Object-Oriented Paradigm Overview

2.1 Key Concepts

- Informal scenarios
- Object-oriented conceptualization
- Inheritance
- Software development life cycle
- Waterfall model
- Prototyping
- Incremental/iterative development
- Hybrid life cycles
- Module cohesion
- Module coupling
- Module reuse
- Unified Modeling Language (UML)

2.2 Getting Acquainted with the Class Project

Because the time frame for developing the class project is dictated by the length of the term for this course, it is essential that students begin familiarizing themselves with the class project immediately. The first step is to read the requirements specification provided in the previous chapter.

Most students taking an introductory software engineering course are well acquainted with programming and perhaps some program design issues. These activities comprise later stages of the software development process. Because much of their education has focused on implementation issues, students tend to struggle with the early stages of software engineering. In fact, these early stages are challenging even to seasoned professionals.

Once the requirements specification has been read, students should begin to create informal scenarios based on the specification. Through an informal
scenario, we tell a “story” of how users will be using the system. The story should be concrete. For example, a poorly written informal scenario tells a story of “user A entering all necessary data in the dialogue window” whereas a well-written informal scenario specifies that “Andrea enters her name and password in the login screen.”

2.2.1 Guidelines for Creating Informal Scenarios

Although we call this portion of the software development process informal, there are some guidelines for creating these informal scenarios to achieve the greatest benefit (see Summary Points box 2.1). The guidelines are as follows:

- A software system should consist of a number of small informal scenarios, rather than one large informal scenario.
- An informal scenario should address one coherent aspect of the system, such as user logs onto system, server deals cards to users, user makes a move, user wins/loses the game, and so on.
- Each informal scenario should specify concrete values whenever possible. Rather than writing that a user spins the spinner and moves the specified number of spaces, students should write that Andrea spins the spinner, which lands on the number 6. She then moves four spaces forward and two spaces left. The latter conveys the potential complexity of managing user moves on the game board, while the former does not.
- In the informal scenarios, you should address some types of user error that the system will handle, but you should not attempt to exhaustively cover all possible errors.
- Implementation details should be omitted in expressing each informal scenario. For example, an informal scenario should not mention linked lists or other data structures.
- Each scenario should describe the state of the system upon initiation of that scenario. For example, the user makes a move informal scenario should describe the example configuration of the game board prior to the user move.
- Each scenario should terminate by indicating which scenario is next.

The tendency is to make the informal scenarios too abstract rather than sufficiently concrete. The problem with characterizing the use of the system in abstract terms is that the complexities of the system are not as apparent when described abstractly as when a detailed situation is discussed. The point of creating informal scenarios is to allow the developers to gain increased understanding of the project to be developed.
2.2.2 Sample Informal Scenario: User Makes a Move

Current System State

The system state consists of each player at his or her starting location on the game board. The three players in the game, Andrea, Max, and Emma, have each been dealt six cards. Because the value of each card is irrelevant to this informal scenario, these values are omitted.

Informal Scenario

Andrea has the next move. She spins the spinner, which lands on the number 5. Andrea has the white playing piece. She moves this piece one space to the left, one space toward the top of the game board, two spaces to the right, and finally, one space to the top of the game board. Because of the final position of her game piece, Andrea has no additional options, and her turn ends.

Next Scenario

This scenario is repeated with the player to the left of Andrea. Therefore, Max has the next turn.

As a warm-up exercise to writing informal scenarios pertaining to the class project, create an informal scenario about using a dishwasher. Pick one of the following scenarios: loading the dishwasher, starting the dishwasher, or unloading the dishwasher.
Break the playing of the game for the class project into as many different informal scenarios as you can think of. Divide these scenarios among your project development team members. Each of you should come up with informal scenario descriptions using the guidelines of the previous subsection. This task is useful for quickly engaging everyone on the team in the project development and should be undertaken as early in the semester as possible. The remainder of this chapter will address general issues concerning object-oriented software engineering.

2.3 Object-Oriented Conceptualization

As discussed in the previous chapter, software development works best when all aspects and stages of the development process are compatible with each other. Therefore, one should conceptualize an object-oriented software project in terms of objects.

Recall the simple object-oriented conceptualization of the book-tracking system from the previous chapter (see Figure 1.3). This figure illustrates our first pass at selecting important objects that might comprise the system. Figure 2.1 depicts another library scene that may aid our selection of important objects for our system. We should note that for any software development project, the initial conceptualization represents a starting point, and further analysis may eliminate one or more of these objects or add other objects. In addition, this initial object configuration is probably similar to the configuration with which most people would start because the elements making up the conceptualization are familiar and common.

We should also note that objects comprising a software system are not always tangible objects that can be pictured. Objects are named with words that are nouns. If a noun can be nontangible, so can an object. In designing software systems, however, many people think of tangible objects first [57]. Ideas and representations of abstract, nontangible elements of a system are frequently essential objects of a system and must be added to the system at some point. For example, the book-tracking system may contain work unit objects in an effort to assess the contribution of each librarian to the running of the library. A work unit object may assign varying work units to the tasks accomplished by the library staff. The work required to reshelve a book might be different from the work required to check a book in. The work unit object determines how much work each task requires. We do not have a concrete picture of a work unit, but a work unit is a legitimate object nonetheless.
To conceptualize an object-oriented system, once we have selected the objects that comprise the system, we must characterize these objects in terms of their behavior. Characterizing object behavior requires thinking about objects of a particular class in relation to objects of other classes. Just as the initial selection of objects reflects a starting point and is expected to change during further analysis, the initial determination of interobject relationships can be expected to change after further analysis. There are three basic types of relationships that can exist between objects:

- Application specific
- Inheritance
- Aggregation/containment

EXERCISE 2.2
Consider a garden management system that automates all physical chores associated with gardening. If you were to conceptualize the problem in object-oriented terms, what objects would you start with?

2.3.1 Application-Specific Relationships
Figure 1.3 shows arrows between certain classes. The arrows are labeled to define the nature of the relationship. Each label represents a particular relationship. For
example, the arrow between the library patron and the book in Figure 2.2 has three labels: Checks out, Returns, and Requests. These three labels represent three relationships between library patrons and books and are examples of application-specific relationships. In particular, library patrons may check out, return, and request books. Similarly, the arrow between the librarian and the book in Figure 1.3 is labeled with Checks in, Checks out, and Reshelves, indicating the things a librarian can do with a book. These examples illustrate the fact that objects of two different classes can be related to each other through the message passing between them.

Another type of application-specific relationship is based on use. In this type of relationship, one object uses an object of another class. The used object frequently appears as a parameter in one of the user object's method definitions. For example, in the book-tracking system, perhaps a Patron object has a method for creating a list of Book objects that the Patron currently has checked out. The Patron object does not contain a list of these books but must create one by looking at all the checked-out books and who has the books checked out. The Patron object uses a List object that is not one of the attributes of the Patron object.

2.3.2 Inheritance

In an object-oriented system, classes can be related to each other through inheritance. Inheritance is a primitive interobject relationship. A primitive relationship is implemented through some aspect of the object-oriented programming language. Inheritance is a mechanism built into the programming language.

The inheritance relationship is important in many software systems. The book-tracking system thus far has no inheritance. If we extend the system slightly, however, we can begin to see the need for the inheritance relationship. For example, if the library owns items other than books, such as videos and compact discs, the
book-tracking system must keep track of all these items. Therefore, we could define a Resource class, which then acts as a generic parent class to the more specific Book, Video, and Compact Disc classes. The system must behave slightly differently when a patron checks out each of these items. For example, a book might be able to be checked out for two weeks, whereas a video can only be checked out for two days. The relationship between the Resource class and the three child classes is hierarchical because, for example, a Book is a Resource and therefore inherits all of the attributes and methods from the Resource class but has additional, more specific behavior.

Figure 2.3 shows another inheritance relationship among classes that may act as library patrons. This figure shows that there are four possible categories of library patrons: faculty, students, citizens, and library staff.

2.3.3 Aggregation/Composition

The second primitive interobject relationship is called aggregation/composition. This relationship occurs when one class contains another class. For example, the Patron class has an address as an attribute. If we create an Address class, a Patron object contains an Address object. A Patron object may be viewed as an aggregation of its attributes. When some of those attributes are objects from other classes, these classes are interrelated to each other by the aggregation/containment relationship.

Figure 2.4 depicts some of the building blocks that make up the Patron class. Some of these building blocks are other classes, such as the patron address and list of checked-out resources, while others are attributes, like the overdue fine and patron
name. When we address the aggregation/composition relationship, it is important to note that we are talking about relationships between classes and not attributes.

**EXERCISE 2.3** Review the objects that you selected in the previous exercise for the garden management system. What application-specific relationships exist between these objects? Do any aggregation or inheritance relationships exist between the objects?

### 2.3.4 Other Categorizations of Relationships

Although other categorizations for interobject relationships exist, the relationships we have described tend to be the common denominator of other such characterizations [14, 16, 23, 47, 88]. Because the **Unified Modeling Language (UML)** will be the object-oriented modeling notation of choice for this text, the primary interobject relationships used in that notation are outlined here. The authors of UML specify the following four relationships [16]:

- **Dependency**: One class may affect the semantics (meaning) of another, because objects of one class use instances of another, for example, as arguments in a parameter list.
- **Association**: One class has a link to another class. This relationship includes the aggregation and application-specific relationships described earlier.
- **Generalization**: One class is the super (parent) class of another class. This relationship is one half of the inheritance relationship described previously.
- **Realization**: One class provides a service for another class.
2.4 The Software Life Cycle

The software life cycle refers to the steps that one goes through to design, develop, implement, use, and maintain a software system. Within the software life cycle, the developer of the system has some choices to make. In particular, the developer must decide which software development paradigm to use. Once the development paradigm has been chosen, the developer must choose which software development process to use. We have discussed software development paradigms earlier. We will now discuss the software development process.

2.4.1 The Software Development Process

The software development process refers to the series of steps one goes through to develop a software system. Although the software development process can be selected independently of the software development paradigm, each paradigm lends itself most readily to a particular software development process. For example, the object-oriented approach is most readily associated with an evolutionary/incremental software development process. This association exists because class definitions lend themselves to iterative change [88]. Additional attributes and methods may be added to a class definition without disrupting its original functionality. Adding data elements to a process-oriented system tends to be more disruptive, because certain functions must include these new data elements, frequently requiring algorithm changes to accommodate these new data elements. Consequently, other software development processes are most easily associated with the process-oriented paradigm.

Waterfall Software Development Process

No discussion of software development processes is complete without discussing the classic waterfall model [8]. Figure 2.5 shows the waterfall model. This model is useful for discussion because it catalogs the traditional essential phases of software development, namely, requirements analysis, design, programming, testing, and maintenance. These basic phases of software development are part of every software development process, independent of the software development paradigm. Since each software development process shares the same basic phases, the difference between the various process approaches is in the duration, completeness, and sequence of the development phases. Additional phases such as risk assessment and quality assurance may be specified as separate phases or may be integrated into each of the basic phases.
EXERCISE 2.4 Contemplate the process of creating a garden from a previously undeveloped piece of land. What decisions and activities go into this process? Relate these activities to your perhaps sketchy understanding of the phases of software development as defined by the waterfall model.

Two primary types of variations of the waterfall model exist. In the first type, a single development phase is broken into more detailed phases [10]. For example, one may replace the design phase of the basic waterfall model with two distinct phases called product design and detailed design. Or, one may replace the test phase with component test, integration test, and user test [10]. In the second type of variation, additional phases are added to the traditional waterfall model. For example, one may explicitly specify the addition of a verification or quality assurance phase of development. In the traditional waterfall model, each development phase is verified before progressing to the next phase of development so that verification is implicitly part of each phase and thus is not shown separately.
The waterfall model has been criticized as overly rigid and therefore risky. In this model, one phase must have been completed before the next can be started. For example, the requirements analysis phase must be complete before any design work begins; the design phase must be complete before any programming can take place; and so on. Such a situation is possible only in the implementation of a system in a very well-defined field, where no experimentation is necessary. Even in such a well-defined field, however, this requirement of completion before moving on engenders huge risk because users will not have an actual, working piece of software to look at until much time and energy have been invested in the system. Although users can give feedback on a requirements analysis or a design, many issues will not become clear to them until they have actually had the chance to use pieces of the system. Therefore, the users may not completely understand the plan for the delivered system and will not see the flaws until the system has undergone testing. This initial investment without substantial user feedback must be avoided. One method for gathering user feedback in a realistic environment is the development of a **prototype**.

**Prototyping**

Prototyping is the process of building a system mock-up with minimal time investment. The resulting mock-up functions as an evaluation system with which the end users can interact. Prototyping is actually more of a software development technique that can be used in conjunction with any software development process than a software development process in its own right. Prototyping, however, can serve as the focal point of a software development process.

Two types of prototypes exist: **throwaway** prototypes, and **nonthrowaway** prototypes. A throwaway prototype is usually programmed in a programming language other than that intended to develop the production system. One may decide to create a throwaway prototype for a number of reasons, including the following:

- Platforms supporting the prototyping languages are not available for use as the platform of the resulting system.
- The prototyping language is too inefficient for the production system.
- The prototype is too poorly structured to retain.

Prototyping typically involves a dialogue between the end user and the system developers. During its development, the prototype is progressively altered in response to the end users’ critique, until a desirable system mock-up results. The advantages to this process are significant. First, the end user has a clear understanding of the functionality of the system to be built. Second, the end user has a critical role in creating the system specifications. It is often the case that the end user does
not truly understand the requirements of the system until the system has been completed. Therefore, the idea behind prototyping is to place a system in the hands of the end user as quickly as possible with as little time investment as possible. Third, with the development of a prototype, the system developers have an unambiguous system specification that has already met with the approval of the end users.

Prototypes are created quickly by implementing major features of interface and output only. Issues critical to a production system such as data integrity and security are ignored during prototype development. An effective prototyping language, therefore, is one that allows quick creation of graphical user interfaces and generation of reports. Programming languages such as Java, Perl, and Tcl/Tk are potential candidates for creating prototypes.

The disadvantage of prototyping is that the end user can easily mistake the prototype for the production system and therefore may not be sympathetic when the production system is not ready for months or years. Depending on the political clout of the end users, such a misunderstanding can pose a serious problem for the technical staff, who are assumed to be unresponsive because they do not deliver the production system immediately. The 20–80 rule regarding systems development states that 20 percent of the development time is required to create the basic structure of the system, including interfaces and files. The other 80 percent of development time is required to ensure data integrity, security, privacy, and foolproofing of the system.

Although some refer to prototyping as a type of development process [88], prototyping does not address issues pertaining to software structure. Prototyping includes no techniques to describe overall software structure or the design of software components, and, therefore, a modular, robust, extensible system is not guaranteed as a result of prototyping alone. We strongly suggest, however, that prototyping be utilized in conjunction with other development process approaches.

Incremental/Iterative Development Process

The object-oriented paradigm naturally lends itself to an incremental development process because of the ease with which class definitions may be evolved to contain increasingly more attributes and methods. In addition, new classes may be integrated into the solution easily and gradually. The phrase “evolving the system” means that an initial system consisting of a subset of functionality of the final target system is first created and is then incrementally modified to include more functionality until the final target system results. Note that this development process is completely different from the traditional waterfall development process in which each phase must be completed before moving onto the next phase. One may, however, select the traditional waterfall development process or any other approach to implement an object-oriented conceptualization of a system.
An incremental software development process still requires careful requirements analysis to produce a well-structured, quality software system. The difference between requirements analysis following a waterfall development process and analysis following an incremental development process is the timing, thoroughness, and frequency of the requirements analysis. In the waterfall model, the requirements analysis occurs once and is to be completed as the first phase of development. Thus the analysis is extensive and requires the systems analyst to understand the system in great detail before design work can proceed. In an incremental/iterative development process, an initial requirements analysis phase exists, but it is shorter and does not require a similar level of detailed understanding before design work can begin. The initial analysis is followed by design, programming, and testing resulting in a partial system, as shown in Figure 2.6. After verification by the end users, the partial system is enhanced with additional functionality, requiring additional analysis to take place. The analysis, design, programming, testing, and verification cycle then repeats itself until the target system is achieved.

The incremental/iterative development process has many advantages. It has the advantages of prototyping because the end users have portions of the system to critique early in the development cycle. The end users provide invaluable proof of concept by assessing finished portions of the system. When it follows the object-oriented or process-oriented paradigm, the incremental development process ensures modularity that produces a well-structured, robust, maintainable
system. Finally, in conjunction with object orientation, the incremental development process is likely to flow smoothly between each iteration of the system implementation. Assuming the system begins with a well-conceived initial class structure, there is no need to restructure existing portions of the system between the development iterations because well-designed classes are easily extended.

Simply combining an incremental development process with an object-oriented paradigm is no panacea, however. Careful preliminary analysis is critical to allow the development team to gain sufficient understanding of the target system to produce a class structure that will accommodate system evolution without restricting the emerging iterations of the system. The initial analysis must therefore have significant breadth, considering the entire system rather than simply the first iteration to be implemented.

EXERCISE 2.5 Contemplate how a garden may be developed incrementally. How would you define each developmental increment? Would this process help ensure the success of your garden more so than the waterfall model of gardening?

Hybrid Software Development Processes

A number of software development processes that combine the previously mentioned software development approaches have been proposed (see Summary Points box 2.2) [11, 28, 77]. The spiral model [11] combines an iterative approach with the control and structure of a more linear, sequential process like the waterfall model. The spiral model explicitly addresses issues of quality assurance and consists of six task regions. These regions convey the importance of the customer in the entire development process:

- Customer communication
- Planning
- Risk analysis
- Engineering
- Construction and release
- Customer evaluation

This task domain represents tasks that are repeated as the software project progresses in the spiral model. The customer is therefore the focal point throughout the entire software development process. The spiral model progresses by achieving a series of objectives or deliverables. These objectives embody the structure or sequential nature of the process. The first objective of the process is to produce a requirements statement, which specifies the functionality that the final system must contain. Each task of the task domain is applied to achieving this end. The
2.5 • Object-Oriented Modeling

SUMMARY POINTS 2.2

Types of Software Development Processes

1. Top-down, e.g., waterfall
2. Prototyping
3. Incremental/iterative
4. Hybrid, e.g., spiral

next objective is to produce a software design document, which again involves exercising each of the six tasks. The development process progresses through the series of deliverables, which include a development schedule, a test plan, and a maintenance schedule.

2.5 Object-Oriented Modeling

In this section, we will discuss modeling as a set of techniques for expressing the target system before the target system has been complete. Using these techniques, we will build a model of the target system. The model will express various aspects of the system to be developed. The representation chosen for communicating the model is a notation. The role of such a notation is to allow the expression of a series of representations that model various aspects of the system to be developed. The modeling of a software system requires the use of a particular notation that will unambiguously communicate a variety of aspects of the system to be developed.

2.5.1 Role of Model Building

The necessity of building models may be rationalized in a number of ways. One reason for modeling is to support the building of software systems whose complexities exceed our mental capacities. For example, most of us are unable to multiply multi-digit numbers together in our heads, but with the aid of a pencil and paper the task is trivial. Our unaided mental capacity has limitations, which can be augmented by techniques that allow us to record previous thinking. Such augmentation permits the shifting of our mental focus to unsolved portions of the problem. We will use a software engineering notation to record previous analysis and design decisions so that we may focus on new areas of the software system.

Model building addresses the complexity of the system to be developed by facilitating the creation of a series of models. The first model begins with an abstract representation of the system. Later models contain progressively more detail. Subsequent models are made to contain more detail than their predecessor models by focusing on a smaller portion of the system and extending the analysis to
produce additional depth in that portion. The process of model building, therefore, parallels the system developers’ understanding of the system. As the system analysts begin their study of the target system, their understanding is very general. During the model-building process, the models become increasingly detailed, and the system developers’ understanding also increases. Modeling software systems also creates concise and unambiguous representations, which facilitate communication between collaborating system developers. These precise representations can also be evaluated, verified, and corrected before significant time is invested in programming a poorly conceptualized system.

2.5.2 Creating Quality Modules

A modeling notation consists of representations of the elements from which the software system will be built. An important aspect of the notation expresses the units of modularity, which, in an object-oriented notation, are classes. When we focus our models on classes, which are software modules, the resulting software is likely to be modular. However, this focus on software modules is no guarantee of a successful software system. The modules must be well designed. Good module design requires that the modules be

- Cohesive
- Loosely coupled
- Encapsulated
- Reusable

For a module to be cohesive, its functionality must be well defined and well focused. A class must have a clear, easily expressed objective. Cohesion refers to the degree to which the internal elements of a module are bound to or related to each other. For example, a Book class is likely to be a cohesive module because the concept of a book is clearly delineated in the minds of most people. An additional requirement for cohesiveness in an object-oriented system is that each class contains methods that relate only to that class. For example, the Book class should contain attributes and methods associated with a single book. Therefore, the Book class should not contain a method to list all books alphabetically or an attribute that represents the total number of books in the library.

For a module to be loosely coupled, the module must be minimally connected to other modules. Coupling, therefore, refers to the degree to which modules are interconnected [101]. For example, the Book class is tightly coupled if it must invoke excessive numbers of methods of other classes in order to accomplish its tasks. Assume the Book class does not have an attribute to indicate whether the book is currently available, checked out, or waiting to be reshelved. To determine the status of a particular Book object, the book must check for itself on a list of
checked-out books and on a list of books waiting for reshelving. If the book does not find itself on either of these lists, the book can assume that it is available. In this example, the Book class is tightly coupled with the Checked-out List and with the Reshelve List. The Book class requires the use of other classes in order to accomplish a standard Book operation.

Encapsulated modules are those that engage in data hiding [83]. The attributes of an object should not be directly available to other objects. The attributes of the object are available only through a predefined interface that consists of the object’s public methods. A user of a class, therefore, must only understand how to use the methods of the class in order to successfully interact with objects of that class. Encapsulation implies that knowledge of the implementation of a particular class is not necessary for users of the class so that users of a class are insulated from the details of the inner workings of the class. Modules that are not encapsulated allow users direct access to their attributes or require understanding of how the class is implemented in order to use it successfully.

Software module reuse is a strategic objective of software engineering and is viewed as an essential approach for improving software development productivity [87]. Reuse, therefore, should be a driving motivation in module design. The implication of designing for reuse is that class functionality should be as broad and general as possible. To achieve maximum reuse, a class may require a few extra attributes or methods beyond the immediate scope of the target system. For example, designing a Book class for a library management system does not require the inclusion of the suggested retail price of the book as an attribute. The design of a Book class for a book store would require that this attribute be added. Conversely, the book store Book class does not require a due date attribute, while the library application clearly requires that this information be part of the Book class. Figure 2.7 shows the two different application areas for the Book class that require different attributes. A reusable book class should include all commonly required attributes and methods, encompassing all anticipated applications. The design of a class for all possible applications adds a great deal of overhead to a system development process and therefore may not be practical in all situations.

If the application areas for a particular class are sufficiently distinct, the creation of a class hierarchy may be necessary. For example, in Figure 2.7, a generic Book class is defined to contain the attributes and methods shared by all books, such as title, author, ISBN, and the respective methods using these attributes. Using inheritance, two additional book classes are derived from the more general Book class to accommodate the specific needs of each application of the class. In particular, a Retail Book class is created for the bookstore application, and a Library Book class is created for the library application. The Retail Book class contains the retail price and any other attributes relevant in a retail context but not relevant in a library
context. Similarly, the Library Book class contains library-specific attributes, such as due date.

Another strategy for creating reusable classes is to create classes that operate on very general data types. In Java, for example, all classes are derived from the same super class, which is called the Object class. Similarly, C++ has an explicit template mechanism, which allows the creation of a parameterized class. The template parameter determines the data type of the variables inside the class. For example, instead of creating a container class that serves as a sorted list of books, we can enhance the reusability of our sorted list by defining it as a template class that will allow the type of item to be sorted to be passed in as a parameter. The sorted list class can then be used any time we have a list of objects that we would like to sort.
2.5.3 Modeling Notation

Each software engineering paradigm has at least one associated notation for representing systems designed using the paradigm. The object-oriented paradigm has several notational options. These options differ primarily in the symbols used to represent various object-oriented concepts. For example, Figure 2.8 shows two different notations for object-oriented systems. The left side of the figure shows Booch notation [14], while the right side of the figure shows the Unified Modeling Language (UML) notation [15]. Both show the same relationship between the Patron class and the Book class.

Throughout the rest of this textbook, we will use UML [15, 16, 36] to represent our software system conceptualizations. UML is widely accepted in industry. According to its creators, prior to the existence of UML, there were several modeling languages with minor differences in their overall expressive power. These languages shared a common core of object-oriented constructs but differed slightly in their notation. This disagreement in notation discouraged software engineering practitioners from engaging in object-oriented modeling. The creators of UML set out to define an industry-standard notation to encourage object-oriented software development [15].

![FIGURE 2.8 An Illustration of Two Different Notations for Class Diagrams](image-url)
The stated goals [15] of UML are as follows:

- To provide users of UML with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
- To provide extensibility and specialization mechanisms to extend core object-oriented concepts.
- To be independent of particular programming languages and development processes.
- To encourage the growth of the object-oriented tools market.
- To support higher-level development concepts such as collaborations, frameworks, patterns, and components.
- To integrate the best software engineering practices.

The creators of UML wish to facilitate the evolution of the object-oriented paradigm by proposing a clear, easy-to-use notation for software modeling that practitioners can agree to use. If practitioners are using the same notation for their conceptualizations, they can easily exchange software models. The UML creators also wish to support a wide range of application types and have thus made provisions for extensions to UML in order to accommodate specialized areas such as real-time system development [15].

### 2.5.4 Use of Models in Software Engineering

The goals of model building during the software life cycle are these:

- To create a nonnarrative, unambiguous expression of objects, classes, and their interrelationships.
- To model a variety of perspectives of the target system for use during analysis and design.
- To express a series of system models that are initially abstract, then refined to be increasingly detailed.
- To facilitate communication and collaboration among technical personnel.
- To create complete, verified, and unambiguous programming specifications.
- To serve as system documentation to be modified to reflect changes made during system maintenance.
- To assist in project planning and management.
- To support quality assurance and verification activities.

As this list of objectives suggests, the set of models used in software engineering is central to the whole software life cycle. Figure 2.9 summarizes the associations between UML models (as expressed in UML diagrams) and the critical phases and tasks of software engineering. We will present a precise characterization of how
2.6 Qualities of a Good Object-Oriented System

The primary objective of applying good software engineering practices to a software project is to produce quality software in a timely, predictable manner. Two questions arise out of this statement. The first question is “What is quality software?” The second question is “Are the objectives of object-oriented software engineering any different from those of the process-oriented software approach?”

A feature that distinguishes object-oriented software development projects from process-oriented projects is that object-oriented development emphasizes the ability of the software to evolve smoothly over each software release. This emphasis implies that the software is easily extensible. The process-oriented approach emphasizes modularity in design so that maintenance is accomplished easily. Object-oriented development also emphasizes modularity, but the modules must also evolve easily. Extensibility of modules is, therefore, an objective of object-oriented development and is a quality of a good software system.

FIGURE 2.9 UML Diagrams and the Software Life Cycle

models are used to support software engineering activities as we discuss each software development phase in greater detail.

EXERCISE 2.6
Consider the process of building a toolshed again. How would you express the desired end product to a builder to ensure that the resulting toolshed meets the precise specifications you need? How does this act of modeling compare to the goals of model building during software engineering?
The desire for reusable software modules transcends the particular choice of software development approach. Certain mechanisms, however, are particularly conducive to the creation of reusable code. Many of these mechanisms are available in object-oriented programming languages. In particular, the inheritance mechanism, unique to object-oriented systems, suggests a strategy for software module reuse. One way of thinking about inheritance is that shared characteristics in a group of classes are extracted to form the parent classes and then the characteristics that differentiate classes are implemented in the child classes. The common code in the parent classes is used over and over (as inherited methods and attributes) in the child classes. Thinking of inheritance in this manner suggests that optimal use of inheritance hierarchies is another quality of a good object-oriented software system.

In the early 1990s, the International Standards Organization (ISO) produced a standard model for assessing the quality of all software independent of paradigm. This standard model, called ISO 9126, is shown in Figure 2.10. The model consists of six major characteristics, each of which has additional properties. The model indicates the qualities good software should possess but does not show how to assess these qualities. A good software development process must incorporate procedures for assessing the elements of quality software.

The ISO 9126 model serves to define the elements of quality software against which all software development methodologies should be compared. If a software development process has mechanisms for ensuring that software resulting from the development process possesses the qualities in the ISO 9126 model, we can feel more comfortable that the result of our development process will be quality soft-

![ISO 9126 Qualities of Good Software](image-url)
ware. **Functionality** addresses how well the resulting software product conforms to the client’s stated and implied needs. Functionality also addresses additional organizational requirements, such as security and interoperability. The domain experts are unlikely to request these qualities explicitly. **Reliability** relates to the robustness of the software and its relative functionality in the context of equipment and media failure. **Usability** describes how easily users learn to use the software, its intuitiveness and ergonomics, and its ability to interact with other systems used by the client organization. **Efficiency** relates to efficient computer resource utilization. **Maintainability** requires software to be easily modified and tested without compromising its stability. Finally, **portability** addresses the ability of the software to run on multiple platforms and its conformance to organizational standards.

### 2.7 Working in Teams

We will address additional information concerning holding effective team meetings, but first we will address a bit of computer science history and look at other ideas concerning how to structure a software engineering team.

#### 2.7.1 The Chief Programmer Team

In the context of a course in software engineering it is advisable that all students have exposure to all aspects of the SE process. Industry has other priorities and consequently has a variety of strategies for creating teams. One classic team structure is based on Harlan Mills’ work [68, 19] and is known as the chief programmer team. This team structure capitalizes on the idea that programming productivity varies between professional programmers by as much as an order of magnitude [91]. Another goal of the chief programming team structure is to minimize the interpersonal lines of communication necessary to develop the software system.

The chief programmer team, as the title suggests, centers on one highly productive programmer, the chief programmer. The remaining positions on the team exist to serve the chief programmer in some capacity. The following list describes Frederick Brooks’ variation on this classic team structure [19]:

- The **surgeon** is the chief programmer who “cuts” the problem into smaller pieces. This person does it all from creating functional specifications to testing the code, and carries responsibility for the entire project.
- The **copilot** is the surgeon’s able assistant who can engage in all the activities of the surgeon but has no ultimate responsibility for the project.
- The **administrator** takes care of any organizational tasks necessary for the team to operate.
Object-Oriented Paradigm Overview

The editor edits any documentation produced by the surgeon.

The program clerk maintains all files associated with developing the system.

The toolsmith is responsible for installing, maintaining, and possibly creating any support tools necessary for the team to operate optimally.

The tester creates test cases for testing the system.

The language lawyer is expert in the languages and tools used to develop the system.

The communication pattern proposed by this team structure requires everyone to communicate directly with the surgeon and thus reduces the number of lines of communication.

It is important to keep in mind that the proposal for this software engineering team structure was created prior to the existence of any real software engineering methodology. While the surgeon does engage in the creation of software specifications, this phase of development is not yet recognized as essential to the success of the overall project.

EXERCISE 2.7 Gather in your software engineering teams and discuss the following questions:

- Would a chief programmer team utilize people effectively? Why or why not?
- Would people in the various positions each have high levels of job satisfaction? Why or why not?
- Would you structure your team according to the chief programmer team model? Who on your team would fill each role? Would everyone be happy with this choice?
- Is it appropriate to place so much emphasis on programming by giving the team structure the name “chief programmer team”?
- Would you enjoy the role of editor? Why or why not?

2.7.2 Holding Effective Team Meetings

As mentioned in the previous chapter, a number of precautions can be taken to ensure that team meeting time is well spent. One of these precautions is to create and adhere to an agenda for every meeting. This of course presumes that the agenda addresses the tasks the team must accomplish for the coming week. The question to be answered here is, How does one create an effective agenda?

The process of software engineering is punctuated by a number of deliverables that result from the tasks that make up a particular software engineering methodology. In fact, when we teach a software engineering course, we time the topics to coincide with project deliverable due dates. Thus, the deliverables determine
the content of our classes. In the same way, these deliverables should be in the forefront of your mind when you are creating an agenda. You should ask yourself, “What deliverables must be completed in the coming week, and what group decisions are necessary in order to ensure they are completed?” Which tasks need to be assigned to team members to ensure the due dates are met? In fact, each meeting should end with each teammate having a clear understanding of what he or she needs to accomplish prior to the next class or team meeting. It is also important to make an effort to ensure that tasks are equitably assigned to each team member.

In order to give you a sense of the deliverables that will be required throughout the semester, Figure 2.11 lists each deliverable and the week that it is due. Your instructor may have additional deliverables or may allow you to revise specified deliverables after you receive feedback. For example, scenarios are due in week 2 and week 3. Thus students can receive feedback on their first attempts at scenarios and resubmit them in the third week for a grade. See also Summary Points box 2.3.

<table>
<thead>
<tr>
<th>Development Phase</th>
<th>Deliverable</th>
<th>Due Date</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Analysis</strong></td>
<td>1. Refined requirements specification</td>
<td>Week 2</td>
</tr>
<tr>
<td></td>
<td>2. Scenarios</td>
<td>Week 2 and week 3</td>
</tr>
<tr>
<td></td>
<td>3. Primary class list</td>
<td>Week 2</td>
</tr>
<tr>
<td></td>
<td>4. Class diagrams</td>
<td>Week 4</td>
</tr>
<tr>
<td></td>
<td>5. Use case diagrams</td>
<td>Week 4</td>
</tr>
<tr>
<td></td>
<td>6. Structured walk-through (in class)</td>
<td>End of week 4</td>
</tr>
<tr>
<td><strong>Product Design</strong></td>
<td>1. Object diagrams</td>
<td>Week 5</td>
</tr>
<tr>
<td></td>
<td>2. Refined class diagrams</td>
<td>Week 5</td>
</tr>
<tr>
<td></td>
<td>3. User interface mock-ups</td>
<td>Week 5–6</td>
</tr>
<tr>
<td></td>
<td>3. State machines</td>
<td>Week 5</td>
</tr>
<tr>
<td><strong>Class Design</strong></td>
<td>1. Collaboration diagrams</td>
<td>Week 6</td>
</tr>
<tr>
<td></td>
<td>2. Sequence diagrams</td>
<td>Week 7</td>
</tr>
<tr>
<td></td>
<td>3. Object diagrams</td>
<td>Week 7</td>
</tr>
<tr>
<td></td>
<td>4. Refined class diagrams</td>
<td>Week 8</td>
</tr>
<tr>
<td></td>
<td>5. Class skeletons</td>
<td>Week 8</td>
</tr>
<tr>
<td></td>
<td>6. Informal walk-through (in class)</td>
<td>End of week 8</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td>1. Implementation plan</td>
<td>Week 8</td>
</tr>
<tr>
<td></td>
<td>2. Source code</td>
<td>Beginning of week 12</td>
</tr>
<tr>
<td><strong>Testing</strong></td>
<td>1. Test plan</td>
<td>Week 9</td>
</tr>
<tr>
<td></td>
<td>2. Test analysis report</td>
<td>End of Week 12</td>
</tr>
<tr>
<td></td>
<td>3. System integration</td>
<td>Week 14</td>
</tr>
<tr>
<td></td>
<td>4. System delivery and demo</td>
<td>Week 15</td>
</tr>
</tbody>
</table>

**FIGURE 2.11 Class Project Deliverables and Approximate Due Dates**
### SUMMARY POINTS 2.3

**Effective Teams**

1. Use the completion of deliverables as objectives when creating agendas for your team meetings.
2. Do not end a team meeting unless each team member has a clear idea of what he or she should accomplish.
3. Keep team meetings well focused on specific objectives.
4. Assign tasks to team members as equitably as possible.

### 2.8 Questions for Review

1. Create an object-oriented conceptualization for an object-oriented personal finance system. Sketch out the primary classes that you find necessary and explain how each relates to the others. Are there any intangible classes included? Is a class hierarchy possible? What class aggregation exists?
2. Outline a software development process that is structured like the waterfall model but uses prototyping as a technique. Does prototyping necessarily alter the sequence of software development techniques?
3. Create a software development process that has prototyping as its main technique. Which phases do you include?
4. Role-play with a classmate. One of you is an end user who wants an automated music system. The other is the systems analyst. How do you use prototyping to elicit the system requirements?
5. Why is an iterative/incremental development approach associated with the object-oriented paradigm?
6. Can the use of single inheritance introduce any redundancy in a class hierarchy? If so, describe the redundancy. Sketch out an inheritance hierarchy that illustrates this.
7. What characteristics do well-structured modular systems share?
8. What considerations are necessary to create reusable modules? Why is it advantageous to maximize reusability?
9. Compare and contrast the programming constructs of Java and C++ that relate to objects.
10. How are software models used in the process of software development?
11. Why is a unified software modeling notation important?
12. How are the objectives of object-oriented software engineering different from process-oriented software engineering?
13. For the class project, design a window to be used by each player to keep track of the evidence shown and the questions asked.