to invoke. The program makes its decision based on the class of the object in a particular variable. Polymorphism is one of the fundamental features of the object-oriented paradigm.

Polymorphism

Definition: *polymorphism* is a property of object-oriented software by which an abstract operation may be performed in different ways, typically in different classes.

As an illustration of polymorphism, imagine a banking application that has an abstract operation calculateInterest. In some types of account, interest is computed as a percentage of the *average* daily balance during a month. In other types of account, interest is computed as a percentage of the *minimum* daily balance during a month. In a mortgage account, to which you can only deposit (make a payment) but from which you cannot withdraw except initially, interest may be computed as a percentage of the balance at the *end* of the month.

In the banking system, the three classes CheckingAccount, SavingsAccount and MortgageAccount would each have their own method for the polymorphic operation calculateInterest. When a program is calculating the interest on a series of accounts, it will invoke the version of calculateInterest specific to the class of each account.

Organizing classes into inheritance hierarchies

If several classes have attributes, associations or operations in common, it is best to avoid duplication by creating a separate *super class* that contains these common aspects. Conversely, if you have a complex class, it may be good to divide its functionality among several specialized *subclasses*.

The relationship between a subclass and an immediate super class is called a *generalization*. The subclass is called a *specialization*. A hierarchy with one or more generalizations is called an *inheritance hierarchy*, a generalization hierarchy or an *is a hierarchy*. The reason for the latter name will become clear shortly.

You can draw inheritance hierarchies graphically as shown in Figure 2.4. The little triangle symbolizes one or more generalizations sharing the same super class, and points to the superclass. It is clearest when such diagrams are drawn with the super class at the top and the subclasses below, although other
arrangements are also allowed.

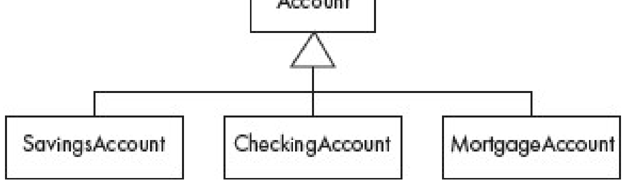


Figure 2.4 Basic inheritance hierarchy of bank accounts

It is also possible to show inheritance hierarchies textually using indentation, like this:

Account

SavingsAccount
CheckingAccount
MortgageAccount

<u>Definition:</u> *inheritance* is the *implicit* possession by a subclass of features defined in a superclass. Features include variables and methods.

You control inheritance by creating an inheritance hierarchy. Once you define which classes are superclasses and which classes are their subclasses, inheritance *automatically* occurs.

Organizing classes into inheritance hierarchies is a key skill in object-oriented design and programming. It is easy to make mistakes and create invalid generalizations. One of the most important rules to adhere to is the *isa* rule. The isa rule says that class A can only be a valid subclass of class B if it makes sense, in English, to say, 'an A *is a* B'.

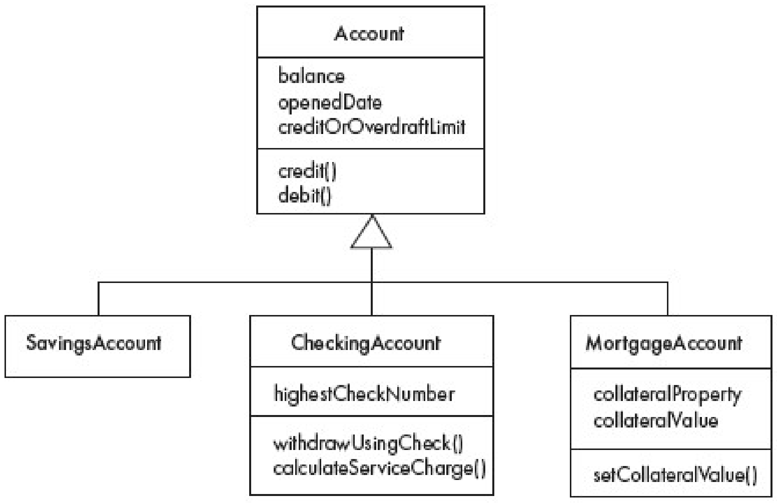


Figure 2.5 **Inheritance hierarchy of bank accounts showing some attributes and operations**

Example 2.2 Organize the following set of classes into hierarchies: Circle, Point, Rectangle, Matrix, Ellipse, Line, Plane.

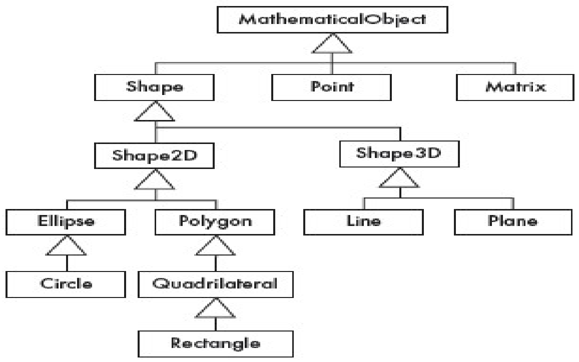


Figure 2.6 **A possible inheritance hierarchy of mathematical objects**

Figure 2.6 shows one possible solution – there can often be more than one acceptable answer to this kind of question.

<u>The effect of inheritance hierarchies on polymorphism and variable declarations</u>

Much of the power of the object-oriented paradigm comes from polymorphism and inheritance working together. In this section we will investigate this *synergy*. Figure 2.8 shows an expanded version of the hierarchy of two-dimensional shapes from Figure 2.6, also incorporating the EllipticalShape class from Figure 2.7, as well as a modified Polygon hierarchy. We will use Figure 2.8 to illustrate several important points; you should study it and try to understand it before proceeding. h) Telephone Phone line Digital line Phone call Conference call Call waiting Extension Feature Call on hold Caller Call forwarding Forwarded call Telephone number Voice mail message Voice mail Voice mail box

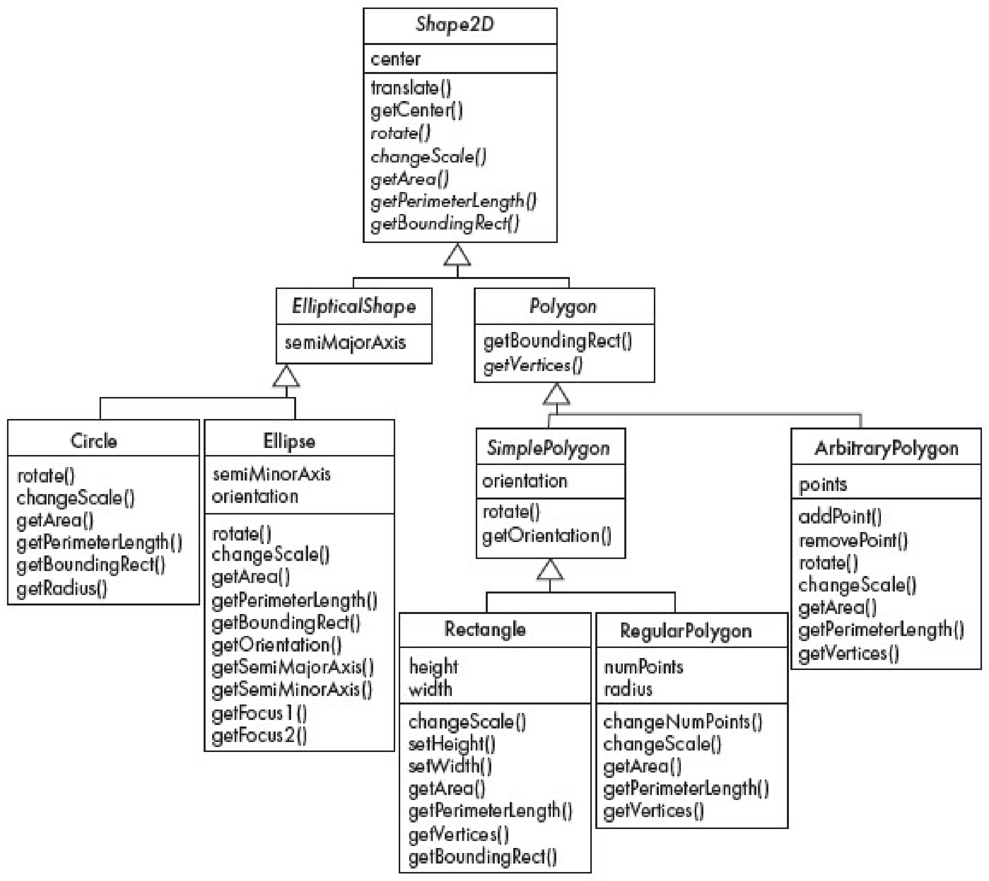


Figure 2.8 A hierarchy of shapes showing polymorphism and overriding

Figure 2.8 is a four-level hierarchy with four generalizations. The classes at the very bottom of the hierarchy are called *leaf classes*.