CHAPTER II

OBJECT ORIENTATION

1. Background

The object-oriented paradigm is often referred to as a new programming approach to software engineering, which was widely publicized and adopted in the late 1980s (Sommerville, 1996a, p. 248). “A programming paradigm is a way of conceptualizing what it means to perform computation and how tasks to be carried out on a computer should be structured and organized” (Budd, 1997, p. 7). Programming paradigms other from the object-oriented paradigm include, for instance, the imperative-programming paradigm (e.g., Pascal or C), the logic programming-paradigm (e.g., Prolog), and the functional-programming paradigm (e.g., FP) (p. 6). Budd argues that object-oriented programming is “a new way of thinking about what it means to compute, about how we can structure information inside a computer” (p. 2).

As Booch (1994a, p. 13-15) points out, the object-oriented paradigm is a more powerful approach in the decomposition of complex systems. Instead of an algorithm this type of decomposition uses objects. These systems can be described with an “is_a” as well as an “is_part_of” hierarchy, which makes the redundancy of the system under consideration more obvious. The first represents the class structure of a complex system, the latter the object structure, where each object in the object structure represents an instance of a class. Combined, both the class structure and the object structure represent the architecture of a system. Kay (1984) — often considered to be the father of object-oriented programming— argues that
the object-oriented paradigm allows to design parts of a computer system that have the same power as the whole.

The computer is divided (conceptually, by capitalizing on its powers of simulation) into a number of smaller computer, or objects, each of which can be given a role like that of an actor in a play. The move to object-oriented design represents a real change in point of view—a change of paradigm—that brings with it an enormous increase in expressive power. (p. 56)

Schach (1997, p. 18-22) states that the object-oriented paradigm makes software development easier and maintenance quicker because of the close correspondence between the objects in a software product and their counterparts in the real world. He explains that there is most often a sharp transition between phases (especially the analysis phase and the design phase), whereas the transition from phase to phase is far more smoother in the object-oriented paradigm, making it a more integrated approach because objects enter the software life cycle at the very beginning, thus reducing the number of faults during development. Finally, the fact that objects are independent entities, that can be utilized in future products, thus promoting reuse, which reduces the time and cost of both development and maintenance. Schach concludes that

a product that has been built using the structured paradigm is essentially a single unit. This is one reason why the structured paradigm has been less successful when applied to larger products. In contrast, when the object-oriented paradigm is correctly used, the resulting product consists of a number of smaller, essentially independent units. The object-oriented paradigm reduces the level of complexity of a software product and simplifies both development and maintenance. (p. 20-21)

2. The software crisis

The object-oriented paradigm is another attempt to address the so-called “software crisis.” This alleged crisis—later renamed the “software depression”—
was based on the fact that software programmers were lacking the necessary theoretical foundations and disciplines for their work. The industry felt the need to increase the discipline and precision of its work because computers were responsible for systems that could put human lives at risk (Ceruzzi, 1998, p. 105; Marciniak, 1994; Schach, 1997; p. 7; Sigfried, 1996, p. 3-6).

In order to address these increased failures in software development projects, the term “software engineering” was introduced, first mentioned at a NATO conference in 1967 (Naur, Randell & Buxton, 1976). Software engineering represents a discipline for the design and construction of fault-free software based on scientific principles of computer science and mathematics that is delivered on time, within budget, and satisfies the user’s needs (Boehm, 1976, p. 1226; Humphrey, 1993, p. 1218; Schach, 1997, p. 6, 17; Sommerville, 1996a, p. 4).

One of the major techniques that was introduced in the mid-1970s to help solve the software crisis was the so-called structured paradigm, which included a set of techniques, among others, structured systems analysis, composite / structured design, structured programming, and structured testing (see Schach, 1997, p. 17-18). Structured design is a process that is concerned with the specific formal design of the components of a system and the interrelationship between those components to help solve some well-specified problem (Yourdon & Constantine, 1978, p. 7). It is rooted in the idea of the Dutch computer scientist Edsger W. Dijkstra who wrote in the late 1960s that GO-TO statements should be eliminated from programming languages (e.g., Dijkstra, 1976; see also the discussion in Arnold, 1994, p. 4; King, 1984, p. 9; Schach, 1997, p. 375-378; Yourdon, 1975, p. 137-138). Thus,
Structured programming is primarily characterized by the elimination of the GO-TO statement, which was replaced by a number of other well-structured branching and control statements, and the concept of top-down program design, i.e. the process of identifying the major functions before proceeding to an identification of the lesser functions which derive from the major ones. The latter aspect is represented by two approaches, i.e. functional decomposition, where the resulting functional hierarchy would form the basic design structure, and data structured design, where logical structure of the data is the basis of the program or system. Structured programming was introduced to help reduce testing problems, increase productivity, and enhance clarity and readability of programs (King, 1984, p. 12, 17-39; Yourdon, 1975, p. 136-194).

However, as F. P. Brooks (1995, p. 179-203) points out, there are “no silver bullets” to solve this software crisis, i.e. no single software engineering development approach will produce an order-of-magnitude improvement in programming activity. The essential difficulties include the complexity of software systems, their conformation to other interfaces, their constant changeability, as well as their invisibility and unvisualizability (p. 181-186; for a summary, see Schach, 1997, p. 43-480). These are some of the issues that the structured paradigm was not able to overcome. Schach (p. 18-19) explains that this paradigm failed to deal with the increasing size of software products. And, it did not help to reduce the large amount
of the software budget that was devoted to maintenance. Schach summarizes the limited success of the structured paradigm in that the structured techniques are either action oriented or data oriented, but not both. The basic components of a software product are the actions of the product and the data on which those actions operate. (p. 18)

Schach (1997, p. 18-22) concludes that the object-oriented paradigm represents an alternative because it considers both data and action to be of equal importance. This corresponds with Brooks’ (1995) observation that the object-oriented programming paradigm is a possible cure of the software crisis. He states that “it is a very promising concept” (p. 220). The use of abstract data types and hierarchical types (“classes”) helps to remove a higher-order sort of accidental difficulty from the development process and allows a higher-order expression of design. However, the complexity of the design itself remains essential and can only partially be reduced using object-orientation (p. 189-190; see also Budd, 1997, p. 2): “Complexity is the business we are in, and complexity is what limits us” (Brooks, 1995, p. 226; italics in original).

3. Historical review

The origins of the object-oriented paradigm are attributed to the programming language SIM ULA (SIM Ulation LAnguage), which was developed in the early 1960s (see language history chart in Figure 5 on page 52). The developers of SIM ULA, Dahl & Nygaard (1966; 1967; Nygaard, 1986), introduced computational concepts that are considered to be the foundations for object-oriented programming languages that would subsequently be developed. In fact, the authors were the first to introduce the concepts of class, data encapsulation and
inheritance (see, for instance, Horowitz, 1983, p. 258-263, 409; Schach, 1997, p. 171-172). The fundamental concept of SIMULA is the process, which describes the organization and classification of actions that take place in a discrete event system and the data involved in the process. An activity is called a class; a process an object. A discrete event system may contain a collection of processes, or group of objects. Several processes with similar data structure and same behavior pattern belong to the same class, and are called activities. Finally, attributes describe the parameters and items of a data structure or process, which can be made accessible from the outside, i.e. from within other processes. Thus, a process is called a referenceable data structure. An individual reference is called an element.

Kay has been greatly influenced by SIMULA when he developed Smalltalk in the early 1970s, the first dynamically typed object-oriented programming language (see, for instance, Abadi & Cardelli, 1996, p. 11; Horowitz, 1983, p. 409), after he had developed FLEX in the late 1960s, the first personal computer to directly support a graphics- and simulation-oriented language (Kay, 1977, p. 232). He used the concepts of class and message from SIMULA. For instance, Smalltalk classifies objects into families that are generalizations of their properties. Also, messages are send to an activity, i.e. processes belonging to the same class as in SIMULA (p. 241). With regards to the concept of class and objects, Kay (1993/1996) states that “everything we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and
4. Object-oriented concepts and elements

Authors like Budd (1997, p. 7-13), Horowitz (1983, p. 409-411) or Kay (1996) present a summary of a set of characteristics that are fundamental to the object-oriented paradigm. First, everything is considered an object. Second,
computation is performed by objects communicating with each other via message passing, requesting that other objects perform actions. Third, each object has its own memory, which consists of other objects. Fourth, every object is an instance of a class. The role of the class stems from the ability to create objects with the view that their properties are the means for communicating with them. Fifth, the class is the repository for behavior associated with an object. And, sixth, classes are organized into a singly rooted tree structure, called the inheritance hierarchy. When objects have certain attributes in common with others, then, they fall into the same class. The attributes of classes, subclasses, and superclasses can be shared through the mechanism called inheritance. To conclude, A. Snyder (1993, p. 32) states that all objects embody an abstraction and can encapsulated; objects provide services that can be requested by clients which issue requests that identify operations as well as objects; the services of objects can be used for their classification; new objects can be created; and objects share a common implementation.

Booch (1994a, p. 27) states that principles in software engineering such as abstraction, encapsulation, modularity, hierarchy, etc. are not new. But, the object-oriented paradigm helps to bring them together in a synergetic way. Booch (p. 40-41) concludes that the conceptual framework for anything object-oriented is the object model—a conceptual representation of the organized complexity of software. It consists of four major elements, i.e. abstraction, encapsulation, modularity, and hierarchy. In addition, the object model contains three minor element, i.e. typing, concurrency, and persistence. The following sections describe the basic concepts and elements of the object-oriented model. The overview diagram shown in
Figure 6 on page 54, describes the relationship between the different categories (see also Forbrig, 2001, p. 18-19). The basic model represents the fundamental terms of the object-oriented paradigm. This model corresponds to the notion of data encapsulation and abstract data types. The static model describes the relationship between the different elements of the basic model. Certain elements of this model, such as association and inheritance, have been introduced from data modeling in software engineering. The behavior of the individual elements is captured in the dynamic model. The system usage is described using use cases.

![Figure 6: Model structure of the object-oriented paradigm](image)

4.1 Fundamental concepts

4.1.1 Objects

As mentioned above, according to the object-oriented paradigm a correspondence exists between a software simulation of a physical system and the
physical system itself. The object concept describes a tangible and/or visible thing that can be apprehended by the human mind and to which action can be directed (Booch, 1994a, p. 82). By analogy, the components of the software system are themselves called objects (Abadi & Cardelli, 1996, p. 7). A simplistic way of looking at an object is as a unified software component that incorporates both the data and the actions that operate on that data (Schach, 1997, p. 18). The author states that objects are simply the next step in the progression of data abstraction from procedures, to modules, to abstract data types, and to objects (p. 82-89, 161; see also Budd, 1997, p. 17; Korson & McGregor, 1990, p. 41), using a process called stepwise refinement (or rather enrichment) of high-level concepts to achieve lower level components by breaking up tasks into a number of subtasks, thus allowing the refinement in the description of tasks to take place at the same time as the refinement of the description of the data (see Schach, 1997, p. 82-89; Wirth, 1971, p. 221, 226; 1976, p. 49-52). The most basic software components are considered modules, i.e. the smallest pieces of a software product. A module is characterized by its action (behavior), its logic in performing the action, and the context where it is actually used (Schach, 1997, p. 144).

Thus, an object is an instantiation (or instance) of an abstract data type, or class, that supports inheritance. This means that “a product is designed in terms of abstract data types, and the variables (objects) of the product are then instantiations of the abstract data type (Schach, 1997, p. 171). Objects are the basic run-time entities in an object-oriented system. As Booch (1994a, p. 83-91) writes, objects have state, behavior, and identity, where objects with similar structure and behavior
are defined in a common class. The state of an object represents the cumulative results of its behavior. It consists of all its (usually static) properties and the current (usually dynamic) values of these properties. The behavior of an object shows how an object acts and reacts that is visible and testable to the outside. Behavior represents state changes and the passing of messages of an object. Both, state and behavior, describe a set of procedures and functions of an object, which define the meaningful operations on that object; they are associated with every object (see also Korson & McGregor, 1990, p. 42). An object’s identity is constituted by its property which distinguishes it from all other objects. To conclude with Sommerville (1996a), an object is defined as an entity that has a state and a defined set of operations which operate on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects (clients) which request these services when some computation is required. Objects are created according to some object class definition. (p. 250)

With Budd (1997, p. 48) and Sigfried (1996, p. 36-37, 58, 118, 148-151) we can speak of an object to have two “faces.” First, the interface containing the methods names is externally visible or available. This outside view is the collection of operations that define the behavior of the object. Second, on the other side of the interface—the implementation face—the individual data variables are located, which maintain the internal state of the object. In this “body” of the object the attributes and the actual implementation code of the method are located. Typically, interfaces should be few in numbers and small (i.e. low coupling), explicit,
consistent, and coherent; they should have a low semantic gap, they should be stable, and support information hiding.

Sigfried (1996, p. 155-157) introduces the notion of information objects and system objects as a way to better capture the idea of information and behavior that actually have been united in the object ideas. The first holds the information or attributes of a system; it represents a data carrier or collection of data needed and used together. The latter is the handler of these information objects and captures the behavior or methods of a system.

As has been indicated earlier, objects are a more abstract implementation of modules. They support maximal interaction within each module and minimal interaction between modules. This means that objects are characterized by high cohesion and low coupling. Cohesion refers to the degree of interaction within a module, i.e. how the actions performed by a module are functionally related. Coupling is the degree of interaction between two modules (Schach, 1997, p. 144-155, 177-178; Sigfried, 1996, p. 48).

4.1.2 Classes

A class is defined as a set of possible objects (Korson & McGregor, 1990, p. 42). It is intended to describe the structures of all objects that are generated from that class (Abadi & Cardelli, 1996, p. 11). Classes help to generate objects with common properties. Because objects are more primitive than classes, they should be understood and explained before classes (p. 49). In fact, classes provide a description of the attributes and methods that should be contained in an object. Thus, a “class is here simply an abstraction principle” (Sigfried, 1996, p. 75). It
represents “a set of objects that share a common structure and a common behavior” (Booch, 1994a, p.103); simply speaking, an object is just an instance of a class (p. 103). Sommerville (1996a, p. 250) defines an (object) class as a template for objects which includes declarations of all the attributes and services which are associated with an object of that class. Budd (1997) summarizes the notion of class in his second principle of object-oriented programming as follows:

All objects are instances of a class. The method invoked by an object in response to a message is determined by the class of the receiver. All objects of a given class use the same method in response to a similar message. (p. 10)

Similar to the two “faces” of an object—as described above—Booch (1994a, p. 105) speaks of an outside view and an inside view of a class. The former is considered the interface of a class, which focuses on the abstraction while hiding its structure and the specifics of its behavior (public, protected or private). The latter describes the implementation of a class that consists of all the specifics of its behavior, i.e. the implementation of all the operations described in that interface.

Budd (1997, 48-49) distinguishes various forms of classes based on their different responsibilities. First, classes can serve as data managers (data or state classes), which maintain data or state information. Second, classes can generate data on demand (for a data source) or accept data in order to process it when called upon (for a data sink). Third, it is helpful to separate the object being viewed (view class) from the view that displays the visual representation of that object (observer
class). And, fourth, facilitator, or helper, classes are classes that assist in the
execution of complex tasks, but maintain little or no state information themselves.

Finally, classes can also form metaclasses (Booch, 1994a, p. 133; Sigfried,
1996, p. 45-46). In this case, a metaclass would have instances which themselves
are classes. Classes with similar characteristics would be grouped together with a
common description of these characteristics. This is similar to the notion that a
class would be treated as an object that can be manipulated.

4.1.3 Attributes

As described earlier, an object is an instance of a class. The different
characteristics of an object are called attributes, i.e. the descriptor of an instance
(Sigfried, 1996, p. 30-34). Typically, a (simple) attribute has a name, a domain (or
data type), and a value. In order to group attributes that are related, they can be
organized into subgroups to form complex or composite attributes. Attributes can
describe an important and relevant property of an instance (i.e. descriptive
attribute). They can provide important state information connected with an
instance (i.e. state attribute). And, they describe the reference to other objects
needed to implement the relationship (i.e. referential attribute).

4.1.4 Methods

Budd (1997, p. 8-9, 73-74) calls the first principle of object-oriented
programming the means by which activities are initiated, i.e. message passing (also
named method lookup). A message passing expression consists of three parts, i.e.
the receiver, the message selector, and the argument. Budd explains that actions are
initiated by the transmission of a message to an agent or receiver (object), which is
responsible for the action. A message has a designated receiver for that message, i.e.
some agent to which the message is sent. Depending on the receiver, the message can be interpreted differently. This is called late binding between the message (function or procedure name) and the method (code fragment) to respond to the message as opposed to early (compile-time or link-time) binding of the name to code fragment in conventional procedure calls where a designated receiver does not exist.

Objects have behaviors which can be described as having responsibilities. The entire collection of responsibilities associated with an object is often called a protocol. A protocol determines the set of available messages that an object can receive, thus constituting the entire static and dynamic outside of an abstraction (Booch, 1994a, p. 43, 90-91; Budd, 1997, p. 9; Sigfried, 1996, p. 40). This allows to increase the level of abstraction and permits greater independence between agents.

A special case of binding is polymorphism, a concept from type theory (Booch, 1994a, p. 115; Sigfried, 1996, p. 55-57). In this case, a name may refer to instances of many different classes as long as they share the same superclass.

Polymorphic functions are functions whose operands (actual parameters) can have more than one type. Polymorphic types may be defined as types whose operations are applicable to operands of more than one type. (Cardelli & Wegner, 1985, p. 475)

Thus, an object referred to by a specific name is able to respond with the shared set of operations in different ways. A polymorphic reference can refer to instances of just one class (Korson & McGregor, 1990, p. 45). This is achieved
because of the fact that the receiving methods in the objects have the same name but
different implementations.

4.2 Major elements

4.2.1 Abstraction

A basic principle in software programming is conceptual modeling. It helps
to handle the complexity of a system. Modeling is a process to organize “knowledge
of an application domain into hierarchical rankings or orderings of abstraction, in
order to obtain a better understanding of the phenomena in concern” (Taivalsaari,
1996, p. 441; see also Sigfried, 1996, p. 7-12). This process is usually referred to as
abstraction, a technique that helps to reduce the complexity of a system. As applied
to software systems, data abstraction encapsulates concrete details of a data
structure in a more abstract framework. Abstraction focuses on the external view of
an object by defining conceptual boundaries to other objects. It helps to separate an
object’s behavior from its implementation (Booch, 1994a, p. 41)

Object-oriented programming provides direct support of the most common
abstraction principles, i.e. classification / instantiation, aggregation /
decomposition, generalization / specialization, and grouping / individualization
(Taivalsaari, 1996, p. 441-443). During classification, details of an instance are
suppressed and certain properties of a class as a whole are emphasized. The reverse
operation of classification is instantiation (or exemplification). Aggregation creates
part-whole hierarchies in which details of components are suppressed and details of
the relationship as a whole are emphasized. The reverse operation of aggregation is
decomposition. Generalization is an abstraction principle that suppresses the
differences between categories and emphasizes common properties. The opposite of
generalization is specialization. Finally, grouping (or association, partitioning, or cover aggregation) is used to describe the properties of objects as a whole by suppressing details of the group members and emphasizing the grouping of those objects together. The converse operation of grouping is individualization.

Abstraction, i.e. suppressing unnecessary details and accentuating relevant details, can be achieved through data encapsulation, i.e. a data structure together with the actions to be performed on that data structure. Thus, encapsulation is defined “as the gathering together into one unit of all aspects of the real-world entity modeled by that unit” (Schach, 1997, p. 162). Other types of abstraction include procedural abstraction and iterative abstraction. On a next level of abstraction are abstract data types, i.e. a data type together with the actions to be performed on instantiations of that data type. Abstract data types support both data abstraction and procedural abstraction (p. 166). These types of abstraction are instances of a more general design concept called information hiding, where the implementation details of the resulting design are hidden from other modules (p. 168-170; see also Sigfried, 1996, p. 46). The term was introduced by Parnas (1972a; 1972b) to describe a method of decomposing software systems called modularization which includes design decisions that need to be made before the actual work on independent modules can begin. According to this concept, sufficient information exists to use a module without the user having to know the method. An example of abstraction is data encapsulation, i.e. a data structure together with the actions to be performed on that data structure, which allows for conceptual independence of this data structure. Because all the actions that are
performed on the data of an object are included in that object, then, the object can be considered a conceptually independent entity, thus supporting encapsulation (Schach, 1997, p. 20; Sigfried, 1996, p. 46).

4.2.2 Inheritance hierarchy

In general, a hierarchy describes a ranking or ordering of abstraction. We have learned earlier that the class structure of an object-oriented system consists of an “is a” hierarchy, which describes single or multiple inheritance, and that the object hierarchy describes an “part of” or whole / part hierarchy, which denotes aggregation (Booch, 1994a, p. 59, 103). During aggregation elements are collected together to form a new idea, a case of abstraction. It also hides the details of its parts (information hiding). The resulting aggregate is considered “a collection of parts that together form something new that is more than just the sum of its parts” (Sigfried, 1996, p. 86), where the whole is described by an aggregate, and the parts by attributes (Booch, p. 103). Inheritance, then, is often regarded as the feature that distinguishes the object-oriented paradigm from other programming paradigms (see, for instance, Budd, 1997, p. 10, 143-145; Korson & McGregor, 1990, p. 42-45; Sigfried, 1996, p. 76-80; Sommerville, 1996a, p. 254; Taivalsaari, 1996, p. 447). As Booch (1994a) writes

inheritance is a relationship among classes wherein one class shares the structure and/or behavior defined in one (single inheritance) or more (multiple inheritance) other classes (p. 133; italics in original).

Inheritance describes the relation between classes, i.e. knowledge of a more general category is applied to a more specific category. This relationship allows for the definition and implementation of one class to be based on that of other existing
classes. Inheritance is considered a language mechanism that allows the definition of new objects based on existing ones. A new class inherits the properties of its parents, and may introduce new properties that extend, modify or defeat its inherited properties. Inheritance supports reuse and sharing of program code (or software components) across systems and allows for rapid changes to be made to an object, leading to consistency in interface. In addition, it facilitates the extensibility within a given system by classifying entities.

Inheritance describes the ability of sharing attributes and operations between objects and their class (Abadi & Cardelli, 1996, p. 15-16; Budd, 1997, p. 12; Sigfried, 1996, p. 79-82; Sommerville, 1996a, p. 253). As object classes are objects themselves, they also inherit their attributes from other classes (i.e. sub-class inherits from super-class). During single inheritance a class inherits attributes from another class. In contrast, multiple inheritance is obtained when a class inherits attributes from more than one class, i.e. one class with multiple superclasses. Class hierarchies can be illustrated with inheritance trees (hierarchical inheritance structure) that show how objects inherit attributes and services from their superclasses; an inheritance network is created to represent multiple inheritance. Taivalsaari (1996) concludes that the object-oriented inheritance mechanisms are essentially incremental modification mechanisms, i.e., mechanisms that allow existing programs to be extended and refined without editing existing code. In general, inheritance can be defined generally as an incremental modification mechanism in the presence of late-bound self-reference. (p. 474-475; italics in original)

Budd (1997, p. 129) mentions that inheritance can be described as both a form of expansion, where the behavior and data associated with the child class or
subclass are always an extension (i.e. larger set) of the parent class, and a contraction because a child class is always a more specialized (or restricted) form of the parent class. Inheritance is used in a variety of ways (see, for instance, Budd, 1997, p. 131-136; Meyer, 1996; Taivalsaari, 1996, p. 447-450). Meyer presents an inheritance taxonomy that consists of twelve forms of inheritance which are grouped in three broad categories, i.e. model inheritance, software inheritance, and variation inheritance. Model inheritance reflects “is-a” relations between abstractions and model. This category includes subtype inheritance, view inheritance, restriction inheritance, and extension inheritance. Software inheritance expresses relations within the software itself. This category comprises reification inheritance, structure inheritance, implementation inheritance, and facility inheritance (including the two common variants of constant inheritance and machine inheritance). Finally, variation inheritance describes a class by how it differs from another class. This category includes functional variation inheritance, type variation inheritance, and unaffecting inheritance. Budd and Taivalsaari organize the different major forms of inheritance according to their practical use in software systems, including cancellation, combination, construction, extension, generalization, inclusion, limitation, specialization, specification, and variance.

The inheritance hierarchy of an object-oriented system can also further be described with different types of associations (see Sigfried, 1996, p. 100-102), most commonly: is_a (simply called inheritance), consists_of (also referred to as aggregation), contains (looser type of aggregation), is_part_of, use, and knows. Finally, non-hierarchical associations can be described with relationships (see
Sigfried, p. 108-115). Generally, it is recommended to use attributes for the
description of these relationships. For instance, the multiplicity or cardinality of the
relationship are often denoted by numbers as one-to-one, one-to-many or many-to-
many. Also, relationships can be described as conditional (all objects participate) or
unconditional (not all objects participate).

4.2.3 Encapsulation

Booch (1994a) defines encapsulation as “the process of compartmentalizing
the elements of an abstraction that constitute its structure and behavior;
encapsulation serves to separate the contractual interface of an abstraction and its
implementation” (p. 50; italics in original). It is a concept that is similar to
information hiding in that a module’s interface only reveals as little as possible
information about the inner structure of the module (Sigfried, 1996, p. 46-47).
Thus, when something changes inside the object, this change does not necessarily
has to be reflected in its interface.

Object-oriented programming and the principles of encapsulation are
defensive techniques whose sole purpose is to immunize the program from
effects of scar tissue. In essence, object-orientation divides the 1000-brick
tower into ten 100-brick towers. (A. Cooper, 1999, p. 55)

Data encapsulation is an example of abstraction. It helps to simplify product
maintenance by identifying aspects of a product that are likely to change and
designing it in order to minimize the effects of future changes (Schach, 1997, p.
161-166).

4.2.4 Modularity

Modularity refers to the property of a system that has been decomposed into
a set of discrete components or modules (Booch, 1994a, p. 57; Sigfried, 1996, p.
In the context of the object-oriented paradigm, an object represents a module as well as a method in an object. The literature (e.g., Schach, 1997, p. 144) distinguishes between the action of a module (i.e. its behavior), the logic of a module (i.e. operation or method), and the context of a module (i.e. its specific usage).

Based on this concept, a module (or object) is characterized by a set of important properties (see Sigfried, 1996, p. 50-52). Objects, when used as a modeling tool, should be close to the mental model of our thinking (i.e. understandability). Objects need to support decomposability (i.e. division into subobjects) as well composability (i.e. the combination of objects to form larger objects). When changes are made to the specifications, this should result only in limited changes of the whole system (i.e. continuity). Protection will prevent objects from being affected by some abnormal condition that occurs during run time of a system. Finally, objects are modeled close to reality, thus they satisfy linguistic modular units property.

4.3 Minor elements

4.3.1 Typing

In contrast to the class concept which defines objects, the type concept is a means to characterize the values an attribute can take in terms of restricting or allowing certain values or types (Danforth & Tomlinson, 1988, p. 32; Sigfried, 1996, p. 35, 38). According to Booch (1994a) typing is defined as “the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted arrays” (p. 66; italics in original). A type definition helps to prevent an untyped
representation from arbitrary or unintended use. “It provides a protective covering
that hides the underlying representation and constrains the way objects may
interact with other objects” (Cardelli & Wegner, 1985, p. 474). As these authors
write, types are generally associated with constants, operators, variables, and
function symbols. The literature (e.g., Budd, 1997, p. 179; Cardelli & Wegner, p.
474-475) make a further distinction between strong and weak typing as well as
static and dynamic typing. Strongly typed languages are those in which all
expressions are type consistent. Static typing requires type detection at compile
time, i.e. a type assigned to a variable using a declaration statement. For dynamic
typing, a type-checking mechanism is introduced at run-time. Budd points out the
polymorphism is characterized by the fact that the dynamic type does not have to
match the static type. Cardelli & Wegner recommend to use strong typing and
adopt static typing whenever possible.

4.3.2 Concurrency

Objects can be characterized as asynchronous (sequential), guarded, or
269-270). Concurrency, then, describes whether an object continues processing
after it has sent or received a message. This concept helps to distinguish an active
object from one that is not active. The state of an active object can be changed by
internal operations executing within the object itself, whereas passive objects are
only realized as a parallel process with entry points corresponding to the specific object operations.

4.3.3 Persistence

The idea of persistence describes the fact whether an object’s existence will transcend over time and/or space (Booch, 1994a, p. 77; Sigfried, 1996, p. 43). Thus, persistence describes whether an object will exist only during the single execution of a system or between individual executions and/or whether the location of an object has moved away from an address space in memory in which it was initially created.

5. The object-oriented development process

5.1 Basic characteristics

The overall process in the object-oriented paradigm is called object-oriented software engineering or object-oriented development (see, for instance, Booch, 1994a, p. 24, 172; Pressman, 1997, part 4; Schach, 1997, p. 268-290, 346-353; Sigfried, 1996, p. 131-137; Sommerville, 1996a, p. 249). This means that an object-oriented approach is used throughout the entire development process. The basic premise of this approach is to decompose a complex system in the real world in order to find a close mapping (see also Sigfried, p. 14-18) between the reality (the model) and the implementation (or software solution) that is comprised of a logical model (class structure and object structure) and a physical model (consisting of a module architecture and a process architecture), where both have static an dynamic entities. This notion is similar to what is discussed in psychology under the heading of objects and object relationships, which are important in how we perceive things in the real world and how this perception affects our mental model (see, for instance, the overview article by Compton, 1995). To conclude with Booch (1994a):
“The fundamental task of the software development team is to engineer the illusion of simplicity — to shield users from this vast and often arbitrary external complexity” (p. 6).

In general, this process consists of the following phases (see, for instance, Booch, 1994a): object-oriented analysis (OOA), object-oriented design (OOD), object-oriented programming (OOP), and object-oriented testing. First, object-oriented analysis refers to the development of an object-oriented model of the application domain. This involves the identification of the classes and objects of the logical model, which form the vocabulary of the problem domain. Second, object-oriented design is concerned with the development of an object-oriented model of a software system to implement the identified requirements. At the end of this stage towards the beginning of the implementation phase, the physical representations of the logical model are determined. Third, object-oriented programming represents the realization of a software design by means of an object-oriented programming language or O O PL (e.g., Ada, C++, Java, or Smalltalk). And, fourth, object-oriented testing applies specific technical metrics to assess the functionality of an object-oriented system. To conclude with Korson & McGregor (1990), the object-oriented paradigm takes a modeling point of view. The analysis and design phases of the traditional life cycle, while remaining distinctly separate activities in the object-oriented life cycle, work together closely to develop a model of the problem domain. The model is constructed by viewing the problem domain as a set of interacting entities. The software-based models of entities and the relationships between them are assembled to form the basic architecture of the application. (p. 46)
The overall development process of a software system includes a set of different stages, which begin with an informal design outline that is developed towards a finished design by adding formality and detail (Sigfried, 1996, p. 132; Sommerville, 1996a, p. 210-211). As Sommerville explains, the activities during the different stages include architectural design, abstract specification, interface design, component design, data structure design, and algorithm design. One strategy to achieve this is object-oriented design, which is based on the assumption that the system consists of a collection of objects (Sommerville, 1996a, p. 215)—object-oriented design in this context refers to the overall object-oriented development process or object-oriented modeling. As objects are related to things in the real world or to new ideas in an analysis model, it is possible to create a one-to-one mapping between objects in the real world and objects in the model (Sigfried, 1996, p. 134).

Booch (1994a, p. 234-264; see summary in White, 1994) suggests a three-step approach consisting of requirements analysis, domain analysis, and system design. During the requirements analysis the goal is to identify the objects and classes and their semantics (i.e. attributes and behaviors). Then, their relationships (e.g., associations, collaborations) are determined, which are implemented in the final stage. This micro development process is completed with the macro development process. This process begins by establishing the core requirements for the software product (i.e. conceptualization). Then, a model of the system’s desired behavior is developed (i.e. analysis). Next, the architecture for the implementation is created (i.e. design). Then, the implementation is refined through a series of
successive iterations (i.e. evolution). Finally, the product is managed after the final delivery (i.e. maintenance).

5.2 Life-cycle models

Because for the need of iterations between phases in the object-oriented development process, a life-cycle model is needed that supports these stepwise refinements for incremental development (e.g., Schach, 1997, p. 72, 280-282). According to him, it is important to determine a life-cycle model before starting with a project (p. 54). With regards to instructional systems design, Banathy (1987) also proposes an iterative and spiralic design model that allows for greater participation of the user during the design process and for better addressing the dynamic interaction of different conceptual spaces during the design inquiry (Banathy, 1987, p. 94). He (p. 92) states that Jones was among the first in the early 1960s to replace the phase-by-phase and step-by-step “linear” design approach with an iterative process where the instructional design would have to cycle through the various phases, compatible with the recurring exploration of the knowledge, problem, and experience spaces in Banathy’s model. The iterative design process allows for various feedback and feedforward loops in order to continuously shape emerging prototypes. Banathy stresses the fact that participative design allows to bring together various stakeholders in the process. Banathy describes this activity as “a process of negotiation among those with different points of view and value systems in order to find a satisfying solution” (p. 93).

In general, a life cycle model in software engineering describes a series of steps through which the product progresses. A life cycle model generally includes
the following stages or phases (e.g., Schach, 1997, p. 30-43; Sommerville, 1996b): Analysis (i.e. system, software, and user requirements and specifications), planning (i.e. project management plan), design (i.e. architectural design, detailed design), implementation (i.e. coding or programming), integration (i.e. top-down or bottom-up integration), maintenance (i.e. correction, perfection, or adaption), and retirement.

The most basic, but very unreliable approach is the build-and-fix model (e.g., Boehm, 1988, p. 61-63; Schach, p. 53-54). According to this model a product is fully built and reworked as many times as necessary. Thus, the development would immediately begin with coding or programming before decisions about requirements, design, testing, and maintenance would be made.

The most common life-cycle model is the waterfall model (e.g., Boehm, 1988, p. 63; Schach, 1997, p. 54-59). According to this model, the development process proceeds from one stage or phase to the next, whereby testing is inherent in every phase. The goal is to deliver a complete, operational product.

The rapid prototyping model (e.g., Beyer & Holtzblatt, 1998, p. 367-411; Constantine & Lockwood, 1999, p. 211-224; Nielsen, 1993b, p. 93-99; Rettig, 1994; Schach, 1997, p. 59-61, 199-211) is a functional working model of a subset of the product. In an educational context, Gentry (1994, p. 160) defines a prototype as “functional version of an instructional unit, usually in an unfinished state, whose effectiveness and efficiency can be tested.” In the educational sector, rapid prototyping is typically recommended as a method for general instructional design (e.g., T. S. Jones & Richey, 2000; Tripp & Bichelmeyer, 1990), educational
software development (e.g., Henson & Knezek, 1991; Moonen, 1994; 1996; Nieveen, 1999), for the design of educational hypermedia (e.g., Moonen, 1999; 2000; Porras & Giodano, 1995) as well as Web-based learning environments (e.g., Boling & Frick, 1997). Typically, rapid prototyping follows the same procedure as in the waterfall model, however only using an early scaled-down version of a system which will later be integrated into a full-scale system. Rapid prototyping supports the improvement of parts of a system in successive increments with continuous (usability) testing. Prototypes can follow different orientations or formats, i.e. action, fidelity, and orientation. Prototypes can be active or passive. A passive prototype is usually an inactive prototype made with pencil and paper. An active prototype is an actual implementation model with some functionality. Prototypes can resemble the user interface at different levels of fidelity. A high-fidelity prototype closely resembles the final product, whereas a low-fidelity prototype (a.k.a. paper prototype) has only a vague resemblance (for a discussion of the use of paper mock-ups in computer systems design, see Ehn & Kyng, 1991). Finally, prototypes can be described with a horizontal or vertical orientation. A horizontal prototype shows a superficial representation of most or all the user interface but no depth, whereas a vertical prototype represents a narrow slice through the system, but no width. A hybrid model using both orientation to some degree is called a deep-and-wide prototype, which is a shallow realization of the entire user interface with a simulation or implementation of selected use cases.

In the incremental model (Schach, 1997, p. 61-66; Sommerville, 1996b, p. 269) the system’s functionality is portioned into a series of increments, which are
developed and delivered one after the other. In contrast to the waterfall model and rapid prototyping, the incremental models delivers an operational product at each stage. However, these increments only satisfy a subset of the client’s requirements, which allows the client to adjust to the new product over time. An extension of the incremental model is the cleanroom approach (Mills, Dyer & Linger, 1987). This method uses a limited set of design increments in order to describe the logic of a software product. It also releases operational products throughout the entire development process instead of after its completion.

The spiral model (Boehm, 1988) was introduced to minimize the risk during the development process. At each cycle the objectives of the portion of the product, alternative means for its implementation, and constraints on the application of the alternatives are identified. Next, the alternatives are evaluated relative to the objectives and constraints. Then, the remaining risks are evaluated. Finally, each cycle is evaluated in order to move to the next phase, leading to the development of various subsets or prototypes of the final product. Because of its risk consideration, the spiral model can be combined with any other software development approach.

As indicated earlier with regards to the object-oriented paradigm, the life-cycle model needs to be iterative in order to provide the necessary feedback during the development process. Thus, incremental models seem to be most suitable as it supports continuous refinement. One such approach for object-oriented paradigm is the fountain model (Henderson-Sellers & Edwards, 1990, p. 152; see also the discussion in Schach, 1997, p. 280, 283). According to this model, the development process evolves along a central vertical line on which the various phases overlap,
which specifically reflects an overlap of activities; each phase is iterated (or refined).

The authors (p. 149-151) suggest a seven-step methodological framework for object-oriented systems development. The first step, which always begins the analysis phase, an object-oriented requirements specification (OORS) is undertaken. Then, the objects are identified along with their attributes and services. Third, the interactions between the objects are established. Next, the analysis phases merges into the design stage, where more internal details of the objects are defined. Fifth, objects are constructed from libraries of more primitive objects, which themselves may contain object classes from previous applications. Sixth, hierarchical inheritance relationships are introduced as needed through iterative analysis of whether new superclasses (parents) or new subclasses (children) would be useful. Finally, an aggregation and/or generalization of classes is undertaken.

5.3 Phases of the object-oriented life cycle

The major steps in object-oriented software development can be summarized as follows (see Booch, 1987a, p. 17-18; 1987b, p. 48-50; 1994b, p. 38-41). First, the objects and their attributes are identified in order to recognize the major actors and their role in the problem space. Then, the operations performed on or by an object are identified, which leads to the characterization of the behavior of each object or class of objects. Third, the visibility of each object is established in order to identify the dependencies among objects and classes of objects, which results in an topology of the model of the problem space. Fourth, the interface of each object is established by producing a module specification of each object. Finally, each
object as well as the interface from step 4 are implemented by choosing a suitable representation for each object or class of objects.

5.3.1 Object-oriented analysis

The purpose of object-oriented analysis (OOA) is to define a system that is supposed to be built (Sigfried, 1996, p. 132-133). Booch (1994a) defines OOA as “a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain” (p. 39; italics in original). The resulting analysis model is developed from a logical or essential model, which is free from technological conditions and/or limitations, towards an physical or implementation model, which incorporates the different physical implementation units.

According to Schach (1997, p. 199, 271-280), OOA consists of three steps. First, during class modeling the classes and their attributes are identified, which results in a class model. Class modeling is based on a concise problem definition, which results in an informal strategy to solve the problem and then in a formal strategy that identifies the nouns of the informal strategy as candidate classes.

Second, dynamic modeling determines the actions which need to be performed by or to each class or subclass, which results in a dynamic model. This is best achieved by listing typical scenarios (or use cases) of the interactions that may take place between users and the system. Finally, functional modeling describes how the results are computed by the product. The result is a functional model.

During OOA, a series of different techniques can be used to identify objects (see summary in Sommerville, 1996a, p. 256). One of them is based on a
grammatical analysis of a natural language description of a system, in which objects are described as nouns, attributes as adjectives or modifiers of nouns or verbs, and operations, services or methods as verbs. This approach was first introduced in 1980 by Abbott (1983) and then further incorporated into the object-oriented software development process by Booch (e.g., 1987a, p. 17-18; 1987b, p. 48-50; 1994b, p. 38-41). This analysis process results in a mapping of nouns into objects, verbs into methods, and adjectives and adverbs into attributes (see also Sigfried, 1996, p. 54-55). In the application domain, tangible entities or things can be used to denote roles, events, interactions, organizational units, etc. Using a behavioral approach, first the overall behavior of the system is described before the various behaviors are assigned to the different parts of the system and the different players are identified of who initiates and participates in these behaviors, i.e. objects. Finally, a scenario-based analysis (or use cases) can be applied where various scenarios of system use are identified and analyzed to specify the required objects, attributes and operations.

During scenario-based analysis, every activity that must take place is identified and assigned to some component as a responsibility. A method of analysis is to record the components on so-called CRC (Component, Responsibility, Collaborator) cards, a method that was developed by Cunningham (Beck & Cunningham, 1989; see also Booch, 1994a, p. 160; Budd, 1997, 32; Forbrig, 2001, p. 129-131; Nielsen, 1993b, p. 99-101; Sommerville, 1996a, p. 256). As illustrated in Figure 7 on page 79, the face of a CRC card contains the name of the software component, the responsibilities of the component, and the names of other
components with which the component must interact (i.e. collaborators). On the back of the card the class is defined with additional comments for logical (essential) model or physical (implementation) model.

<table>
<thead>
<tr>
<th>Class Name: ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superclasses: ...</td>
</tr>
<tr>
<td>Subclasses: ...</td>
</tr>
<tr>
<td>Description of responsibilities:</td>
</tr>
<tr>
<td>Methods...</td>
</tr>
</tbody>
</table>

Figure 7: A Class-Responsibility-Collaborator (CRC) card

5.3.2 Object-oriented design

The goal of object-oriented design (OOD) is to design a product in terms of objects (Schach, 1997, p. 346). According to Booch (1994a), object-oriented design is defined as a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design. (p. 39; italics in original)

Typically, during OOD new objects are added to existing objects that have been identified during OOA, or to partition objects into several layers of subobjects through aggregation (Sigfried, 1996, p. 134). Partitioning allows a system to be divided into smaller isolated parts in order to reduce its complexity. In this case, “an object is an isolated world to which we have delegated some responsibilities”
Using aggregation, the software developer can collect together objects that belong together that, once collected, form a new idea, something whole that can easily be referred to. These collections form aggregates which have characteristics that belong to the whole rather than to the parts. Generally, an association of “is_part_of” is used to describe an aggregate (p. 84-87, 95-96).

Usually, OOD consists of three steps (Schach, 1997, p. 347). First, the actions (or methods) of the classes are determined. Then, the product is designed in terms of clients (i.e. a module that makes use of an object) of objects. Finally, detailed design is conducted during which each component is developed in detail (p. 350-352).

5.3.3 Object-oriented programming

As indicated earlier (e.g., Booch, 1994a, p. 38-39), an object-oriented program must use objects as its fundamental building blocks, where each object is an instance of a class, and classes are related to one another via inheritance relationships. An object-oriented system, then, is developed using an object-oriented programming language (see the history chart of object-oriented languages in Figure 5 on page 52). The literature distinguishes between object-based languages, which support data abstraction and classes (e.g., Ada), and object-oriented languages, which are object-based and also provide support for inheritance and polymorphism (e.g., C++, Java, Smalltalk). Generally, the choice of a programming
language is a matter of cost-benefit analysis (see for instance Mantai & Teorey, 1988; Schach, 1997, p. 89-90, 369).

Schach (1997, p. 378-385) points out that it is important to establish coding standards that make maintenance easier. He suggests a set of good language-independent programming practices that would help to implement these standards. First, consistent and meaningful variable names should be used. Then, self-documenting code should be used, i.e. variable names and their code are crafted in such a way that no additional comments are needed. However, each variable name should be explained at the beginning of a module (i.e. in the prologue comments). Next, parameters should be used for any apparent constants, i.e. variables whose values will never change. Then, the code should be laid out in such a way that it increases readability, e.g., no more statement on one line, the use of indentation, or the use of blank lines to separate methods. Finally, nested if statements should be avoided because they would make the code too complex. A concrete example of how coding standards can be implemented is provided, for instance, by Vermeulen et al. (2000) for the Java programming language.

Also, the issue of portability needs to be addressed (Schach, 1997, p. 396-404). A software product is considered portable when it can be adapted to a new computer rather than writing a new product from scratch. Portability needs to
consider incompatibilities of hardware configurations, operating systems, and compilers.

5.3.4 Object-oriented testing

One advantage of the object-oriented paradigm is that it reduces the need for testing (e.g., Binder, 1994; 2000; McGregor & Sykes, 2001; Schach, 1997, p. 420-423). For instance, the inheritance hierarchy ensures that when one class has been tested no further retesting is required. This applies to new methods that are defined within a subclass of such a tested class. Because a class is an abstract data type, it has no concrete realization. Therefore, it can only be tested during non-execution as opposed to execution-based testing.

Booch (1994a, p. 136-138) provides a list of metrics that can be used for assessing the quality of an object. According to this list, testing would, then, look at coupling (i.e. strength of the association), cohesion (i.e. the degree of connectivity), sufficiency (i.e. level of characteristics of the abstraction captured by class or module), completeness (i.e. level of characteristics captured in the interface of the class or module), and primitiveness (i.e. efficient implementation of primitive operations) of an object.

Typically, the development cycle would be completed with product testing and acceptance testing (Schach, 1997, p. 448-450). The former looks at the correctness, robustness, performance, and documentation of the product. The latter provides the client with the opportunity to determine whether the product does satisfy the initial specifications. Acceptance testing follows the same procedures as during product testing, however, now performed on actual data. According to
testing can be achieved by two phases either unit testing (i.e. to test a single unit) and integration testing (i.e. to test how objects work together) or static testing (e.g., code inspection and walk-throughs) and dynamic testing (i.e. execution of code using test cases, etc.).

6. The Unified Modeling Language (UML)

6.1 Overview

The Unified Modeling Language (UML) is a modeling notation to specify, visualize, construct, and document software. UML represents a standard for the modeling of object-oriented specifications—and a promising enhancement for CBI authoring tools by helping to integrate suites of reusable program modules in component-based systems using learning objects (Douglas, 2001, p. 3-4; Gibbons & Fairweather, 2000, p. 436).

As shown in Figure 8 on page 84, the beginning of UML dates back to 1994, when Booch and Rumbaugh—both at the Rational Software Corporation at that time (see the Web site at www.rational.com)—began to combine their object-oriented development methods into a single approach called the Unified Method (see overview in Booch, 1999; Forbrig, 2001, p. 41-44; Kobryn, 1999; 2002; Osterreicher, 1997/1999, p. 6-7; OMG unified modeling language specification, 2001, p. 1-11-1-14; R. Smith, 2000; Williams, 1997). This effort was initiated in response the growing number of alternative object-oriented methods in the late 1980s in order to find a single standardized modeling language that would meet all these different approaches. In 1991, Booch (1994a) had first published his “Booch method” for object-oriented analysis and design including diagrams for the
In 1991, Rumbaugh published his object modeling technique (OMT), which focuses on the partitioning of a model among real-world objects. Later in 1995, Jacobson (2000, p. 100, 287, 330-331) joined Booch and Rumbaugh and integrated his object-oriented software engineering (OOSE) model, which represents a system in terms of its use cases based on his development process called Objectory, which was first released in 1987. Objectory became the recommended software development process for using UML at the Rational Software Corporation and was renamed the Unified Software Development Process (or short the Unified Process). These three...
authors are considered the primary developers of UML, who are often referred to as the “amigos”). Another important concept that was fully integrated into UML are Harel’s (1987; Harel & Gery, 1997) statecharts. This concept is an extension of state transition diagrams which represent graphs with nodes denoting states and arrows denoting transitions. A statechart describes the behavior of a system. When attached to a class, it describes all the behavioral aspects of the objects in that class. In addition, the UML developers also integrated ideas and concepts from various other companies, such as Microsoft Corporation, Hewlett-Packard Company, Oracle Corporation, IBM Corporation, etc.

After the first draft versions of UML were developed from 1995-1996, UML version 1.0 was finally issued in 1997 by the Rational Software Corporation with input from different industry groups, who joined the UML Consortium, and other representatives from the industry. Finally, UML version 1.1 was accepted in late 1997 as an industry standard by the Object Management Group (OMG)—a not-for-profit consortium that develops computer industry specifications for interoperable enterprise applications (see the OMG Web site at www.omg.org). Since then, OMG is now responsible for the further development and maintenance of UML through its Revision Task Force (RTF). In 2000, UML version 1.3 was submitted for standardization to the International Organization for Standardization (ISO). At the time of this writing, the current version of UML is version 1.4, which was formally released in September 2001 (OMG unified modeling language specification, 2001; see also the UML Web site at www.uml.org). Finally, requests for proposals (RFP) have been solicited for a major revision of UML
The UML version 2.0 specification is planned to be decomposed into four separate and complementary parts to better address UML’s increasing size and complexity by defining a language kernel that contains the core elements of the language that are used most of the time (see also the UML 2.0 Web site at www.celgent.com/omg/uml2wg.htm).

According to the UML specifications (OMG unified modeling language specification, 2001, chap. 1), UML represents a visual modeling language; it is not intended to be a visual programming language. It defines a semantic metamodel; it does not provide a tool interface, storage, or run-time model. UML was created to be independent from any particular programming language and development process. It supports a process that is use-case driven, architecture-centric, iterative and incremental (see also Oesterreich, 1997/1999, p. 50-52). This means that the identification of use cases is needed to build the foundation of the development process. The focus on the system’s architecture means that the process of application development needs to account for the properties of an application domain. Finally, the development process evolves in several phases or increments during which different components are created in repeated cycles or iterations.

Oesterreich (1997/1999, p. 52-95) suggests the following development process involving UML, which consists of a preliminary study and requirement analysis, coarse design and component creation, iterative incremental development, and system test and integration. The requirement analysis provides the necessary information about the underlying problem of the application domain. This information can be represented by business process models (also called analysis
model), use case models, activity models, CRC cards, mind maps, etc. These models include the properties and constraints of the application domain and its components; it is a description of what the system needs to be able to accomplish, not of how it will achieve this. For instance, the system under development will be integrated into an existing business process, which mostly organizational aspects. A business process is defined as a combination of related activities that are necessary to accomplish a business event, i.e. an instance of a business process (see also Forbrig, 2001, p. 138-140). In fact, UML provides an extension specifically designed for business modeling (OMG unified modeling language specification, 2001, chap. 4, part 2). Once the components models have been established, the iterative and incremental phase of the application development begins. This phase includes the stepwise planning and realization of the different components and subsystems identified in the requirements analysis phase. The overall development process is completed with testing and the stepwise integration of the application.

According to the UML semantics (OMG unified modeling language specification, 2001, chap. 2), UML consists of a four-layer metamodel architecture. First, the meta-metamodel prescribes the infrastructure of a metamodeling architecture. Second, the metamodel is an instance of a meta-metamodel. It defines the language for specifying a model. Third, the specific model is an instance of a metamodel. It defines a language to describe an information model. Finally, user
objects (a.k.a. user data) are an instance of a model. They define a specific information domain. Users can also extend UML to better serve their specific needs. These extensions are enabled through the use of stereotypes, tagged values, and constraints. For instance, Donnelly (2001) and Conallen (2000) provide concrete examples of how UML can be used to build Web applications. Conallen actually has developed a set of Web application extensions for UML that can be applied to a Web page and other HTML elements that are architecturally significant components of the system. And, Naiburg & Maksimchuk (2001) show how UML can be effectively applied to the design of databases.

6.2 The fundamentals of UML

As Jacobson (2000, p. 106) explains, UML provides users with a vocabulary to describe things, relationships, and diagrams in a model. There are three kinds of things, i.e. structural, behavioral, and groupings. Structural things include use cases, classes, interfaces, collaborations, components, and nodes. Behavioral things include interactions and state machines. And, groupings are comprised of models, systems, and packages. The second category—relationships—includes dependency, association, and generalization. The last category, diagrams, provides graphical means to describe views of a model, including use cases, classes, objects, sequences, collaborations, statecharts, activities, components, and deployment. The respective diagrams are grouped accordingly for use cases, classes, behaviors, and implementation. The models and diagrams developed with UML provide different views at different levels of fidelity of a system that is under analysis or development.
(OMG unified modeling language specification, 2001, chap. 3). In fact, Oesterreich (1997/1999, p. 8-9) suggests to concentrate only on a limited set of UML elements. This educated selection of elements to be employed should be based on the kind of application needed and the depth of detail required. UML includes provisions for the following diagrams and model elements as shown in Table 1 on page 89:

<table>
<thead>
<tr>
<th>Model Types</th>
<th>Diagrams and Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case model</td>
<td>Use case diagram</td>
</tr>
<tr>
<td></td>
<td>Use case</td>
</tr>
<tr>
<td></td>
<td>Actor</td>
</tr>
<tr>
<td>Static structure model</td>
<td>Class diagram</td>
</tr>
<tr>
<td></td>
<td>Object diagram</td>
</tr>
<tr>
<td></td>
<td>Basic elements:</td>
</tr>
<tr>
<td></td>
<td>Classes</td>
</tr>
<tr>
<td></td>
<td>Objects</td>
</tr>
<tr>
<td></td>
<td>Attributes</td>
</tr>
<tr>
<td></td>
<td>Methods</td>
</tr>
<tr>
<td></td>
<td>Relational elements:</td>
</tr>
<tr>
<td></td>
<td>Generalization</td>
</tr>
<tr>
<td></td>
<td>Specialization</td>
</tr>
<tr>
<td></td>
<td>Association</td>
</tr>
<tr>
<td></td>
<td>Aggregation</td>
</tr>
<tr>
<td>Behavioral model</td>
<td>Statechart diagram</td>
</tr>
<tr>
<td></td>
<td>Activity diagram</td>
</tr>
<tr>
<td></td>
<td>Interaction diagram:</td>
</tr>
<tr>
<td></td>
<td>Sequence diagram</td>
</tr>
<tr>
<td></td>
<td>Collaboration diagram</td>
</tr>
<tr>
<td>Implementation model</td>
<td>Component diagram</td>
</tr>
<tr>
<td></td>
<td>Deployment diagram</td>
</tr>
</tbody>
</table>

The next section provides an overview of the individual elements or diagrams based on the UML version 1.4 (OMG unified modeling language specification, 2001, chap. 3). The author has also consulted Oesterreich (1997/1999; 2001) and Forbrig (2001) in writing the following explanations. The
relationship among the individual models and diagrams can be summarized as follows (Figure 9 on page 90):

6.2.1 Use case diagram

A use case diagram shows the relationship among users (or actors) and a set of uses that are all part of a system. It is applied to describe how a business event is conducted. A use case, as described earlier, represents the interaction between a user and a system. It provides a view of the external behavior of a system from the perspective of a user. The use case model is comprised of a typical working process, including the individual steps or activities involved in the process. The diagram presents a graph of actors, a set of use cases, the relationships between these elements, and possibly additional information, e.g., interfaces. Each use case has a unique name and may include information about, e.g., actors, preconditions, postconditions, invariants, process description, rules, services, diagrams, etc. An actor is defined as a set of roles that users of an entity can play when interacting with the entity. For each use case, the actor may play a different role. In addition to
actors, one could also list chronological events, dialogs, external systems, and external passive objects (so-called entities), if they are involved in the use case.

6.2.2 Class diagram

UML provides means for describing the different elements of classes. As indicated earlier, a class is a definition of the attributes (i.e. data elements), the operations, and the semantics of a set of objects. Thus, in UML a rectangle is used to represent a class with these basic elements. It bears the name of the class and may show the attributes (its name, type, constraint, initial value) and the operations (parameter, constraint). Objects as instances of a class are also represented with a rectangle showing the instance name, attribute name and attribute value.

Operations are the services that may be required from an object. They are implemented by messages delivering a message to an object with the information about the activity that is expected to be carried out. Operations are listed in the lower part of a class notation.

The static behavior of the different elements is described in terms of inheritance (generalization, specialization), association, and aggregation. Empty arrows pointing from the subclass to the superclass indicate inheritance, i.e. properties that are structured hierarchically. A discriminator indicates the distinctive feature or characteristic that distinguishes superclasses from subclasses. Association describes the common semantics and structure of a set of object connections, i.e. the connection between classes. The concrete connection between objects is called object connection or link. An association is needed so that objects can communicate with each other. The cardinality of an association indicates the
number of elements. The multiplicity specifies the number of cardinality permitted in the association, i.e. the number of objects of the opposite class to which an object can be associated. The multiplicity may be optional (0) or indicated by a range with minimum and maximum values (e.g., 1..5). Also, the association, represented by an arrow between elements, should also be given a name to describe that this relation exists or why it exists. An aggregation describes an association in which the involved classes represent a whole-part hierarchy. In this hierarchy, the entirety assumes tasks in substitution of its parts. An aggregation is represented with a line drawn between two classes, and marked with a small diamond which is located at the aggregate. A special case of an aggregation is a composition, in which the parts are existence-depend on the whole. This relationship is also drawn like that of an association with the exception that the diamond on the side of the aggregate is solid instead of empty.

6.2.3 Behavioral diagrams

The dynamic behavior of the different elements is represented with behavioral diagrams. An activity diagram describes the procedural possibilities of a system in terms of activities. It is a special case of a state diagram. An activity is defined as a single step in a processing procedure. It represents a state that contains an internal action and at least one outgoing transition, which are triggered by the completion of actions or subactivities in the source state.

The interactions among instances are shown with interaction diagrams, i.e. collaboration diagram and sequence diagram. A collaboration diagram describes an interaction, which is organized around the roles of the interaction and their links to
each other. The chronological course of communication between the objects is shown by the numbering of messages. A sequence diagram is similar to a collaboration diagram. However, it focuses on the chronological sequence of messages. The objects are simply shown by vertical lifelines.

Finally, a statechart diagram is used to describe the behavior of a model element, such as an object or an interaction. Typically, it indicates the sequence of states an object can have during its lifetime. A state is defined as a condition during the life of an object or an interaction. States can be labeled with the following keywords: entry, exit, do, and include.

6.2.4 Implementation diagrams

Implementation diagrams show parts of an implementation, including source code structure and run-time implementation structure. There are two forms of implementation diagrams: First, component diagrams represent the structure of the code itself, and second, deployment diagrams show the structure of the run-time system.

6.2.5 Model management

The different elements of a model (e.g., classes and use cases) are grouped in packages. Packages provide a generic mechanism to organize model elements. These packages may also be hierarchically structured, i.e. they can be nested within other packages. A package is represented as a folder, indicating its name on the file tab when it contains other model elements.

Subsystems are another mechanism to describe the different elements of a model. In this case, the focus is on the behavioral units by describing the interfaces
and operations. A subsystem is represented in the same way as a package, with the addition of a fork symbol placed on the upper right corner of the rectangle.

Finally, a model represents an abstraction of a physical system, with a certain purpose. It is comprised of all elements needed to completely describe a physical system. The elements of a model are organized into a package / subsystem hierarchy.