**Chapter 1: Understanding the Basics–Object-Oriented (OO) Concepts**

# 1.1 A Brief Overview of OO Concepts

This chapter discusses the concepts that make up the foundation of OO development techniques. For example, many concepts that go to the heart of OO—encapsulation, inheritance, and class and object etc—come from software engineering. These concepts are important because they underpin/support good design regardless of the technology you are working with, good design is a big part of object orientation, there is a lot more to it than that.

**Object-oriented System analysis and design (OOSAD)** is a popular technical approach for analyzing, designing an application, system, or business by applying the [object-oriented paradigm](https://en.wikipedia.org/wiki/Object-oriented_programming) and visual modeling throughout the [development life cycles](https://en.wikipedia.org/wiki/Software_development_process) to foster better stakeholder communication and product quality.

**OOSAD** is the use of modeling( using class. object, Actors, use case, use case diagrams, class diagram, sequence diagram ,etc) to [define](http://www.businessdictionary.com/definition/define.html) and [analyze](http://www.businessdictionary.com/definition/analyze.html) the [requirements](http://www.businessdictionary.com/definition/requirements.html) necessary for [success](http://www.businessdictionary.com/definition/success.html) of a [system.](http://www.businessdictionary.com/definition/system.html)

According to the popular guide [Unified Process,](https://en.wikipedia.org/wiki/Unified_Process) OOSAD in modern software engineering is best conducted in an iterative and incremental way. Iteration by iteration, the outputs of OOSAD activities, analysis models for object oriented analysis (OOA) and design models for object oriented system design (OOD) respectively, will be refined and evolve continuously driven by key factors like risks and business value.

***Software Engineering is the field of study that applies the principles of engineering for software development. Software Engineers are person who develop Software.***

Just as there was more to the structured paradigm/standard than a few simple concepts, there is also more to the OO paradigm. To give you a taste for what this chapter is about, some of the concepts and the terms are summarized briefly in Table 1.1.

|  |  |  |
| --- | --- | --- |
|  | Table 1.1: A Summary of Object-Oriented Concepts and Terms | |
| **Term** |  | **Description** |
| Abstraction |  | The essential characteristics of an item (perhaps a class or |
| Table 1.1: A Summary of Object-Oriented Concepts and Terms | | |
| **Term** | | **Description** |
|  | | operation) |
| Association | | A relationship between two classes or objects |
| Attribute | | Something a class knows (data/information) |
| Class | | A software abstraction of similar objects, a template from which objects are created |
| Collaboration | | Classes work together (collaborate) to fulfill their responsibilities |
| Encapsulation | | The grouping of related concepts into one item, such as a class or a component |
| Information hiding | | The restriction of external access to attributes |
| Inheritance | | Relationships defined as "is a" and "is like" |
| Instance | | An object that is an example of a particular class |
| Instantiate | | To create objects from class definitions |
| Interface | | A collection of one or more operation signatures that defines a cohesive set of behaviors |
| Message | | A request either for information or to perform an action |
| Messaging | | The process of collaboration between objects by sending messages to each other |
| Method | | A process implemented by a class that performs an action of value (similar to a function in structured programming) |
| Object | | A person, place, thing, event, concept, screen, or report, based on a class definition |
| Override | | To redefine attributes and/or methods in subclasses so that they are different from the definition in the superclass |
| Pattern | | A reusable solution to a common problem taking relevant forces into account |
| Multiple inheritance | | The direct inheritance from more than one class |
| Single inheritance | | The direct inheritance from only one class |
| Subclass | | A class that inherits from another class |
| Superclass | | A class from which another class inherits |

# 1.2 OO Concepts from a Structured Point of View

Let us see the basic OO concepts in detail

# Objects and Classes

* The OO paradigm is based on building systems from items called objects. An **object** is any person, place, thing, event, concept, screen, or report. A **class** generalizes/represents a collection of similar objects and is effectively a template from which to create objects. In a university system, Azeb is a student object and student is a class. In a banking system, Azeb is a customer object and customer is a class.

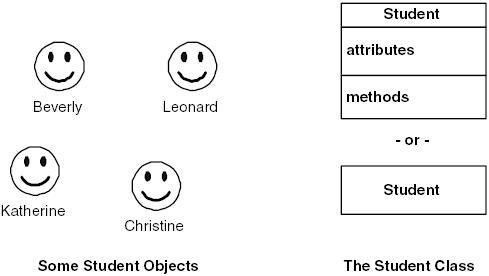


Figure 1.1: Student objects in the real world and two ways to model the class *Student*.

* Figure 1.1 depicts how we have student objects and how we model the class *Student.* It also shows the standard notations to model a class using the UML (Unified Modeling Language). Classes are typically modeled in one of two ways: as either a rectangle that lists its attributes and methods or as just a rectangle (in fact a class box can have as many sections as needed, but one or three sections are the norm). There are reasons for modeling classes either way. On the one hand, listing the attributes and methods can be quite helpful. It enables readers of your class models to gain a better understanding of your design at a single glance. On the other hand, listing the attributes and methods can clutter your diagrams and obscure readability.

* ***Class names are typically singular nouns***. The name of a class should be one or two words, usually a noun, and should accurately describe the class using common business terminology. You should model classes with names like *Student, person*, customer ,Book , *Course*, etc.
* Class can also represent concepts that are not nouns, like the process of checking out a book from a library.
* When object-oriented software is running, objects are instantiated (created/defined) from classes. We say an object is an instance of a class and we instantiate those objects from classes.

# Attributes and Operations/Methods

The object-oriented paradigm is based on the concepts that systems should be built out of objects, and that objects have both data and functionality. Attributes define the data, while methods define the functionality.

When you define a class, you must define the attributes it has, as well as its methods. The definition of an attribute is straightforward. You define its name, perhaps its type (whether it is a number, a string, or a date, and so forth). You should define the type of an attribute before you actually use it. You may also choose to indicate any business rules or constraints applicable to the attribute, such as the valid values the attribute may have.

The definition of a **method** is simpler: you define the logic for it, just as you would code for a function or a procedure. An important implication is that methods do one of two things: they return a value and/or they do something of value; that is, they have a side effect.

In Fig. 1.2, you see two different types of objects: a *student* and a *seminar*. Both objects know and do certain things, and you want to make sure you record this in your models, as you see in Fig. 1.3. The three-section class notation in this case: the top section for the name, the middle section to list the attributes, and the bottom section to list the methods.

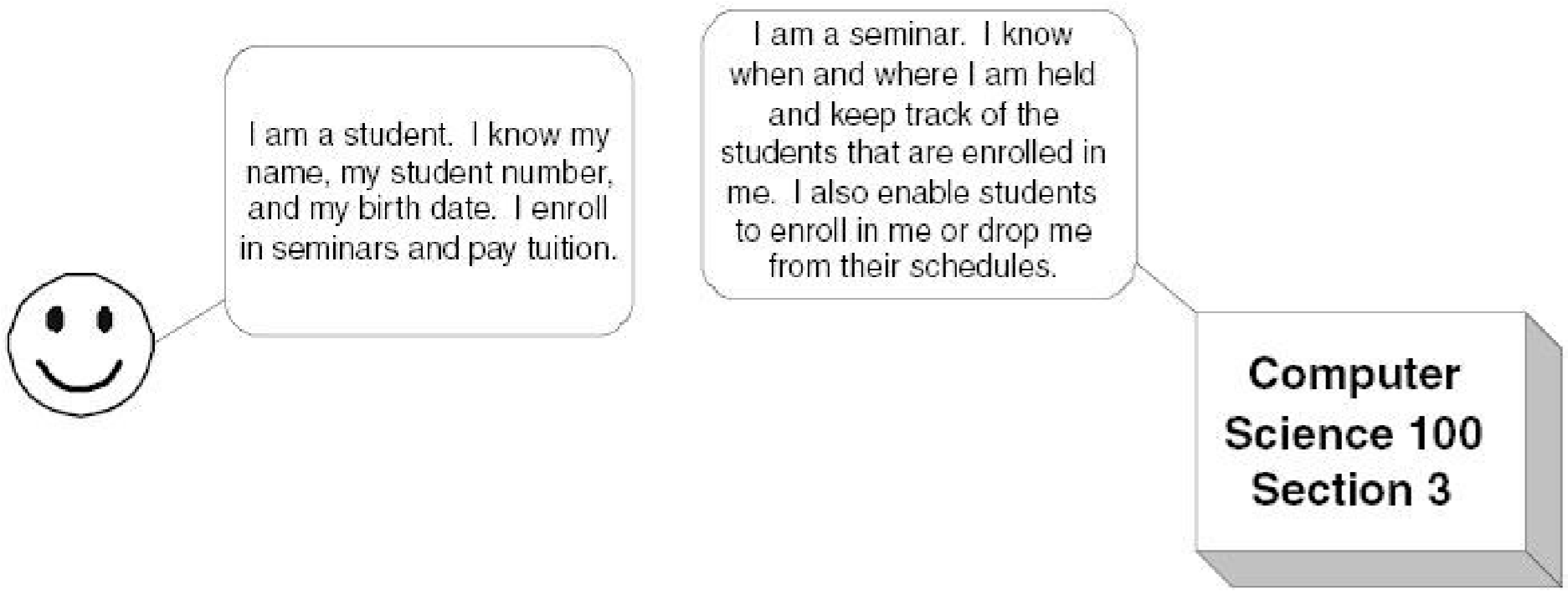


Figure 1.2: Objects in the "real world."

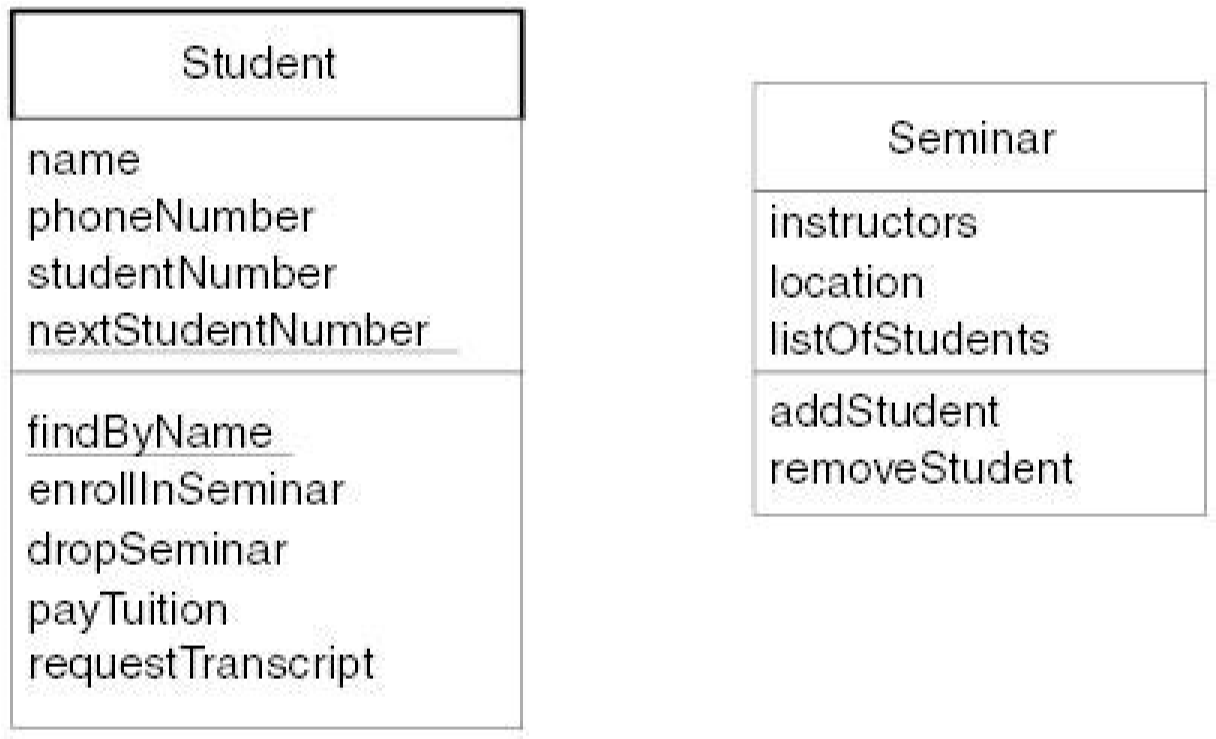


Figure 1.3: The *Student* and *Seminar* classes.

**Abstraction**

The world is a complicated place. To deal with that complexity we form abstractions of the things in it. For example, consider the abstraction of a person. From the point of view of a university, it needs to know the person's name, address, telephone number, social security number, and educational background. From the point of view of the police, they need to know a person's name, address, phone number, weight, height, hair color, eye color, and so on. It is still the same person, just a different abstraction, depending on the application at hand.

Abstraction is an analysis issue that deals with what a class knows or does. Your abstraction should include the responsibilities, the attributes, and the methods of interest to your application—and ignore the rest. That is why the abstraction of a student would include the person's name and address, but probably not his or her height and weight. ***OO systems abstract only what they need to solve the problem at hand***.

Abstraction is a method of collecting relevant information (possible attributes and methods) from the given problem domain.

# Encapsulation

Although the act of abstraction tells us that we need to store a student's name and address, as well as be able to enroll students in seminars, it does not tell us how we are going to do this. Encapsulation deals with the issue of how you intend to **modularize** the features of a system. In the object-oriented world, you modularize/bind systems into classes, which, in turn, are modularized into methods and attributes. We say that we encapsulate behavior into a class or we encapsulate functionality into a method.

Encapsulation is a design issue that deals with how functionality is compartmentalized /grouped within a system. You should not have to know how something is implemented to be able to use it.

Encapsulation is the grouping of related items into one unit.

* Attributes and behaviors /methods are encapsulated to create objects.
* Implementation details are hidden from the outside world.
* The packaging of operations and attributes representing state into an object type so that state is accessible or modifiable only through the objects' interface**.**

# Information Hiding

To make your applications maintainable, you want to ***restrict access*** to data attributes and some methods. The basic idea is this: if one class wants information about another class, it should have to ask for it, instead of taking it. By restricting access to attributes, you prevent other programmers from writing highly coupled code. When code is highly coupled, a change in one part of the code forces you to make a change in another, and then another, and so on.

# Inheritance

Similarities often exist between different classes. Two or more classes often share the same attributes and/or the same methods. Because you do not want to have to write the same code repeatedly, you want a mechanism that takes advantage of these similarities. Inheritance is that mechanism. Inheritance models "is a", "is kind of", and "is like" relationships, enabling you to reuse existing data and code easily from the existing class.

Inheritance is a technique for creating a new class (subclass) from an existing class (super class) by adding more functionality to it. We say that the new class inherits all the functionality from the existing class.

* A subclass is derived from a super class. Example: An Employee is a Person.
* The subclass inherits the attributes and behavior of the superclass.
* The subclass can override the behavior of the superclass.
* Inheritance supports code re-use.

## Inheritance Tips and Techniques

The following tips and techniques should help you to apply inheritance effectively.

1. **Look for similarities.** Whenever you have similarities between or among two or more classes, either similar attributes or similar methods, then you probably have an opportunity for inheritance.
2. **Look for existing classes.** When you identify a new class, you might already have an existing class to which it is similar. Sometimes you can directly inherit from an existing class, and just code the differences of the new class. For example, assume your university information system also needed to support university administrators. The *Person* class already has many of the features an *Administrator* class needs so you should consider having *Administrator* inherit from *Person*.
3. **Follow the sentence rule.** One of the following sentences should make sense: "A subclass *is a kind of* superclass" or "A subclass *is like a* superclass." For example, it makes sense to say *a student is a kind of person* and a dragon is like a bird. It does not make sense to say a student is a kind of vehicle or is like a vehicle, so the class *Student* likely should not inherit from *Vehicle*.
4. **Inherit everything.** The subclass should inherit everything from the super- class, a concept called pure inheritance. If it does not, the code becomes harder to understand and maintain. For example, say Class *B* inherits from *A*. To understand *B*, you need to understand what *A* is all about, plus all the features *B* adds on. If you start removing functionality, you also need to understand what *B* **is not**. This is a lot of work and becomes a maintenance nightmare.

## Single and Multiple Inheritance

When a class inherits from only one other class, we call this single inheritance. When a class inherits from two or more other classes, we call this multiple inheritance. Remember this: the subclass inherits all the attributes and methods of its superclass (es).

Not all languages support multiple inheritances. C++ is one of the few languages that do, whereas languages such as Java, Smalltalk, and C# do not. The point to be made is if your target implementation language does not support multiple inheritances, then you should not use it when you are modeling.

# Relationships

In the real world, objects have relationships with other objects. The relationships between objects are important because they help us to define how they interact with each other. For example, students *take* courses, professors *teach* courses, criminals *rob* banks, and captains *command* starships. Take, teach, rob, and command are all **verbs** that define **associations** between objects. You want to identify and potentially document these **relationships**; therefore, you can gain a better understanding as to how objects interact with one another.

Not only must you identify what the relationship(s) are between classes, you must also describe the relationship. For example, it is not enough to know that students take seminars. How many seminars can students take? None, one, or several? Furthermore, relationships are two-way streets: not only do students take seminars, but also seminars are taken by students. This leads to questions like how many students can be enrolled in any given seminar and is it possible to have a seminar with no one in it?

# Collaboration

Classes often need to work together to fulfill their responsibilities. Actually, it is typically the objects and the instances of the classes that are working together. Collaboration occurs between objects when one object asks another for information or to do something. For example, an airplane collaborates with its engines to fly. For the plane to go faster, the engines must go faster. When the plane needs to slow down, the engines must slow down. If the airplane did not collaborate with its engines, it would be unable to fly.

Objects collaborate with one another by sending each other message. A message is either a request to do something or a request for information. Messages are modeled in UML sequence diagrams and UML communication diagrams (formerly called collaboration diagrams in UML) Figure 1.14 depicts a simple sequence. You see how a student object requests to be enrolled in a seminar; the seminar object, in turn, sends a message to the course object to which it is associated because it needs to know whether the student object is qualified to enroll in the course (for example, the student has the prerequisites for the course). Figure 1.15 shows the same example as a communication diagram.

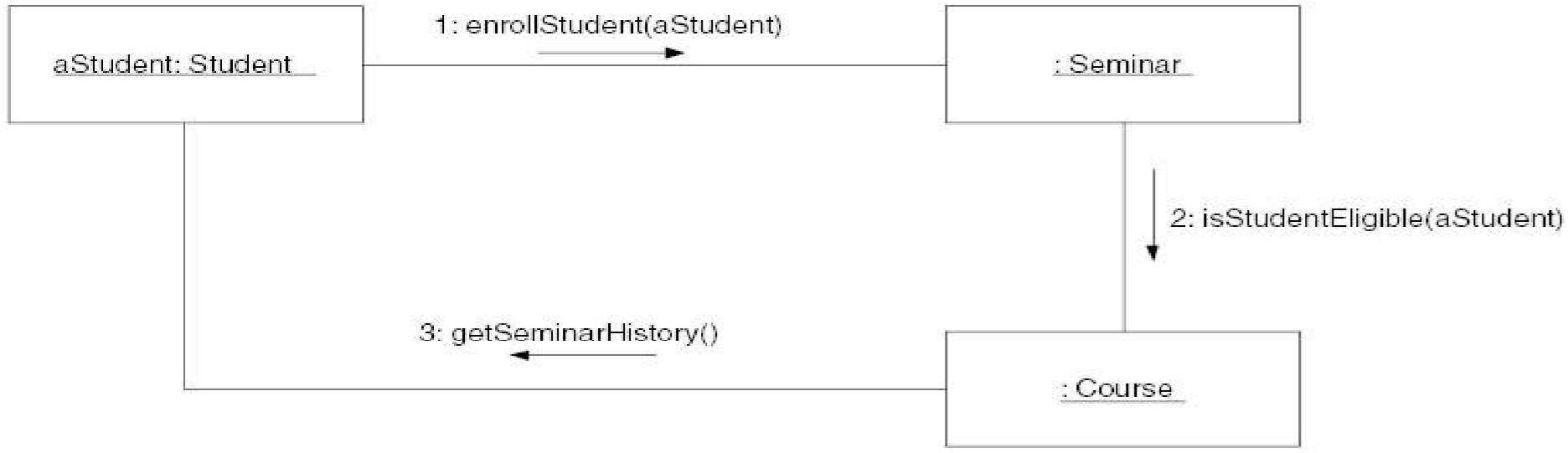


Figure 1.15: A UML communication diagram depicting messaging.

The boxes across the top of the diagram represent classifiers, in this case objects. The dashed lines hanging from them are called lifelines, which represent the life span of the object during the scenario being modeled. Objects have labels in the format *name: class* where *name* is optional (objects that have not been given a name on the diagram are called anonymous objects). The instance of *Student* was given a name because it is used as a parameter in a message, whereas the instances of *Seminar* and *Course* did not need to be referenced anywhere else in the diagram and, thus, could be anonymous. Messages are indicated as labeled arrows, the label being the signature of the method. Return values are optionally indicated using a dashed arrow with a label indicating the return value.

# . Interfaces

An interface is the definition of a collection of one or more methods, and zero or more attributes. Interfaces ideally define a cohesive set of behaviors. Interfaces are implemented by classes and components. To implement an interface, a class or component must include the methods defined by the interface. Any given class or component may implement zero or more interfaces, and one or more classes or components can implement any given interface. Interfaces are used to promote consistency within your models and source code.

# Patterns

Doesn't it always seem as if you are solving the same problems repeatedly? If you personally have not solved a given problem before, then chances are pretty good you could find somebody who had tackled the same or, at least, a similar problem in the past. Sometimes the problem you are working on is simple, sometimes it is complex, but usually it has been worked on before. Wouldn't it be nice to be able to find a solution easily, or at least a partial solution, to your problem? Think how much time and effort could be saved if you had access to a library of solutions to common system development problems. This is what patterns are all about.

A pattern is a solution to a common problem taking relevant forces into account, effectively supporting the reuse of proven techniques and approaches of other developers. Several flavors of patterns exist, including analysis patterns, design patterns, and process patterns. Analysis patterns describe a solution to common problems found in the analysis/business domain of an application, design patterns describe a solution to common problems found in the design of systems, and process patterns address software process- related issues.