Module 4

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

It is recommended that you prioritize the readings: studying the "primary" ones first and then looking at as many of the "secondary" ones as you can. Unless otherwise noted, readings are from Systems Analysis and Design: An Object-Oriented Approach with UML, by Dennis, Wixom, & Tegarden (6th Edition, 2021). In the readings listed below, you are given a page range for the reading, but you are only required to read the subsections that are itemized. If no subsections are mentioned, you are required to read the entire page range.

Module 4 Study Guide and Deliverables

Readings:

Primary Reading for Module 4:

The following readings should be completed after the module parts that they pertain to.

- Pages 32-39: Basic Characteristics of Object-Oriented Systems
- Pages 169-173: Structural Modeling: (read "Introduction," "Structural Models," and "Object Identification"—through "Textual Analysis" only)
- Pages 176-190: Class Diagrams
- Pages 215-224: Sequence Diagrams

Secondary Reading for Module 4

The following readings are not required, however they provide additional depth and examples for concepts in this module.

- Pages 224-229: Examples of Building Sequence Diagrams
- Pages 373: 2nd Paragraph on Stereotypes

Supplementary Reference

- OMG Unified Modeling Language Specification (opens PDF) from OMG v 2.5

Assignments:

- Draft Assignment 4: Part 2 of Term Project due Sunday, February 11 at 6:00 am ET

	• Assignment 4: Part 2 of Term Project due Thursday, February 15 at 6:00 am ET
Live Classroom:	• Tuesday, February 6 from 8:00-10:00 pm ET - Class Lecture • Wednesday, February 7 from 8:00-9:00 pm ET - Assignment Preview • Live Office: Saturday, February 10 from 1:00-2:00 pm ET

Module 4 Objectives

Learning Objectives

By reading the lectures and completing the assignments, you will be able to:

- Specify classes attributes, and methods in Object-Orientation
- Relate classes to each other
- Relate models to code that implements them
- Create detailed sequence diagrams
- Integrate UML component notation
- Determine when to apply what type of model

Learning Topics

The lecture discusses the issues related to UML modeling. We will cover the following topics:

- Classes in UML
    - Entity Classes, Attributes, Methods, Non-Entity Classes
- Textual Analysis strategies to determine objects
    - Example of extracting classes, attributes and methods from a use case
- Class relationships in UML
    - Inheritance, Association, Aggregation, Composition, Other Dependencies
- Class diagrams
- Detailed sequence diagrams

Appendix

- Object=Oriented Design Review

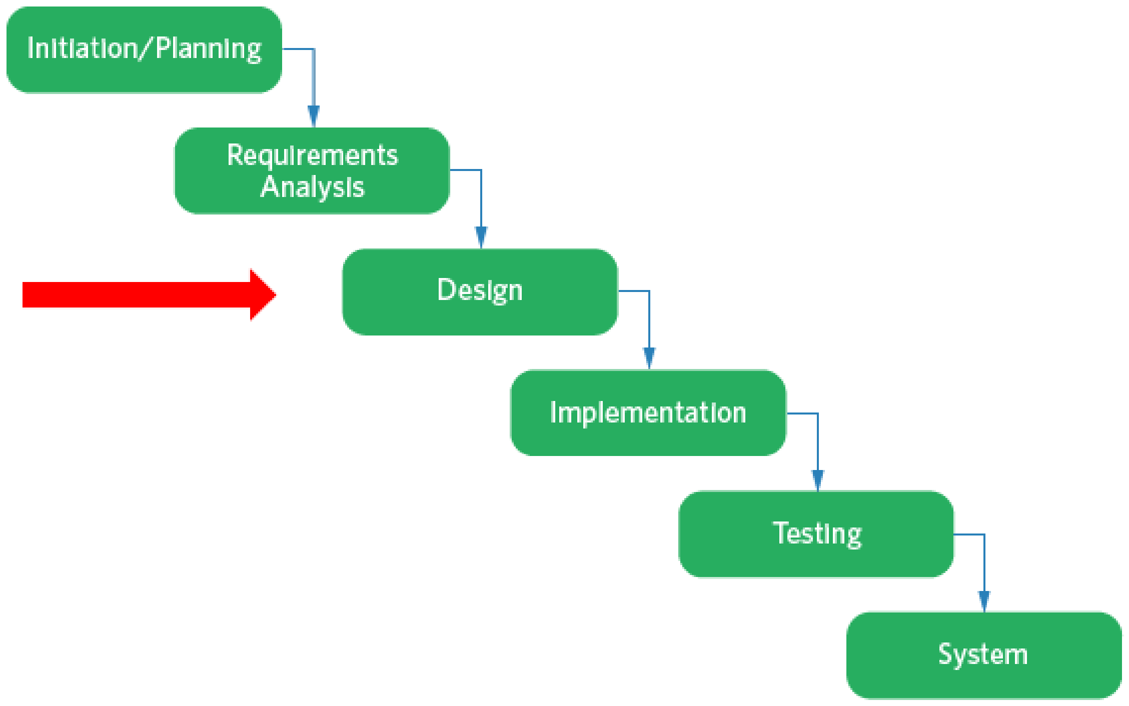
- Examples of determining classes, attributes and methods for *quickMessage*
- Examples of implementation of class relationships in Java
- *quickMessage* Class and Sequence design examples

Part I: Introduction to UML

Introduction

In Module 1 and Module 3 you learned the process of requirements analysis—you gained the tools needed to understand what is wanted or needed in an application. In Module 2 you learned the Software Development Life Cycle (SDLC) and the phases of the SDLC. Now that you have gained an understanding of requirements and put together a specification of the requirements for an application, it is time to move to the next phase, the Design phase. The design phase specifies *how* the system will be put together: the parts of the system, their responsibilities, and how the parts fit together.

You may find it helpful to review the Module 2 topic "Distinguishing Between Requirements and Design."



Waterfall Development Methodology, based on Figure 1-2 in (Dennis, Wixom, & Tegarden, 2021)

Types of Models

In Module 2 you learned various ways that the SDLC process is approached, including Waterfall, Rapid Application Development, and Agile. No matter what SDLC process is chosen, a key focus in software development is to build the solution in a timely manner, with an efficient and effective use of resources that can be sustained over time. Modeling and design are used to help realize these goals (Booch, Rumbaugh, & Jacobson, 2005).

Recall the example from Module 2 where, in order to build a kitchen, blueprints were created. These blueprints represent the technical specifications of the system—a model of how the system will be constructed.

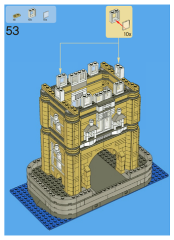
Types of Models

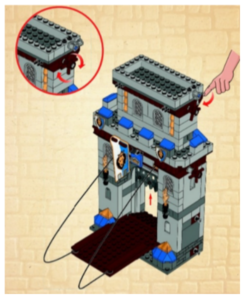
Structural Models—visualize and specify the structure of the system, its parts, and how it is to be assembled.

Behavior Models—visualize operational components of the system, and how components will operate once constructed.

Note that there are many other kinds of modeling techniques both inside and outside of software development, but within the context of this course we will focus on these two types of models.

Designs are not just for large scale and complex construction projects like an office building, or a cloud-based social media platform. If you have ever worked with a toy construction set or assembled furniture for your home, you have used models. Let's look at an example from a Lego toy construction kit that comes with detailed step by step directions—a model of how to assemble the project. Below is a page from the instructions on how to assemble a "Tower Bridge":

Here we are adding windows to one of the towers. This model shows what parts are used (top left), how to put parts together and where parts fit together. This is an example of a **structural model**. This model does not have any narrative directions on purpose because it may be used by different age groups who speak different languages across the world. However, you will notice some standards are applied, like arrows showing which parts are used in this step and how they fit together within the larger design.	 Source: Lego

 Source: Lego	Continuing with the Lego example let's look at how models demonstrate operations of the system after assembly—this is a **behavior model**. Continuing with our example, the adjacent behavior model shows how the drawbridge will operate in the "King's Kastle" Lego set.

Importance of Design and Modeling

"We build models of complex systems because we cannot comprehend such a system in its entirety"
(Booch, Rumbaugh, & Jacobson, 2005).

- **Visualize** desired structure and behavior of the system.
- **Decompose** technical details into comprehendible parts.
- Gain deeper **understanding** of technical details that may be hard to grasp.
- **Communicate** desired structure and behavior of the system to system builders.
- Strive to realize **Design goals** such as:
  - Sufficiency (build what was requested)
  - Reusability (in another project)
  - Flexibility (make modifications easily after the initial build)
- Provide system builders with a **Guide** of the desired structure and behavior that they can use during implementation.

The next three modules will explore various modeling tools, techniques, and strategies utilizing UML. The following principles of modeling will be applied as we explore these concepts. One of the learning objectives that you will want to strive for is to know when, for what, and for whom you should use different kinds of models.

Principles of Modeling

Choose what to model—focus on complex parts and decide which model will help visualize the design most effectively.

Decide who to model for—who will read it and how they will use it.

Consider different levels of detail—various models will complement each other and show how the system fits together from various points of view.

Apply standards—models need to be consistent, comprehendible and maintainable.

Unified Modeling Language (UML)

The Unified Modeling Language (UML), which we introduced you to earlier in the course, is a widely accepted graphical notation for expressing object-oriented analysis and design. Parts can be used even when the approach is not object-oriented. The official UML standard is managed by the Object Management Group, a consortium of companies, provides hundreds of pages of specifications. A link to this reference from the UML consortium can be found in Supplementary Readings, on the Study Guide page for this module.

UML can be used to show varying degrees of complexity and as such can be used to verify and document application requirements as well as support a visual representation of an application design. The level of detail and complexity should be appropriate to the anticipated audience. Today UML is an important component of the modern toolbox for systems analysis. However, our coverage of UML will be limited to its essential elements, outlined in the table below.

Please see Appendix A if you need a refresher onthe object-oriented approach.

UML 2.5 Diagrams Covered within this Course

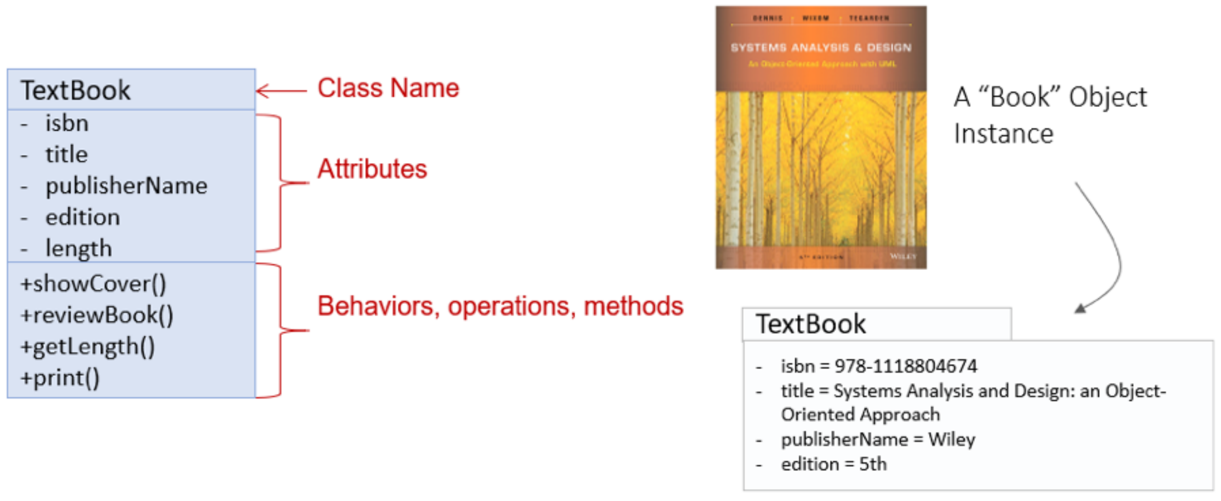
Diagram	Description	Module Introduced
Use Case Model/Narrative (Behavior Model)	Depicts interactions between the system and external systems and users. Graphically depicts users who will use the system and in what ways the user expects to interact with the system. The use-case narrative is used in addition to a textual description of the sequence of steps of each interaction.	Module 1 and Module 3
Activity (Behavior Model)	Depicts sequential flow of activities in a use case or business process at high level, or model logic within the system at a very detailed level.	Module 1 and Module 6
State Transition and State Machine (Behavior Model)	Models how events can change the state of an object (or a system) over its lifetime, showing both the various states that an object (or system) can assume and the transitions between those states.	Module 3
Sequence	Graphically depicts how objects interact with each other via messages in the execution of a	Module 4

(Behavior Model)	use case or operation. It illustrates how messages are sent and received between objects and in what *sequence*.	
Class (Structural Model)	Depicts the system's class structure. It shows the classes that the system is composed of as well as the relationships between those classes.	Module 4
Package (Structural Model)	Depicts how classes or other UML constructs are organized into packages (corresponding to Java packages) and the dependencies of those packages.	Module 5
Data Flow (Behavior & Structural Model)	Depicts the flow of data through a system and the work or processing performed by the system.	Module 5
Network Architecture (Structural Model)	A type of physical data flow diagram that allocates processors and devices to a network showing their connectivity.	Module 5
* Adapted and expanded from Whitten & Bentley, 2007.		

Classes in UML

In object-oriented sesign, the basic construct is a **class,** which is composed of structure (attributes) and behaviors.

Classes in UML diagrams are represented by rectangles containing from one to three segments depending on the desired level of detail (as we said earlier, it is important to be able to represent requirements and designs at different levels of detail). At a minimum, the rectangle contains only one segment with the class name. A more detailed version could include attribute and operation names, signatures, visibility, return types, etc. In the UML diagram below, class TextBook is represented with a rectangle with three segments: the name of the class, its attributes and behaviors.



Attributes describe data components that belong to objects of this class. In our TextBook class example above **attributes** are isbn, title, publisherName, edition and length.

Simpler UML diagrams might list attribute names only, or a more detailed diagram might list attribute data types (string, integer and so on). Although we won't focus on it in our course, in database models such as entity relational diagrams there is a similar approach where logical models show entities, attributes and their relationships, while physical models provide a bit more detail including data types.

Operations describe the behavior of objects of a class. Before we get too technical, let's consider our book example. Our TextBook class contains an attribute called Title. Our class might have two basic operations for this attribute, to update it and to retrieve it (these are referred to as setters and getters). A more advanced operation might calculate number of pages or provide a listing of key words. In other words, an operation is an action or a set of actions that an object can. Operations are also called methods or messages (see sequence diagrams later in this module). Now what is an **object**? It is a specific instance of a class. More on this a bit later, lets go back to our example, for a class called TextBook, an Object instance would be our specific textbook for the course: "Systems Analysis and Design."

In order to initiate a method (an operation) we often need to supply it with one or many parameters. For example, consider if we needed to print specifically page 3 from our textbook. We would call the TextBook class's print method, and we would specifically tell the method that we need it to print page 3. Here is a more specific TextBook example: Consider method reviewBook(integer reviewRating) which contains an argument reviewRating of type integer (Assume that our book is reviewed by a single reviewer—other implementations may handle this differently). To show that a method may take arguments, we follow the method name with open and closed parenthesis to designate where the arguments will be provided, i.e., reviewBook(). We do not need to show the internals of the parameter itself, i.e. reviewBook(integer reviewRating), as this is left to the developer. Methods can take several arguments. For example, if we created a class called Author having attributes firstName and lastName, method setAuthor(*String firstName,String lastName*) contains two arguments of type String, one for the first name, another for the second name.

The types of arguments are called the method signature. For the programmer who writes the code that calls the method it is important to know the method's signature, because modern object-oriented languages do not allow the programmer to use a value of a type different from what is expected. Method reviewBook (integer reviewRating) would expect integer reviewRating (4) and not a floating point value reviewBook (3.07) as the compiler will see a data type mismatch and reject the call.

In our reviewBook example above there are no return values—we are simply setting the attributes in our TextBook class. Other methods often return values such as getLength() which would return the length of the book.

Code outline (an outline ready to be filled with specific programming details) for this class could be written as in Appendix B. When you compare the UML class model and its code, you see that they closely approximate one another. They tell the reader (a developer in this case) that the application will implement the TextBook class. An even more detailed UML diagram would list the parameter types and the types of return values. Sometimes UML diagrams list the visibility of attributes and operations—we will discuss this later in this module.

Not all the attributes and operations need be specified in the class model. We show as much detail as needed—no more. Showing more detail clutters a diagram and can make it harder to understand and maintain when changes are made to the code itself. Some required attributes are left to the discretion of the implementers. For example, we began by introducing an operation which would set the TextBook's title and retrieve it (setTitle() and getTitle()), because most attributes will likely have getter and setter methods to set and return their values.

Class, Attribute and Method Naming Conventions

Notice the use of naming and capitalization conventions, especially for names that are composed of several component words—no spaces or special characters are used and all name components are written without spaces. For single word class names, start with an upper-case letter (i.e. Book). If a class name has multiple parts, start with an upper-case letter for both words with no spaces (i.e. TextBook). This is referred to as Upper CamelCase. For data attributes, operations, parameters and for actual arguments a similar pattern is followed where the first component starts with a lower-case letter, and other name components start with a capital letter (i.e. title, publisherName). This is referred as lower camelCase.

Summary of Entity Class, Attribute, and Method

Entity Class: a class derived from business requirements, also known as a domain class, which is composed of structure (attributes) and functionality.

- Example: TextBook, Book

Attribute: contains information about the class.

- Examples of attributes for TextBook class: isbn, title, publisherName, edition and length

Method: “functionality,” which describes the actions that an object can perform—these are interchangeably called behaviors, operations, methods, or messages within object-oriented design.

- Examples of methods for TextBook class: showCover(), reviewBook(), getLength(), and print().

In our application, when we want to track and work with a specific instance of a class, we create an **object instance** of that class. “Systems Analysis and Design” is a specific object instance of a class TextBook. In other words, think of a class as a structure, or a container, when we **instantiate** it as a specific **object instance**, we populate the attributes to track something specific, and we interact with that object through methods. Attributes distinguish a specific object from other objects of the same class. Systems Analysis and Design is a title of one specific textbook. Another textbook might have the name “Introduction to Java Programming,” which is another object instance of the class TextBook. This is similar to records in a database table, each object is like a record.

A **Class** is a set of **<u>object</u> instances** that share common attributes and behavior (methods).

- Example: TextBook class can have instances of “Systems Analysis and Design,” “Introduction to Java Programming”

Entity Classes (Domain Classes/Business Objects)

In Module 3 you learned techniques for gathering requirements. One additional technique utilizing the object-oriented approach is to gather requirements through classes. Classes derived from business requirements are known as domain classes or business objects—they may also be called **entity classes.** We have just explored this concept by looking at the TextBook class in the previous section.

Entity Class: “A class of persons, places, objects, events or concepts about which we need to capture and store data.“ (Whitten & Bentley, 2007).

- Examples: person, student, teacher, school, book, course, class, assignment, exam

A common **first step** in identifying domain/entity classes is to gather the nouns or their equivalent used in requirements. Dennis, Wixom, and Tegarden introduce this concept as textual analysis by reviewing each of the use cases. The same process can be applied to user stories and functional requirements.

Merriam-Webster defines a noun as “a word that refers to a thing (*book*), a person (*Betty Crocker*), an animal (*cat*), a place (*Omaha*)...” (Noun. (n.d.). Retrieved from https://www.merriam-webster.com/dictionary/noun).

Let’s consider the following users story and use textual analysis to determine classes:

As a student, **I want** to select a textbook **so that** I can read the material assigned in the course.

In the above user story, we have identified the following nouns**: student, textbook, material, course.**

We consider these nouns significant for two reasons. First, they describe an aspect of the business that is important for the application. Second, they denote entities that have important capabilities (attributes and methods).

The **second step** is to select significant nouns by eliminating duplicates as well as questionable or vague nouns that might describe less important classes, attributes, or non-entity objects (we will cover this later in this module). In the user story above “material” is a bit vague or can be another word for textbook. Another target for elimination are shallow entities—the entities that do not have data (attributes) that the application needs to keep track of. No duplicates are evident in this list as user stories are high level. When working off a use case there is a higher chance of finding shallow entities (see Appendix C - *quickMessage* example).

Guidelines for eliminating nouns to select objects

- **Is it a synonym of another object?**
  - A phrase that is similar to or represents an object already selected
- **Is it outside the scope of the system or is it a secondary/non-important object?**
- **Is it unclear or in need of focus?**
  - Look to review additional use cases to gain clarity
- **Is it an action or an attribute that describes another object?**
  - Make sure this is not an attribute or a method
- **Is it used to implement the system?**
  - Describes a window, frame, button, etc.

The **third and last step** is to inspect the list to ensure that the set of entities adequately covers the functional requirements. For example, review GUI sketches and state transition diagrams to see if any other entities may have been missed.

This approach puts more demands on the systems analysts because they have to translate the customer requirements into a set of classes, and consider attributes and behavior for each class. The advantage is a more systematic approach in translation of requirements into code.

In addition to textual analysis, Dennis, Wixom, and Tegarden introduce Brainstorming as a technique for discovering classes, which involves individuals suggesting classes by discussing scenarios they envision the system would comprise of. Such an approach could be an example of Agile methodology, which leans more on communication and collaboration over comprehensive documentation.

Besides entity classes, there are other types of classes to consider. In the context of this course, we will focus predominantly on entity classes and briefly explore UI Design and Interface Classes.

Types of Non-Entity Classes

User Interface Design Class—provides the means by which an actor interacts with the system (e.g., a window, dialog box, or screen). In the context of our course we will call these **Design classes.**

Control Class—contains application logic with a level of abstraction that prevents its being categorized as an entity class (e.g., Search).

Persistence Class—provides functionality for reading and writing to a database.

Operating Environment Class—handles operating system-specific functionality.

Interface Classes—handles communication with other systems (these are sometimes called application program interfaces API). Often defined as a collection of operations used to specify a service of a class or a component. (Booch, Rumbraugh, Jacobson, 2014)

* Above list is based on and adapted from Whitten and Bentley, Dennis et al., and Booch et al.

Organizing Classes Using a Table

Once we have a set of entities, we look to construct a table organized by class, one that can be directly mapped to the code later. Dennis, Wixom, and Tegarden provide a similar approach called CRC cards (Class Responsibility and Collaboration), which are outlined in Chapters 5 and 8.

Let's organize our TextBook class in the following table—we will include the classes, their attributes, and their methods, as well as their relationships to other classes (we will review relationships later in this module).

Class Name	Attributes	Functionality - Methods	Relationships
TextBook	-isbn a unique alpha	+showCover() returns image of the textbook cover.	Has **Aggregation** relationship to Library as TextBook can be part of other

	numeric value for each textbook
	
	-title
	the title of the textbook, a string
	
	-publisherName
	publisher, a string
		+reviewBook()
		will take an integer parameter 1 through 5 and set the book review. Does not return any values.
		
		+getLength()
		will take a string parameter of ISBN and return textbook length.
		
		+print()
		takes in parameters for printing and outputs selected pages in PDF format for printing.
			classes but is also part of Library. Multiplicity is one of many textbooks being part of one Library. (Aggregation is explained in the Relationship section of this module.)

Attributes (object data)

The “attributes” section describes (in complete detail) the properties of the entities in question that the application must possess. Here we have outlined example data and data types we might consider for the TextBook class.

Functionality (Behavior)

The “functionality” section describes what the application does as applied to the entities. It is important to remember that all behavior that class objects provide is not only for their own benefit, but in response to messages from other objects. For example, another class may call upon the TextBook class to set the review of the book—by calling reviewBook() method from the TextBook class with the ISBN as a parameter.

As mentioned previously, not all the attributes and operations need be specified in the class model. **We show as much detail as is needed and or relevant to communicate the design**. Showing more detail clutters a diagram and can make it harder to understand. Some required attributes are left to the discretion of the implementers. In the class design of the TextBook class we chose not to show setTitle() as it is already an attribute, and we should probably remove getLength(), as it’s not good practice to include it and leave the setting and getting methods to the developers when they implement the class.

Modeling Classes through UML

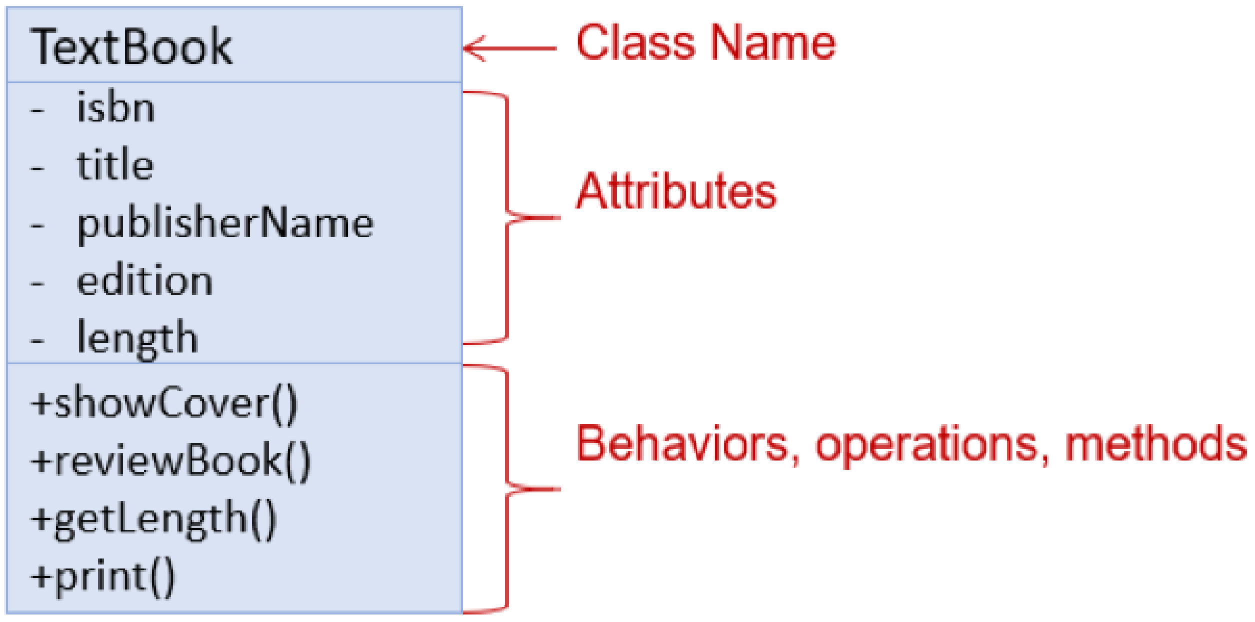
Class Diagram

The textbook example that was introduced earlier is an example of a Class within Unified Modeling Language's (UML) **Class Diagram.** A class diagram is an example of a structural model: we will use Class Models to visualize and specify the structure of the system, its parts, and how it is to be assembled.

Class Diagram—graphical model of a system's static object structure showing **object** classes (their attributes and methods) that the system is composed of as well as the **relationships** between those object classes.

Adapted from Whitten and Bentley, 2007.

Let's begin by reviewing how to model a single class specific to its attributes and methods. We will then move to class relationships. Below is a representation of the TextBook class.



Attribute and Method Visibility

So far, we have introduced you to how to determine entity classes and represent these within a class model using a class name, attributes and methods. Let's review **Visibility** that is shown within the class model. Visibility relates to the level of information hidden within the object. In object-oriented design this is often referred to as part of **Encapsulation.** Dennis et al. define encapsulation as a combination of the processes and data contained within a class. Some of this is visible to other classes, so that other classes can use it (call it), and some parts are hidden. In the example above, the reviewBook() method is public, meaning that we can call this method from other classes to perform an operation. However, the internals of this method as well as the internals of TextBook class attributes such as the title are hidden. This is also referred to as **Information Hiding**. This is a good thing! In object-oriented design it allows developers to worry only about what is needed to perform the task at hand and improves reusability—a topic which we will discuss later in the course. Reusability is quite prevalent in modern programming. For example, if you use Python for data analysis, you will often incorporate Pandas (Python Data Analysis Library) so that you don't have to start creating data manipulation algorithms from scratch. A good analogy to this is if you are making a meal, you will use a stove instead of re-creating fire manually every single time. You don't need to understand how the stove works internally, just how to turn it on and off and how to control the level of heat.

To understand visibility a little bit better, consider how you might order a pizza. You might place an order (with specific parameters such as extra cheese and peperoni) and pick it up when it is ready for you (encapsulated) in a pizza box, but you are not going to worry about all the ingredients and detailed steps of how the pizza is prepared in the kitchen. You can start to see the paradox here: sometimes it is helpful to know the internals if you need to make changes. This is an important topic, and something we will discuss later in the course—flexibility. Visibility is denoted in the diagram next to the attributes and methods with a minus ("–"), which stands for "private" or hidden from all other classes. A plus ("+") stands for public, meaning that, in this case, a method can be called by other classes. Note that although a method may be denoted as public, its internals can still be private to other classes. In Java or C++ code, methods or variables of a class have additional levels of visibility. These are called member access modifiers and include public, protected, private.
the code outline for the TextBook class. When you compare the UML class model and its code, you see that they closely approximate one another.

Please Review Appendix B for the code outline for the TextBook class. When you compare the UML class model and its code, you see that they closely approximate one another.

Class Relationships

Once we have identified our classes and their internals we need to think about the relationships between these classes. These relationships will help us think of how classes collaborate and connect with one another, so that the developer can fully see how the design fits together structurally.

In a class diagram, relationships are represented with different kinds of lines. The three categories of relationships that we will focus on in this course are outlined below:

Class Relationships

- **Inheritance-Generalization-Specialization**
    - One class is **a kind of** another class: Superclasses & Subclasses
        - Example: People superclass, Student & Teacher subclasses
- **Aggregation-Composition-Association**
    - One class is **part of/has** another class
        - Example: Book is part of a Library
- **Dependencies**
    - One class **uses** another class
        - Example: Search depends on (uses) Book

Inheritance

Inheritance is a relationship between classes; we use it to idicate that these classes have common attributes and/or operations.

Let’s consider an example where we wanted to create a class model for students and teachers for a university. When we think about students and faculty, we know that they share the attributes such as name and date of birth, as well as behavior such as talk and listen. You can define these application business concepts (student and faculty) as separate and independent entities. This might result in duplicating some of the code (similar attributes “firstName” and “lastName” in both classes). There are various disadvantages to duplicated code, such as complexity in maintaining it. Another developer might have a hard time understanding and modifying the design and code of the system later. Also, this is really a simple example, and in real life, classes might have many attributes, making it hard to notice and evaluate the relationships. This is especially true if the same attribute gets different names in different classes—such as “User” in one class and “Person” in another.

We can then **Generalize** this concept as a Person class, and Student and Faculty are **Specializations** of the Person class. These specialization classes would contain attributes and methods that are specific to these entity classes. To make the relationships immediately apparent, you can view class Person as a “base” class (or superclass) and define Student as a “derived” class (subclass) of class Person. We say that Student inherits from (“extends,” in Java parlance) class Person.

Defining class Student as extending Person, you will specify only the attributes that are added by the subclass to the superclass attributes (e.g., GPA). There is no need to explicitly define first and last name attributes as these are inherited from the Person superclass.

The same is true about operations (methods or messages). If class Person has method talk(), there is no need to define talk() for Student as well; this method is inherited from the superclass. Thus, using inheritance significantly simplifies the description of subclasses. You need to define only those capabilities (data and operations) that are added in the subclass. There is no need to repeat attributes and methods inherited from the superclass (actually, it would be a mistake to repeat attributes and operation definitions in both classes). There is one exception to this: if in the derived class the method inherited from the superclass needs to be adjusted in some way. This is called **overriding**. For example, let's say that talk() method in Person class is generalized—however in the Student subclass we need to make the distinction that Student can only talk when instructor asks them to, We thus override the logic of the talk() method by outlining the changes to it in the subclass Student.

Very often, one class is used as the superclass for more than one subclass. In our example, Student will have attributes such as a GPA and a level, such as graduate or undergraduate, and behaviors such as enroll() and learn(). Faculty on the other hand will have attributes such as role (i.e., lecturer, teaching assistant), area of concentration, and ranking as well as behaviors such as teach() and gradeAssignment(). One way to think about inheritance is that Student and Faculty are **a kind of** Person. When deciding on what goes where within inherited classes, consider what are the similarities and differences. This can also be referred to as the level of **cohesion** —how well things fit together. Similarities will be part of generalized classes, while differences or unique elements will be part of specialized classes and inherit from the generalized class.

Inheritance relationships can be organized into hierarchies. For example, we could continue to specialize Faculty into subclasses, decomposing further into Lecturer, Researcher, and Teaching Assistant, which would give us a more granular way to represent objects.

The added benefit of a design with inheritance is that subclass objects may be used in a wide set of contexts. For example, consider creating a banking application with various types of accounts, such as Savings, Checking and Investment, which may share similar attributes and behavior yet also show specialization within each type of account.

In the diagram above Student and Faculty are "derived" **subclasses** that **Inherit** (incorporate) all the attributes and methods of the Person "base" **superclass**.

In a UML class diagram, it is customary to arrange class models on class diagrams so that base classes are physically placed above derived classes on the diagram, with the inheritance indicated by lines from the subclasses converging on a single superclass with a hollow arrowhead. However, this positioning is not necessary in any technical sense. Abstract classes, i.e., those that cannot be instantiated into objects, are denoted with italics. Please review Appendix D for a Java example of implementing the design below.

Association

Booch et al define **Association** as a structural relationship specifying that one class is connected to another in a structural manner. Associations typically represent natural business associations between classes. We can think of associations when one class **has** (incorporates) another class. Association, denoted with a solid line between two classes, commonly means that objects of each class are somehow associated with objects of the other class in a structural manner. We can annotate the relationship, which may be one- or two-way. This is illustrated below.

Associations are useful in the early stages of building class model relationships, when we know a pair of objects to be related but are postponing the ultimate specification of the relationship.

Whenever it makes logical sense, we should replace associations and dependencies with aggregations and compositions.

One-way associations are aggregations and compositions, which will be covered next. Two-way associations are problematical because we need to be sure that both ends of the implied information are kept consistent when two objects try to modify each other.

For now, let’s consider our earlier example of a book class and expand on it, by modeling a class diagram for a library, adding a Library and a Librarian class.

Natural business associations that we could consider here would be:

- Library **has** one or many books
- Library **has** one or many Librarians

Our design using Association represented with a solid line between two classes.

Next, we need to consider **Multiplicity** to represent the business associations of the relationship. Whitten et al define multiplicity as the “minimum and maximum number of occurrences of one object class for a single occurrence of the related object class” (2017). Using the business associations above we can define multiplicity for our associative relationships. Let’s expand and add two more natural business associations to the list above:

- Library **has** one or many books
- Library **has** one to three Librarians
- Library **has** one and only one Catalog
- Library **has** zero or many meeting spaces

The chart below summarizes multiplicity notations.

Type	Example	Notes
Exactly 1		• Multiplicity is depicted by 1 • If multiplicity is not shown – 1:1 multiplicity is implied
One or more		• 1 to Many books is depicted by 1..* • A specific range can be defined; in this example a Library has at least 1 or as many as 3 librarians
One or more		• 0 to Many books is depicted by 0..* • In this example, a Library may not have any meeting spaces

Please see Appendix H for a comparison of representing many to many relationships in Object-Oriented design vs. Relational Database Design.

Putting these all together, our design of a Library application is beginning to take shape.

As stated earlier associations are useful in the early stages of building class model relationships, when we know a pair of objects to be related but are postponing the ultimate specification of the relationship, as in our Library example above.

However, we should look to replace associations with aggregation, composition, or dependency.

In translating our models to object-oriented code such as Java, an association is usually implemented as an **aggregation**, **composition,** or **dependency**. Once you have reviewed the relationship types, please see the Java implementation examples in the appendices.

Aggregation

Aggregation indicates the structural inclusion of objects of one class in an object of another class—in other words, one object **is part of** another object. On UML diagrams, this relationship is denoted with a hollow diamond. It is usually implemented by means of a containing class (the aggregate) having an attribute whose type is the included (aggregated) class. See the diagram below, where there is a book attribute within the Library class designating the aggregation relationship of Book to Library. Unlike inheritance, this is a relationship among objects, not among classes. The aggregate can be very loosely thought of as the “whole” and the class aggregated as the “part” or as a component. Aggregation is shown below where we can be more semantically accurate and say that a book is **part of** a library. The key point in this design is that the Library class has an attribute for book and when a Library object is created, we also create a book object. This is an important element of showing a “part of” relationship. Please see Appendix E for a Java implementation of aggregation design.

Let’s review Multiplicity within this design: the numeral at the end of an aggregation arrow denotes number of objects aggregated. For example, the “1” at the end of the *Book/Library* relationship implies that each *Book* object aggregates exactly one *Library* object. Instead of single numerals, a range can be given, such as *3..7*. The symbol “*” denotes an undetermined number of Book objects aggregated.

In Aggregation relationships, the object that is part of another object (or the aggregated object) can exist on its own. Moreover, the same object could be a component of several aggregates of the same class or even of different classes. In the example below a Book object could be a component of several ReadingList objects (i.e., Summer, Top 10, etc) and of a Library object. This means that aggregate membership is not exclusive and a component object could be shared by several aggregate objects. We can say that a Book can be part of a Library or it can be part of a ReadingList in the system without the need for a Library.

Unlike composition, which is discussed next, if the aggregate object (e.g., a ReadingList) is eliminated during program execution, it might or might not entail elimination of aggregated objects (e.g,, Book objects) because these objects might exist independently of the aggregate object as part of the Library object.

Aggregations, like inheritance, can be hierarchical. In the example below, we introduce a class called BookShelf. We can say that Books are part of a BookShelf and a BookShelf is part of a Library.

Boston University Metropolitan College