Chapter 1
Introduction

The object-oriented paradigm is currently the most popular way of analysing, designing, and developing application systems, especially large ones. To obtain an understanding of this paradigm, we could begin by asking: *What exactly does the phrase 'object-oriented' mean?* Looking at it quite literally, labelling something as 'object-oriented' implies that objects play a central role, and we elaborate this further as *a perspective that views the elements of a given situation by decomposing them into objects and object relationships.* In a broad sense, this idea could apply to any setting and examples of its application can in fact be found in business, chemistry, engineering and, even philosophy. Our business is with creating software and therefore this book concentrates on the object-oriented analysis, design, and implementation of software systems. Our situations are therefore problems that are amenable to software solutions, and the software systems that are created in response to these problems.

Designing is a complex activity in any context simply because there are competing interests and we have to make critical choices at each step with incomplete information. As a result, decisions are often made using some combination of rules of thumb derived from past experience. Software design is no exception to this, and in the process of designing a system, there are several points where such decisions have to be made. Making informed choices in any field of activity requires an understanding of the underlying philosophy and the forces that have shaped it. It is therefore appropriate to start our study of object-oriented software analysis and design by outlining its philosophy and the developments in this field up to the present time. Throughout the case studies used in this text, the reader will find examples of how this guiding philosophy is helping us make choices at all stages.

This chapter, therefore, intends to give the reader a broad introduction to the complex topic of object-oriented software development. We start with an overview of the circumstances that motivated its development and why it came to be the desired approach for software development. In the course of this discussion, we present

B. Dathan and S. Ramnath, *Object-Oriented Analysis, Design and Implementation*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-24280-4_1

the central concepts that characterise the methodology, how this development has influenced our view of software, and some of its pros and cons. We conclude by presenting a brief history of the evolution of the object-oriented approach.

1.1 What Is Object-Oriented Development?

The traditional view of a computer program is that of a process that has been encoded in a form that can be executed on a computer. This view originated from the fact that the first computers were developed mainly to automate a well-defined process (i.e., an algorithm) for numerical computation, and dates back to the first stored-program computers. Accordingly, the software creation process was seen as a translation from a description in some 'natural' language to a sequence of operations that could be executed on a computer. As many would argue, this paradigm is still the best way to introduce the notion of programming to a beginner, but as systems became more complex, its effectiveness in developing solutions became suspect. This change of perspective on part of the software developers happened over a period of time and was fuelled by several factors including the high cost of development and the constant efforts to find uses for software in new domains. One could safely argue that the software applications developed in later years had two differentiating characteristics:

- Behaviour that was hard to characterise as a process
- Requirements of reliability, performance, and cost that the original developers did not face

The 'process-centred' approach to software development used what is called *top-down functional decomposition*. The first step in such a design was to recognise what the process had to deliver (in terms of input and output of the program), which was followed by decomposition of the process into *functional modules*. Structures to store data were defined and the computation was carried out by invoking the modules, which performed some computation on the stored data elements. The life of a process-centred design was short because changes to the process specification (something relatively uncommon with numerical algorithms when compared with business applications) required a change in the entire program. This in turn resulted in an inability to reuse existing code without considerable overhead. As a result, software designers began to scrutinise their own approaches and also study design processes and principles that were being employed by engineers in other disciplines. Cross-pollination of ideas from other engineering disciplines started soon after, and the disciplines of 'software design' and 'software engineering' came into existence.

In this connection, it is interesting to note the process used for designing simple electromechanical systems. For several decades now, it has been fairly easy for people with limited knowledge of engineering principles to design and put together simple systems in their backyards and garages. So much so, it has become a hobby that even a 10 years old could pursue. The reasons for this success are easy to see: *easily understandable designs, similar (standard) solutions for a host of problems, an*

easily accessible and well-defined 'library' of 'building-blocks', interchangeability of components across systems, and so on. Some of the pioneers in the field of software design began to ask whether they could not also design software using such 'off-the-shelf' components. The object-oriented paradigm, one could argue, has really evolved in response to this outlook. There are, of course, several differences with the hardware design process (inevitable, because the nature of software is fundamentally different from hardware), but parallels can be drawn between many of the defining characteristics of hardware design and what today's advocates of good software design recommend. This methodology, as we shall see in the chapters to follow, provides us with a step-by-step process for software design, a language to specify the output from each step of the process so that we can transition smoothly from one stage to the next, the ability to reuse earlier designs, standard solutions that adhere to well-reasoned design principles and, even the ability to incrementally fix a poor design without breaking the system.

The overall philosophy here is to define a software system as a collection of objects of various types that interact with each other through well-defined interfaces. Unlike a hardware component, a software object can be designed to handle multiple functions and can therefore participate in several processes. A software component is also capable of storing data, which adds another dimension of complexity to the process. The manner in which all of this has departed from the traditional process-oriented view is that instead of implementing an entire process end-to-end and defining the needed data structures along the way, we first analyse the entire set of processes and from this identify the necessary software components. Each component represents a data abstraction and is designed to store information along with procedures to manipulate the same. The execution of the original processes is then broken down into several steps, each of which can be logically assigned to one of the software components. The components can also communicate with each other as needed to complete the process.

1.2 Key Concepts of Object-Oriented Design

During the development of this paradigm, as one would expect, several ideas and approaches were tried and discarded. Over the years the field has stabilised so that we can safely present the key ideas whose soundness has stood the test of time.

The Central Role of Objects

Object-orientation, as the name implies, makes objects the centrepiece of software design. The design of earlier systems was centred around processes, which were susceptible to change, and when this change came about, very little of the old system was 're-usable'. The notion of an object is centred around a piece of data and the operations (or **methods**) that could be used to modify it. This makes possible the creation of an abstraction that is very stable since it is not dependent on the changing

requirements of the application. The execution of each process relies heavily on the objects to store the data and provide the necessary operations; with some additional work, the entire system is 'assembled' from the objects.

The Notion of a Class

Classes allow a software designer to look at objects as different types of entities. Viewing objects this way allows us to use the mechanisms of classification to categorise these types, define hierarchies and engage with the ideas of specialisation and generalisation of objects.

Abstract Specification of Functionality

In the course of the design process, the software engineer specifies the properties of objects (and by implication the classes) that are needed by a system. This specification is abstract in that it does not place any restrictions on how the functionality is achieved. This specification, called an **interface** or an **abstract class**, is like a *contract* for the implementer which also facilitates formal verification of the entire system.

A Language to Define the System

The Unified Modelling Language (UML) has been chosen by consensus as the standard tool for describing the end products of the design activities. The documents generated in this language can be universally understood and are thus analogous to the 'blueprints' used in other engineering disciplines.

Standard Solutions

The existence of an object structure facilitates the documenting of standard solutions, called **design patterns.** Standard solutions are found at all stages of software development, but design patterns are perhaps the most common form of reuse of solutions.

An Analysis Process to Model a System

Object-orientation provides us with a systematic way to translate a functional specification to a *conceptual design.* This design describes the system in terms of *conceptual classes* from which the subsequent steps of the development process generate the *implementation classes* that constitute the finished software.

The Notions of Extendability and Adaptability

Software has a flexibility that is not typically found in hardware, and this allows us to modify existing entities in small ways to create new ones. *Inheritance*, which creates a new *descendant class* that modifies the features of an existing (**ancestor**) class, and **composition**, which uses objects belonging to existing classes as elements to constitute a new class, are mechanisms that enable such modifications with classes and objects.

1.3 Other Related Concepts

As the object-oriented methodology developed, the science of software design progressed too, and several desirable software properties were identified. Not central enough to be called object-oriented concepts, these ideas are nonetheless closely linked to them and are perhaps better understood because of these developments.

1.3.1 Modular Design and Encapsulation

Modularity refers to the idea of putting together a large system by developing a number of distinct components independently and then integrating these to provide the required functionality. This approach, when used properly, usually makes the individual modules relatively simple and thus the system easier to understand than one that is designed as a monolithic structure. In other words, such a design must be *modular*. The system's functionality must be provided by a number of well-designed, cooperating modules. Each module must obviously provide certain functionality that is clearly specified by an interface. The interface also defines how other components may interact or communicate with the module.

We would like that a module clearly specify what it does, but not expose its implementation. This separation of concerns gives rise to the notion of **encapsulation**, which means that the module hides details of its implementation from external agents. The **abstract data type (ADT)**, the generalisation of primitive data types such as integers and characters, is an example of applying encapsulation. The programmer specifies the collection of operations on the data type and the data structures that are needed for data storage. Users of the ADT perform the operations without concerning themselves with the implementation.

1.3.2 Cohesion and Coupling

Each module provides certain functionality; **cohesion** of a module tells us how well the entities within a module work together to provide this functionality. Cohesion is a measure of how focused the responsibilities of a module are. If the responsibilities of a module are unrelated or varied and use different sets of data, cohesion is reduced. Highly cohesive modules tend to be more reliable, reusable, and understandable than less cohesive ones. To increase cohesion, we would like that all the constituents contribute to some well-defined responsibility of the module. This may be quite a challenging task. In contrast, the worst approach would be to arbitrarily assign entities to modules, resulting in a module whose constituents have no obvious relationship.

Coupling refers to how dependent modules are on each other. The very fact that we split a program into multiple modules introduces some coupling into the system. Coupling could result because of several factors: a module may refer to variables defined in another module or a module may call methods of another module and use the return values. The amount of coupling between modules can vary. In general, if modules do not depend on each others implementation, i.e., modules depend only on the published interfaces of other modules and not on their internals, we say that the coupling is *low.* In such cases, changes in one module will not necessitate changes in other modules as long as the interfaces themselves do not change. Low coupling allows us to modify a module without worrying about the ramifications of the changes on the rest of the system. By contrast, *high* coupling means that changes in one module would necessitate changes in other modules, which may have a domino effect and also make it harder to understand the code.

1.3.3 Modifiability and Testability

A software component, unlike its hardware counterpart, can be easily modified in small ways. This modification can be done to change both *functionality* and *design*. The ability to change the functionality of a component allows for systems to be more **adaptable**; the advances in object-orientation have set higher standards for adaptability. Improving the design through incremental change is accomplished by *refactoring*, again a concept that owes its origin to the development of the object-oriented approach. There is some risk associated with activities of both kinds; and in both cases, the organisation of the system in terms of objects and classes has helped develop systematic procedures that mitigate the risk.

Testability of a concept, in general, refers to both *falsifiability*, i.e., the ease with which we can find counterexamples, and the *practical feasibility* of reproducing such counterexamples. In the context of software systems, it can simply be stated as the ease with which we can find bugs in a software and the extent to which the structure of the system facilitates the detection of bugs. Several concepts in software testing (e.g., the idea of *unit testing*) owe their prominence to concepts that came out of the development of the object-oriented paradigm.

1.4 Benefits and Drawbacks of the Paradigm

From a practical standpoint, it is useful to examine how object-oriented methodology has modified the landscape of software development. As with any development, we do have pros and cons. The advantages listed below are largely consequences of the ideas presented in the previous sections.

1. Objects often reflect entities in application systems. This makes it easier for a designer to come up with classes in the design. In a process-oriented design, it is much harder to find such a connection that can simplify the initial design.
2. Object-orientation helps increase productivity through reuse of existing software. Inheritance makes it relatively easy to extend and modify functionality provided by a class. Language designers often supply extensive libraries that users can extend.
3. It is easier to accommodate changes. One of the difficulties with application development is changing requirements. With some care taken during design, it is possible to isolate the varying parts of a system into classes.
4. The ability to isolate changes, encapsulate data, and employ modularity reduces the risks involved in system development.

The above advantages do not come without a price tag. Perhaps the number one casualty of the paradigm is efficiency. The object-oriented development process introduces many layers of software, and this certainly increases overheads. In addition, object creation and destruction is expensive. Modern applications tend to feature a large number of objects that interact with each other in complex ways and at the same time support a visual user interface. This is true whether it is a banking application with numerous account objects or a video game that has often a large number of objects. Objects tend to have complex associations, which can result in *non-locality*, leading to poor memory access times.

Programmers and designers schooled in other paradigms, usually in the imperative paradigm, find it difficult to learn and use object-oriented principles. In coming up with classes, inexperienced designers may rely too heavily on the entities in the application system, ending up with systems that are ill-suited for reuse. Programmers also need acclimatisation; some people estimate that it takes as much as a year for a programmer to start feeling comfortable with these concepts. Some researchers are of the opinion that the programming environments also have not kept up with research in language capabilities. They feel that many of the editors and testing and debugging facilities are still fundamentally geared to the imperative paradigm and do not directly support many of the advances such as design patterns.

1.5 History

History of the object-oriented programming approach could be traced to the idea of ADTs and the concept of objects in Simula 67 programming language, which was developed in the 1960s for performing simulations. The first true object-oriented programming language that appeared before the larger software development community was Smalltalk in 1980, developed at Xerox PARC. Smalltalk used objects and messages as the basis for computation. Classes could be created and modified dynamically. Most of the vocabulary in object-oriented paradigm has originated from this language.

Toward the end of the 1970s, Bjarne Stroustrup, who was doing doctoral work in England, needed a language for doing simulation of distributed systems. He developed a language based on the class concept in Simula, but this language was not particularly efficient. However, he pursued his attempt and developed an object-oriented language at Bell Laboratories as a derivative of C, which would blossom into one of the most successful programming languages, C++. The language was standardised in 1997 by the American National Standards Institute (ANSI).

The 1980s saw the development of several other languages such as ObjectLisp, CommonLisp, Common Lisp Object System (CLOS), and Eiffel. The rising popularity of the object-oriented model also propelled changes to the language Ada, originally sponsored by the U.S. Department of Defense in 1983. This resulted in Ada 9x, an extension to Ada 83, with object-oriented concepts including inheritance, polymorphism, and dynamic binding.

The 1990s saw two major events. One was the development of the Java programming language in 1996. Java appeared to be a derivative of C++, but many of the controversial and troublesome concepts in C++ were deleted in it. Although it was a relatively simple language when it was originally proposed, Java has undergone substantial additions in later versions making it a moderately difficult language. Java also comes with an impressive collection of libraries (called packages) to support application development. A second watershed event was the publication of the book *Design Patterns* by Gamma et al. in 1994. The book considered specific design questions (23 of them) and provided general approaches to solving them using object-oriented constructs. The book (as also the approach it advocated) was a huge success as both practitioners and academicians soon recognised its significance.

The last few years saw the acceptance of some dynamic object-oriented languages that were developed in the 1990s. Dynamic languages allow users more flexibility, for example the ability to dynamically add a method to an object at execution time. One such language is Python, which can be used for solving a variety of applications including web programming, databases, scientific and numeric computations and networking. Another dynamic language, Ruby, is even more object-oriented in that everything in the language, including numbers and primitive types, is an object.

1.6 Discussion and Further Reading

In this chapter, we have given an introduction to object-oriented paradigm. The central object-oriented concepts such as classes, objects, and interfaces will be elaborated in the next three chapters. Cohesion and coupling, which are major software design issues, will be recurring themes for most of the text.

The reader would be well-advised to learn or refresh the non-object-oriented concepts of the Java language by reading Appendix before moving onto the next chapter. It is worthwhile and enjoyable to read a short history of programming languages from

a standard text on the subject such as Sebesta [1]. The reader might also find it helpful to get the perspectives of the designers of object-oriented languages (such as the one given on C++ by Stroustrup [2]).

1.7 Exercises

1. Identify the players who would have a stake in software development process. What are the concerns of each? How would they benefit from the object-oriented model?
2. Think of some common businesses and the activities software developers are involved in. What are the sets of processes they would like to automate? Are there any that need software just for one process?
3. How does the object-oriented model support the notion of ADTs and encapsulation?
4. Consider an application that you are familiar with, such as a university system. Divide the entities of this application into groups, thus identifying the classes.
5. In Question 4, suppose we put all the code (corresponding to all of the classes) into one single class. What happens to cohesion and coupling?
6. What are the benefits of learning design patterns?

References

1. R.W. Sebesta, *Concepts of Programming Languages* (Addison-Wesley, Boston, 2007)
2. B. Stroustrup, *The Design and Evolution of C++* (Addison-Wesley, Boston, 1994)