

1

Object-oriented Design

In software development, design is often considered the step done **before** programming. This isn't true; in reality, analysis, programming, and design tend to overlap, combine, and interweave. In this chapter, we will learn:

- What object-oriented means
- The difference between object-oriented design and object-oriented programming
- The basic principles of object-oriented design
- Basic Unified Modeling Language and when it isn't evil

Object-oriented?

Everyone knows what an object is: a tangible "something" that we can sense, feel, and manipulate. The earliest objects we interact with are typically baby toys. Wooden blocks, plastic shapes, and over-sized puzzle pieces are common first objects. Babies learn quickly that certain objects do certain things. Triangles fit in triangle-shaped holes. Bells ring, buttons press, and levers pull.

The definition of an object in software development is not so very different. Objects are not typically *tangible somethings* that you can pick up, sense, or feel, but they are models of *somethings* that can do certain things and have certain things done to them. Formally, an object is a collection of **data** and associated **behaviors**.

So knowing what an object is, what does it mean to be object-oriented? Oriented simply means directed toward. So object-oriented simply means, "functionally directed toward modeling objects". It is one of many techniques used for modeling complex systems by describing a collection of interacting objects via their data and behavior.

If you've read any hype, you've probably come across the terms object-oriented analysis, object-oriented design, object-oriented analysis and design, and object-oriented programming. These are all highly related concepts under the general object-oriented umbrella.

In fact, analysis, design, and programming are all stages of software development. Calling them object-oriented simply specifies what style of software development is being pursued.

Object-oriented Analysis (OOA) is the process of looking at a problem, system, or task that somebody wants to turn into an application and identifying the objects and interactions between those objects. The analysis stage is all about *what* needs to be done. The output of the analysis stage is a set of requirements. If we were to complete the analysis stage in one step, we would have turned a task, such as, "I need a website", into a set of requirements, such as:

Visitors to the website need to be able to (*italic* represents actions, **bold** represents objects):

- *review* our **history**
- *apply* for **jobs**
- *browse, compare, and order* our **products**

Object-oriented Design (OOD) is the process of converting such requirements into an implementation specification. The designer must name the objects, define the behaviors, and formally specify what objects can activate specific behaviors on other objects. The design stage is all about *how* things should be done. The output of the design stage is an implementation specification. If we were to complete the design stage in one step, we would have turned the requirements into a set of classes and interfaces that could be implemented in (ideally) any object-oriented programming language.

Object-oriented Programming (OOP) is the process of converting this perfectly defined design into a working program that does exactly what the CEO originally requested.

Yeah, right! It would be lovely if the world met this ideal and we could follow these stages one by one, in perfect order like all the old textbooks told us to. As usual, the real world is much murkier. No matter how hard we try to separate these stages, we'll always find things that need further analysis while we're designing. When we're programming, we find features that need clarification in the design. In the fast-paced modern world, most development happens in an **iterative development model**. In iterative development, a small part of the task is modeled, designed, and programmed, then the program is reviewed and expanded to improve each feature and include new features in a series of short cycles.

The rest of this book is about object-oriented programming, but in this chapter we will cover the basic object-oriented principles in the context of design. This allows us to understand these rather simple concepts without having to argue with software syntax or interpreters.

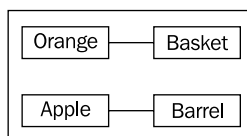
Objects and classes

So, an object is a collection of data with associated behaviors. How do we tell two types of objects apart? Apples and oranges are both objects, but it is a common adage that they cannot be compared. Apples and oranges aren't modeled very often in computer programming, but let's pretend we're doing an inventory application for a fruit farm! As an example, we can assume that apples go in barrels and oranges go in baskets.

Now, we have four kinds of objects: apples, oranges, baskets, and barrels. In object-oriented modeling, the term used for kinds of objects is **class**. So, in technical terms, we now have four **classes** of objects.

What's the difference between an object and a class? Classes describe objects. They are like blueprints for creating an object. You might have three oranges sitting on the table in front of you. Each orange is a distinct object, but all three have the attributes and behaviors associated with one class: the general class of oranges.

The relationship between the four classes of objects in our inventory system can be described using a **Unified Modeling Language** (invariably referred to as **UML**, because three letter acronyms are *cool*) **class diagram**. Here is our first class diagram:



This diagram simply shows that an **Orange** is somehow associated with a **Basket** and that an **Apple** is also somehow associated with a **Barrel**. **Association** is the most basic way for two classes to be related.

UML is very popular among managers, and occasionally disparaged by programmers. The syntax of a UML diagram is generally pretty obvious; you don't have to read a tutorial to (mostly) understand what is going on when you see one. UML is also fairly easy to draw, and quite intuitive. After all, many people, when describing classes and their relationships, will naturally draw boxes with lines between them. Having a standard based on these intuitive diagrams makes it easy for programmers to communicate with designers, managers, and each other.

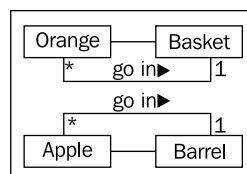
However, some programmers think UML is a waste of time. Citing iterative development, they will argue that formal specifications done up in fancy UML diagrams are going to be redundant before they're implemented, and that maintaining those formal diagrams will only waste time and not benefit anyone.

This is true of some organizations, and hogwash in other corporate cultures. However, every programming team consisting of more than one person will occasionally have to sit down and hash out the details of part of the system they are currently working on. UML is extremely useful, in these brainstorming sessions, for quick and easy communication. Even those organizations that scoff at formal class diagrams tend to use some informal version of UML in their design meetings, or team discussions.

Further, the most important person you ever have to communicate with is yourself. We all think we can remember the design decisions we've made, but there are always, "Why did I do that?" moments hiding in our future. If we keep the scraps of paper we did our initial diagramming on when we started a design, we'll eventually find that they are a useful reference.

This chapter, however, is not meant to be a tutorial in UML. There are many of those available on the Internet, as well as numerous books available on the topic. UML covers far more than class and object diagrams; it also has syntax for use cases, deployment, state changes, and activities. We'll be dealing with some common class diagram syntax in this discussion of object-oriented design. You'll find you can pick up the structure by example, and you'll subconsciously choose UML-inspired syntax in your own team or personal design sessions.

Our initial diagram, while correct, does not remind us that apples go in barrels or how many barrels a single apple can go in. It only tells us that apples are somehow associated with barrels. The association between classes is often obvious and needs no further explanation, but the option to add further clarification is always there. The beauty of UML is that most things are optional. We only need to specify as much information in a diagram as makes sense for the current situation. In a quick whiteboard session, we might just quickly draw lines between boxes. In a formal document that needs to make sense in six months, we might go into more detail. In the case of apples and barrels, we can be fairly confident that the association is, "many apples go in one barrel", but just to make sure nobody confuses it with, "one apple spoils one barrel", we can enhance the diagram as shown:



This diagram tells us that oranges **go in** baskets with a little arrow showing what goes in what. It also tells us the **multiplicity** (number of that object that can be used in the association) on both sides of the relationship. One **Basket** can hold many (represented by a *) **Orange** objects. Any one **Orange** can go in exactly one **Basket**.

It can be easy to confuse which side of a relationship the multiplicity goes on. The multiplicity is the number of objects of **that** class that can be associated with **any one** object at the other end of the association. For the apple goes in barrel association, reading from left to right, many instances of the **Apple** class (that is many **Apple** objects) can **go in** any one **Barrel**. Reading from right to left, **exactly one Barrel** can be associated with any one **Apple**.

Specifying attributes and behaviors

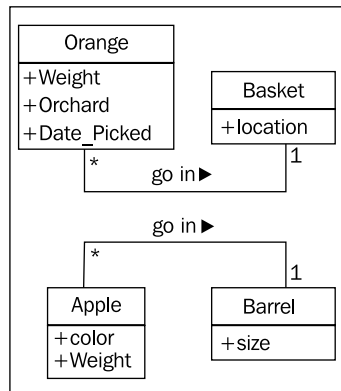
We now have a grasp on some basic object-oriented terminology. Objects are **instances** of classes that can be associated with each other. An object instance is a specific object with its own set of data and behaviors; a specific orange on the table in front of us is said to be an instance of the general class of oranges. That's simple enough, but what are these data and behaviors that are associated with each object?

Data describes objects

Let's start with data. Data typically represents the individual characteristics of a certain object. A class of objects can define specific characteristics that are shared by all instances of that class. Each instance can then have different data values for the given characteristics. For example, our three oranges on the table (if we haven't eaten any) could each have a different weight. The orange class could then have a weight **attribute**. All instances of the orange class have a weight attribute, but each orange might have a different value for this weight. Attributes don't have to be unique though; any two oranges may weigh the same amount. As a more realistic example, two objects representing different customers might have the same value for a first name attribute.

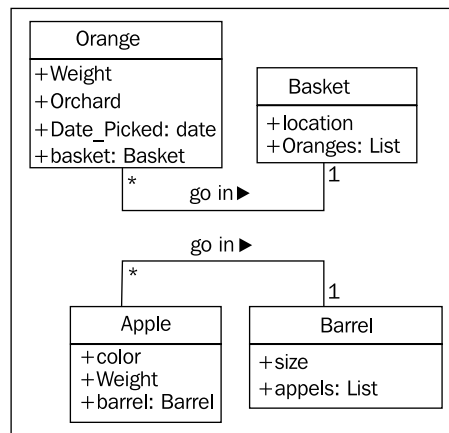
Attributes are frequently referred to as **properties**. Some authors suggest that the two terms have different meanings, usually that attributes are settable, while properties are read only. In Python, the concept of "read only" is not really used, so throughout this book we'll see the two terms used interchangeably. In addition, as we'll discuss in *Chapter 5*, the property keyword has a special meaning in Python for a particular kind of attribute.

In our fruit inventory application, the fruit farmer may want to know what orchard the orange came from, when it was picked, and how much it weighs. They might also want to keep track of where each basket is stored. Apples might have a color attribute and barrels might come in different sizes. Some of these properties may also belong to multiple classes (we may want to know when apples are picked, too), but for this first example, let's just add a few different attributes to our class diagram:



Depending on how detailed our design needs to be, we can also specify the type for each attribute. Attribute types are often primitives that are standard to most programming languages, such as integer, floating-point number, string, byte, or boolean. However, they can also represent data structures such as lists, trees, or graphs, or, most notably, other classes. This is one area where the design stage can overlap with the programming stage. The various primitives or objects available in one programming language may be somewhat different from what is available in other languages. Usually we don't need to concern ourselves with this at the design stage, as implementation-specific details are chosen during the programming stage. Use generic names and we'll be fine. If our design calls for a list container type, the Java programmers can choose to use a `LinkedList` or an `ArrayList` when implementing it, while the Python programmers (that's us!) can choose between the `list` built-in and a `tuple`.

In our fruit farming example, so far, our attributes are all basic primitives. But there are implicit attributes that we can make explicit: the associations. For a given orange, we might have an attribute containing the basket that holds that orange. Alternatively, one basket might contain a list of the oranges it holds. The next diagram adds these attributes as well as including type descriptions for our current properties:



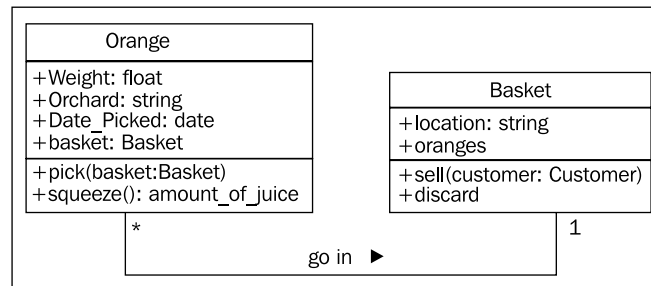
Behaviors are actions

Now we know what data is, but what are behaviors? Behaviors are actions that can occur on an object. The behaviors that can be performed on a specific class of objects are called **methods**. At the programming level, methods are like functions in structured programming, but they magically have access to all the data associated with that object. Like functions, methods can also accept **parameters**, and return **values**.

Parameters to a method are a list of objects that need to be **passed** into the method that is being called. These objects are used by the method to perform whatever behavior or task it is meant to do. Return values are the results of that task. Here's a concrete example; if our objects are numbers, the number class might have an **add** method that accepts a second number as a parameter. The first number object's **add** method will return the sum when the second number is passed to it. Given an object and its method name, a calling object can call, or **invoke** the method on the target object. Invoking a method, at the programming level, is the process of telling the method to execute itself by passing it the required parameters as arguments.

We've stretched our, "comparing apples and oranges" example into a basic (if far-fetched) inventory application. Let's stretch it a little further and see if it breaks. One action that can be associated with oranges is the **pick** action. If you think about implementation, **pick** would place the orange in a basket by updating the **basket** attribute on the orange, and by adding the orange to the **oranges** list on the **Basket**. So **pick** needs to know what basket it is dealing with. We do this by giving the **pick** method a **basket** parameter. Since our fruit farmer also sells juice, we can add a **squeeze** method to **Orange**. When squeezed, **squeeze** might return the amount of juice retrieved, while also removing the **Orange** from the **basket** it was in.

Basket can have a sell action. When a basket is sold, our inventory system might update some data on as-yet unspecified objects for accounting and profit calculations. Alternatively, our basket of oranges might go bad before we can sell them, so we add a **discard** method. Let's add these methods to our diagram:



Adding models and methods to individual objects allows us to create a **system** of interacting objects. Each object in the system is a member of a certain class. These classes specify what types of data the object can hold and what methods can be invoked on it. The data in each object can be in a different state from other objects of the same class, and each object may react to method calls differently because of the differences in state.

Object-oriented analysis and design is all about figuring out what those objects are and how they should interact. The next section describes principles that can be used to make those interactions as simple and intuitive as possible.

Hiding details and creating the public interface

The key purpose of modeling an object in object-oriented design is to determine what the public **interface** of that object will be. The interface is the collection of attributes and methods that other objects can use to interact with that object. They do not need, and are often not allowed, to access the internal workings of the object. A common real-world example is the television. Our interface to the television is the remote control. Each button on the remote control represents a method that can be called on the television object. When we, as the calling object, access these methods, we do not know or care if the television is getting its signal from an antenna, a cable connection, or a satellite dish. We don't care what electronic signals are being sent to adjust the volume, or whether that volume is being output to speakers or a set of headphones. If we open the television to access the internal workings, for example to split the output signal to both external speakers and a set of headphones, we will void the warranty.

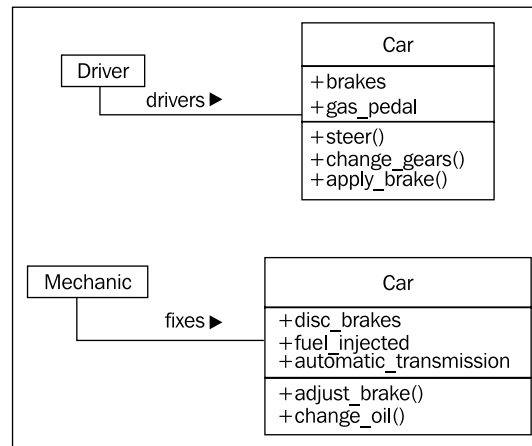
This process of hiding the implementation, or functional details, of an object is suitably called **information hiding**. It is also sometimes referred to as **encapsulation**, but encapsulation is actually a more all-encompassing term. Encapsulated data is not necessarily hidden. Encapsulation is, literally, creating a capsule, so think of creating a time capsule. If you put a bunch of information into a time capsule, lock and bury it, it is both encapsulated and the information is hidden. On the other hand, if the time capsule has not been buried and is unlocked or made of clear plastic, the items inside it are still encapsulated, but there is no information hiding.

The distinction between encapsulation and information hiding is largely irrelevant, especially at the design level. Many practical references use the terms interchangeably. As Python programmers, we don't actually have or need true information hiding, (we'll discuss the reasons for this in *Chapter 2*) so the more encompassing definition for encapsulation is suitable.

The public interface, however, is very important. It needs to be carefully designed as it is difficult to change it in the future. Changing the interface will break any client objects that are calling it. We can change the internals all we like, for example, to make it more efficient, or to access data over the network as well as locally, and the client objects will still be able to talk to it, unmodified, using the public interface. On the other hand, if we change the interface, by changing attribute names that are publicly accessed or by altering the order or types of arguments that a method can accept, all client objects will also have to be modified.

Remember, program objects represent real objects, but they are not real objects. They are models. One of the greatest gifts of modeling is the ability to ignore details that are irrelevant. A model car may look like a real 1956 Thunderbird on the outside, but it doesn't run and the driveshaft doesn't turn, as these details are overly complex and irrelevant to the youngster assembling the model. The model is an **abstraction** of a real concept.

Abstraction is another object-oriented buzzword that ties in with encapsulation and information hiding. Simply put, abstraction means dealing with the level of detail that is most appropriate to a given task. It is the process of extracting a public interface from the inner details. A driver of a car needs to interact with steering, gas pedal, and brakes. The workings of the motor, drive train, and brake subsystem don't matter to the driver. A mechanic, on the other hand works at a different level of abstraction, tuning the engine and bleeding the breaks. Here's an example of two abstraction levels for a car:



Now we have several new terms that refer to similar concepts. Condensing all this jargon into a single sentence, abstraction is the process of encapsulating information with separate public and private interfaces. The private interfaces can be subject to information hiding.

The important thing to bring from all these definitions is to make our models understandable to the other objects that have to interact with them. This means paying careful attention to small details. Ensure methods and properties have sensible names. When analyzing a system, objects typically represent nouns in the original problem, while methods are normally verbs. Attributes can often be picked up as adjectives, although if the attribute refers to another object that is part of the current object, it will still likely be a noun. Name classes, attributes, and methods accordingly. Don't try to model objects or actions that **might** be useful in the future. Model exactly those tasks that the system needs to perform and the design will naturally gravitate towards one that has an appropriate level of abstraction. This is not to say we should not think about possible future design modifications. Our designs should be open ended so that future requirements can be satisfied. However, when abstracting interfaces, try to model exactly what needs to be modeled and nothing more.

When designing the interface, try placing yourself in the object's shoes and imagine that the object has a strong preference for privacy. Don't let other objects have access to data about you unless you feel it is in your best interest for them to have it. Don't give them an interface to force you to perform a specific task unless you are certain you want them to be able to do that to you.

This is also a good practice for ensuring privacy on your social networking accounts!

Composition and inheritance

So far, we've learned to design systems as a group of interacting objects, where each interaction is viewing the objects involved at an appropriate level of abstraction. But we don't know yet how to create those levels of abstraction. There are a variety of ways to do this; we'll discuss some advanced design patterns in *Chapter 8* and *Chapter 9*. But even most design patterns rely on two basic principles known as **composition** and **inheritance**.

Composition is the act of collecting together several objects to compose a new one. Composition is usually a good choice when one object is part of another object. We've already seen a first hint of composition in the mechanic example. A car is composed of an engine, transmission, starter, headlights, and windshield, among numerous other parts. The engine, in turn, is composed of pistons, a crank shaft, and valves. In this example, composition is a good way to provide levels of abstraction. The car object can provide the interface required by a driver, while also providing access to its component parts, which offers a deeper level of abstraction suitable for a mechanic. Those component parts can, of course, be further broken down if the mechanic needs more information to diagnose a problem or tune the engine.

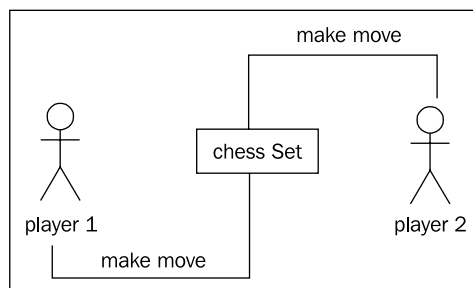
This is a common first example of composition, but it's not a very good one when it comes to designing computer systems. Physical objects are easy to break into component objects. People have been doing it at least since the ancient Greeks originally postulated that atoms were the smallest unit of matter (they, of course, didn't have access to particle accelerators). Computer systems are generally less complicated than physical objects, yet identifying the component objects in such systems does not happen as naturally. The objects in an object-oriented system occasionally represent physical objects like people, books, or telephones. More often, however, they represent abstract ideas. People have names, books have titles, and telephones are used to make calls. Calls, titles, accounts, names, appointments, and payments are not usually considered objects in the physical world, but they are all frequently modeled components in computer systems.

Let's try modeling a more computer-oriented example to see composition in action. We'll be looking at the design of a computerized chess game. This was a very popular pastime among academics in the '80s and '90s. People were predicting that computers would one day be able to defeat a human chess master. When this happened in 1997 (IBM's Deep Blue defeated world chess champion, Gary Kasparov), interest in the problem waned, although there are still contests between computer and human chess players, and the program has not yet been written that can defeat a human chess master 100% of the time.

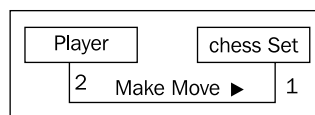
As a basic, high-level analysis: a game of chess is **played** between two *players*, using a *chess set* featuring a *board* containing sixty-four *positions* in an 8x8 grid. The board can have two sets of sixteen *pieces* that can be **moved**, in alternating *turns* by the two players in different ways. Each piece can **take** other pieces. The board will be required to **draw** itself on the computer *screen* after each turn.

I've identified some of the possible objects in the description using *italics*, and a few key methods using **bold**. This is a common first step in turning an object-oriented analysis into a design. At this point, to emphasize composition, we'll focus on the board, without worrying too much about the players or the different types of pieces.

Let's start at the highest level of abstraction possible. We have two players interacting with a chess set by taking turns making moves.



What is that? It doesn't quite look like our earlier class diagrams. That's because it isn't a class diagram! This is an **object diagram**, also called an instance diagram. It describes the system at a specific state in time, and is describing specific instances of objects, not the interaction between classes. Remember, both players are members of the same class, so the class diagram looks a little different:



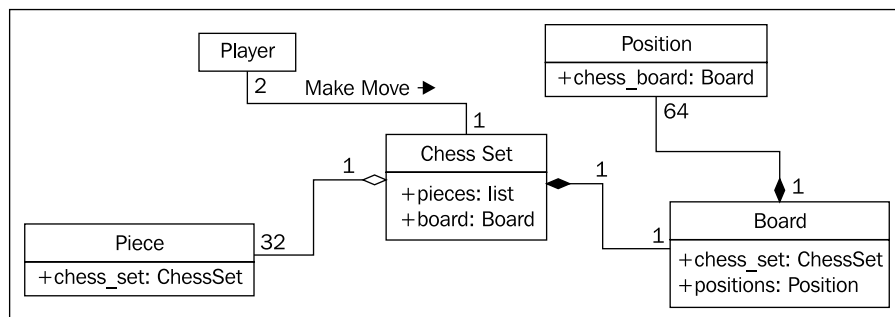
The diagram shows that exactly two players can interact with one chess set. It also indicates that any one player can be playing with only one chess set at a time.

But we're discussing composition, not UML, so let's think about what the **Chess Set** is composed of. We don't care what the player is composed of at this time. We can assume that the player has a heart and brain, among other organs, but these are irrelevant to our model. Indeed, there is nothing stopping said player from being Deep Blue itself, which has neither a heart nor brain.

The chess set, then, is composed of a board and thirty-two pieces. The board is further comprised of sixty-four positions. You could argue that pieces are not part of the chess set because you could replace the pieces in a chess set with a different set of pieces. While this is unlikely or impossible in a computerized version of chess, it introduces us to **aggregation**. Aggregation is almost exactly like composition. The difference is that aggregate objects can exist independently. It would be impossible for a position to be associated with a different chess board, so we say the board is composed of positions. But the pieces, which might exist independently of the chess set, are said to be in an aggregate relationship with that set.

Another way to differentiate between aggregation and composition is to think about the lifespan of the object. If the composite (outside) object controls when the related (inside) objects are created and destroyed, composition is most suitable. If the related object is created independently of the composite object, or can outlast that object, an aggregate relationship makes more sense. Also keep in mind that composition is aggregation; aggregation is simply a more general form of composition. Any composite relationship is also an aggregate relationship, but not vice versa.

Let's describe our current chess set composition and add some attributes to the objects to hold the composite relationships:



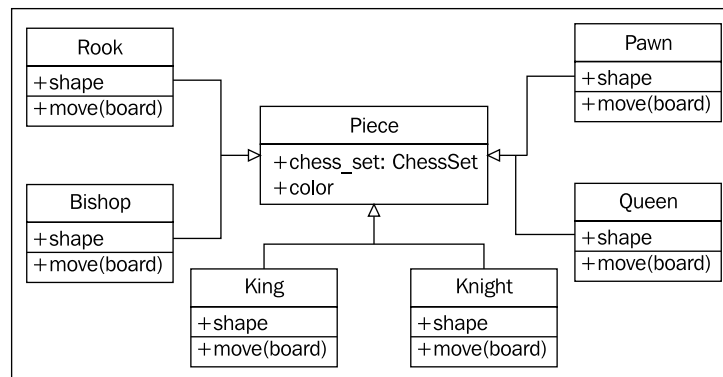
The composition relationship is represented in UML as a solid diamond. The hollow diamond represents the aggregate relationship. You'll notice that the board and pieces are stored as part of the chess set in exactly the same way a reference to them is stored as an attribute on the chess set. This shows that once again, in practice, the distinction between aggregation and composition is often irrelevant once you get past the design stage. When implemented, they behave in much the same way. However, it can help to differentiate between the two when your team is discussing how the different objects interact. Often you can treat them as the same thing, but when you need to distinguish between them, it's great to know the difference (this is abstraction at work).

Inheritance

We have discussed three types of relationships between objects: association, composition, and aggregation. But we have not fully specified our chess set, and these tools don't seem to give us all the power we need. We discussed the possibility that a player might be a human or it might be a piece of software featuring artificial intelligence. It doesn't seem right to say that a Player is *associated* with a human, or that the artificial intelligence implementation is *part of* the Player object. What we really need is the ability to say that "Deep Blue *is a* player" or that "Gary Kasparov *is a* player".

The *is a* relationship is formed by **inheritance**. Inheritance is the most famous, well-known, and over-used relationship in object-oriented programming. Inheritance is sort of like a family tree. My grandfather's last name was Phillips and my father inherited that name. I inherited it from him (along with blue eyes and a penchant for writing). In object-oriented programming, instead of inheriting features and behaviors from a person, one class can inherit attributes and methods from another class.

For example, there are thirty-two chess pieces in our chess set, but there are only six different types of pieces (pawns, rooks, bishops, knights, king, and queen), each of which behaves differently when it is moved. All of these classes of piece have properties, like color and the chess set they are part of, but they also have unique shapes when drawn on the chess board, and make different moves. See how the six types of pieces can inherit from a **Piece** class:



The hollow arrows, of course, indicate that the individual classes of pieces inherit from the **Piece** class. All the subtypes automatically have a **chess_set** and **color** attribute inherited from the base class. Each piece provides a different **shape** property (to be drawn on the screen when rendering the board), and a different **move** method to move the piece to a new position on the board at each turn.

We actually know that all subclasses of the **Piece** class need to have a **move** method, otherwise when the board tries to move the piece it will get confused. It is possible we want to create a new version of the game of chess that has one additional piece (the wizard). Our current design would allow us to design this piece without giving it a **move** method. The board would then choke when it asked the piece to move itself.

We can implement this by creating a dummy move method on the **Piece** class. The subclasses can then **override** this method with a more specific implementation. The default implementation might, for example, pop up an error message that says, **That piece cannot be moved**. Overriding methods in subtypes allows very powerful object-oriented systems to be developed. For example, if we wanted to implement a player class with artificial intelligence, we might provide a `calculate_move` method that takes a **Board** object and decides which piece to move where. A very basic class might randomly choose a piece and direction and move it. We could then override this method in a subclass with the Deep Blue implementation. The first class would be suitable for play against a raw beginner, the latter would challenge a grand master. The important thing is that other methods on the class, such as the ones that inform the board as to which move was chosen would not need to be changed; this implementation can be shared between the two classes.

In the case of chess pieces, it doesn't really make sense to provide a default implementation of the move method. All we need to do is specify that the move method is required in any subclasses. This can be done by making **Piece** an **abstract class** with the move method declared **abstract**. Abstract methods basically say "We need this method in a subclass, but we are declining to specify an implementation in this class."

Indeed, it is possible to make a class that does not implement any methods at all. Such a class would simply tell us what the class should do, but provides absolutely no advice on how to do it. In object-oriented parlance, such classes are called **interfaces**.

Inheritance provides abstraction

Now it's time for another long buzzword. **Polymorphism** is the ability to treat a class differently depending on which subclass is implemented. We've already seen it in action with the pieces system we've described. If we took the design a bit further, we'd probably see that the **Board** object can accept a move from the player and call the **move** function on the piece. The board need not ever know what type of piece it is dealing with. All it has to do is call the **move** method and the proper subclass will take care of moving it as a **Knight** or a **Pawn**.

Polymorphism is pretty cool, but it is a word that is rarely used in Python programming. Python goes an extra step past allowing a subclass of an object to be treated like a parent class. A board implemented in Python could take any object that has a move method, whether it is a Bishop piece, a car, or a duck. When **move** is called, the Bishop will move diagonally on the board, the car will drive someplace, and the duck will swim or fly, depending on its mood.

This sort of polymorphism in Python is typically referred to as **duck typing**: "If it walks like a duck or swims like a duck, it's a duck". We don't care if it really *is* a duck (inheritance), only that it swims or walks. Geese and swans might easily be able to provide the duck-like behavior we are looking for. This allows future designers to create new types of birds without actually specifying an inheritance hierarchy for aquatic birds. It also allows them to create completely different drop-in behaviors that the original designers never planned for. For example, future designers might be able to make a walking, swimming penguin that works with the same interface without ever suggesting that penguins are ducks.

Multiple inheritance

When we think of inheritance in our own family tree, we can see that we inherit features from more than just one parent. When strangers tell a proud mother that her son has, "his fathers eyes", she will typically respond along the lines of, "yes, but he got my nose".

Object-oriented design can also feature such **multiple inheritance**, which allows a subclass to inherit functionality from multiple parent classes. In practice, multiple inheritance can be tricky business, and some programming languages, (most notably, Java) strictly prohibit it. But multiple inheritance can have its uses. Most often, it can be used to create objects that have two distinct sets of behaviors. For example, an object designed to connect to a scanner and send a fax of the scanned document might be created by inheriting from two separate `scanner` and `faxer` objects.

As long as two classes have distinct interfaces, it is not normally harmful for a subclass to inherit from both of them. But it gets messy if we inherit from two classes that provide overlapping interfaces. For example, if we have a `motorcycle` class that has a `move` method, and a `boat` class also featuring a `move` method, and we want to merge them into the ultimate amphibious vehicle, how does the resulting class know what to do when we call `move`? At the design level, this needs to be explained, and at the implementation level, each programming language has different ways of deciding which parent class's method is called, or in what order.

Often, the best way to deal with it is to avoid it. If you have a design showing up like this, you're *probably* doing it wrong. Take a step back, analyze the system again, and see if you can remove the multiple inheritance relationship in favor of some other association or composite design.

Inheritance is a very powerful tool for extending behavior. It is also one of the most exciting advancements of object-oriented design over earlier paradigms. Therefore, it is often the first tool that object-oriented programmers reach for. However, it is important to recognize that owning a hammer does not turn screws into nails. Inheritance is the perfect solution for obvious *is a* relationships but it can be abused. Programmers often use inheritance to share code between two kinds of objects that are only distantly related, with no *is a* relationship in sight. While this is not necessarily a bad design, it is a terrific opportunity to ask just why they decided to design it that way, and if a different relationship or design pattern would have been more suitable.

Case study

Let's tie all our new object-oriented knowledge together by going through a few iterations of object-oriented design on a somewhat real-world example. The system we'll be modeling is a library catalog. Libraries have been tracking their inventory for centuries, originally using card catalogs, and, more recently, electronic inventories. Modern libraries have web-based catalogs that we can query from our home.

Let's start with an analysis. The local librarian has asked us to write a new card catalog program because their ancient DOS based program is ugly and out of date. That doesn't give us much detail, but before we start asking for more information, let's consider what we already know about library catalogs:

Catalogs contain lists of books. People search them to find books on certain subjects, with specific titles, or by a particular author. Books can be uniquely identified by an **International Standard Book Number (ISBN)**. Each book has a **Dewey Decimal System (DDS)** number assigned to help find it on a particular shelf.

This simple analysis tells us some of the obvious objects in the system. We quickly identify *Book* as the most important object, with several attributes already mentioned, such as author, title, subject, ISBN, and DDS number, and catalog as a sort of manager for books.

We also notice a few other objects that may or may not need to be modeled in the system. For cataloging purposes, all we need to search a book by author is an `author_name` attribute on the book. But authors are also objects, and we might want to store some other data about the author. As we ponder this, we might remember that some books have multiple authors. Suddenly, the idea of having a single `author_name` attribute on objects seems a bit silly. A list of authors associated with each book is clearly a better idea. The relationship between author and book is clearly association, since you would never say "book is an author" (it's not inheritance), and saying "book has an author", though grammatically correct, does not imply that authors are part of books (it's not aggregation). Indeed, any one author may be associated with multiple books.

We should also pay attention to the noun (nouns are always good candidates for objects) *shelf*. Is a shelf an object that needs to be modeled in a cataloging system? How do we identify an individual shelf. What happens if a book is stored at the end of one shelf, and later moved to the beginning of the next shelf because another book was inserted in the previous shelf?

DDS was designed to help locate physical books in a library. As such, storing a DDS attribute with the book should be enough to locate it, regardless of which shelf it is stored on. So we can, at least for the moment, remove shelf from our list of contending objects.

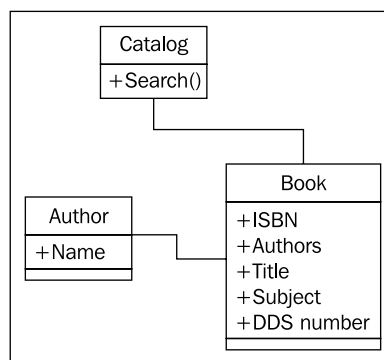
Another questionable object in the system is the user. Do we need to know anything about a specific user? Their name, address, or list of overdue books? So far the librarian has told us only that they want a catalog; they said nothing about tracking subscriptions or overdue notices. In the back of our minds, we also note that authors and users are both specific kinds of people; there might be a useful inheritance relationship here in the future.

For cataloging purposes, we decide we don't need to identify the user, for now. We can assume that a user will be searching the catalog, but we don't have to actively model them in the system, beyond providing an interface that allows them to search.

We have identified a few attributes on the book, but what properties does a catalog have? Does any one library have more than one catalog? Do we need to uniquely identify them? Obviously, the catalog has to have a list of the books it contains, somehow, but this list is probably not part of the public interface.

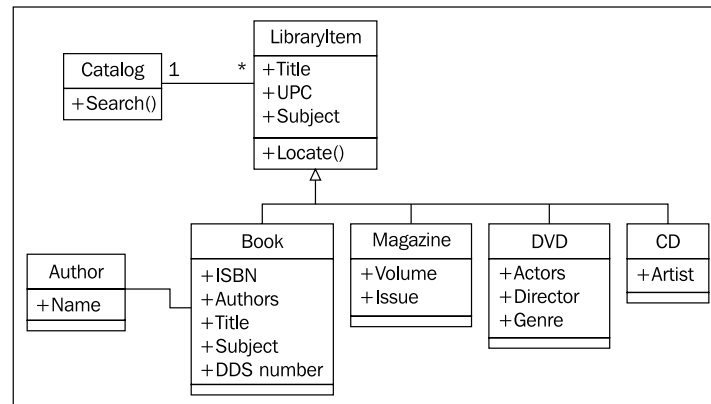
What about behaviors? The catalog clearly needs a search method, possibly separate ones for authors, titles, and subjects. Are there any behaviors on books? Would it need a preview method? Or could preview be identified by a first pages attribute, instead of a method?

The questions in the preceding discussion are all part of the object-oriented analysis phase. But intermixed with the questions, we have already identified a few key objects that are part of the design. Indeed, what you have just seen is several micro-iterations between analysis and design. Likely, these iterations would all occur in an initial meeting with the librarian. Before this meeting, however, we can already sketch out a most basic design for the objects we have concretely identified:



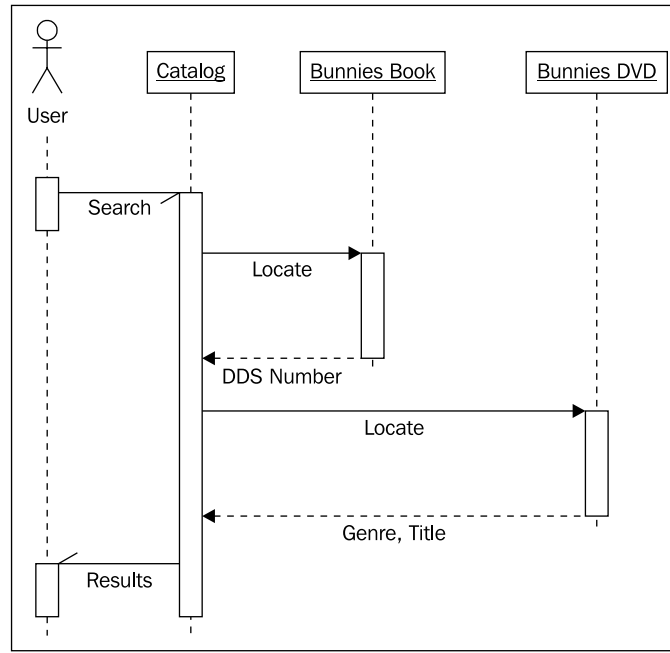
Armed with this basic diagram and a pencil to interactively improve it, we meet up with the librarian. They tell us that this is a good start, but libraries don't serve only books, they also have DVDs, magazines, and CDs, none of which have an ISBN or DDS number. All of these types of items can be uniquely identified by a UPC number, though. We remind the librarian that they have to find the items on the shelf, and these items probably aren't organized by UPC. The librarian explains that each type is organized in a different way. The CDs are mostly audio books and they only have a couple dozen in stock, so they are organized by the author's last name. DVDs are divided into genre and further organized by title. Magazines are organized by title and then refined by volume and issue number. Books are, as we had guessed, organized by DDS number.

With no previous object-oriented design experience, we might consider adding separate lists of DVDs, CDs, magazines, and books to our catalog, and search each one in turn. The trouble is, except for certain extended attributes, and identifying the physical location of the item, these items all behave in much the same. This is a job for inheritance! We quickly update our UML diagram:



The librarian understands the gist of our sketched diagram, but is a bit confused by the **locate** functionality. We explain using a specific use case where the user is searching for the word "bunnies". The user first sends a search request to the catalog. The catalog queries its internal list of items and finds a book and a DVD with "bunnies" in the title. At this point, the catalog doesn't care if it is holding a DVD, book, CD or magazine; all items are the same, as far as the catalog is concerned. But the user wants to know how to find the physical items, so the catalog would be remiss if it simply returned a list of titles. So it calls the **locate** method on the two items it has uncovered. The book's **locate** method returns a DDS number that can be used to find the shelf holding the book. The DVD is located by returning the genre and title of the DVD. The user can then visit the DVD section, find the section containing that genre, and find the specific DVD as sorted by title.

As we explain, we sketch a UML **sequence diagram** explaining how the various objects are communicating:

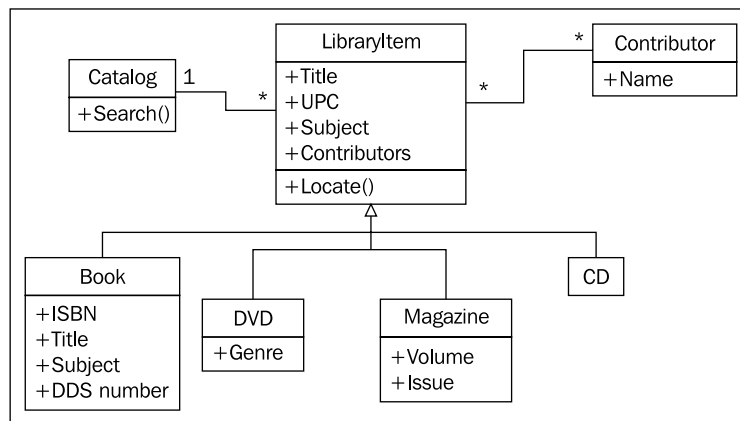


Where class diagrams describe the relationships between classes, sequence diagrams describe specific sequences of messages passed between objects. The dashed line hanging from each object is a **lifeline** describing the lifetime of the object. The wider boxes on each lifeline represent active processing in that object (where there's no box, the object is basically sitting idle, waiting for something to happen). The horizontal arrows between the lifelines indicate specific messages. The solid arrows represent methods being called, while the dashed arrows with solid heads represent the method return values. The half arrowheads indicate asynchronous messages sent to or from an object. An asynchronous message typically means the first object calls a method on the second object which returns immediately. After some processing, the second object calls a method on the first object to give it a value. This is in contrast to normal method calls, which do the processing in the method, and return a value immediately.

Sequence diagrams, like all UML diagrams, are best used when they are needed. There is no point in drawing a UML diagram for the sake of drawing a diagram. But when you need to communicate a series of interactions between two objects, the sequence diagram is a very useful tool.

Unfortunately, our class diagram so far is still a messy design. We notice that actors on DVDs and artists on CDs are all types of people, but are being treated differently from the book authors. The librarian also reminds us that most of their CDs are audio books, which have authors instead of artists.

How can we deal with different kinds of people that contribute to a title? An obvious implementation is to create a `Person` class with the person's name and other relevant details and then create subclasses of this for the artists, authors, and actors. But is inheritance really necessary here? For searching and cataloging purposes, we don't really care that acting and writing are two very different activities. If we were doing an economic simulation, it would make sense to give separate actor and author classes different `calculate_income` and `perform_job` methods, but for cataloging purposes, it is probably enough to know how the person contributed to the item. We recognize that all items have one or more **Contributor** objects, so we move the author relationship from the book to its parent class:

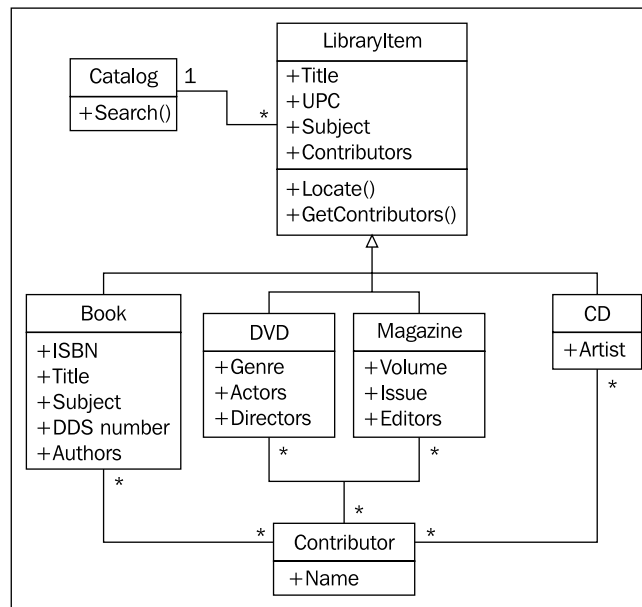


The multiplicity of the **Contributor/LibraryItem** relationship is **many-to-many**, as indicated by the * at each end of the relationship. Any one library item might have more than one contributor (for example, several actors and a director on DVD). And many authors write many books, so they would be attached to multiple library items.

This little change, while it looks a bit cleaner and simpler has lost some vital information. We can still tell who contributed to a specific library item, but we don't know how they contributed. Were they the director or an actor? Did they write the audio book, or were they the voice that narrated the book?

It would be nice if we could just add a `contributor_type` attribute on the **Contributor** class, but this will fall apart when dealing with multi-talented people who have both authored books and directed movies.

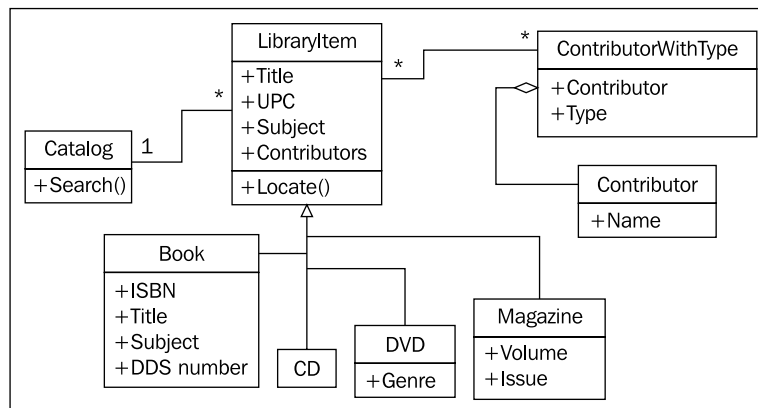
One option is to add attributes to each of our **LibraryItem** subclasses that hold the information we need, such as **Author** on **Book**, or **Artist** on **CD**, and then make the relationship to those properties all point to the **Contributor** class. The problem with this is that we lose a lot of polymorphic elegance. If we want to list the contributors to an item, we have to look for specific attributes on that item, such as **Authors** or **Actors**. We can alleviate this by adding a **GetContributors** method on the **LibraryItem** class that subclasses can override. Then the catalog never has to know what attributes the objects are querying; we've abstracted the public interface:



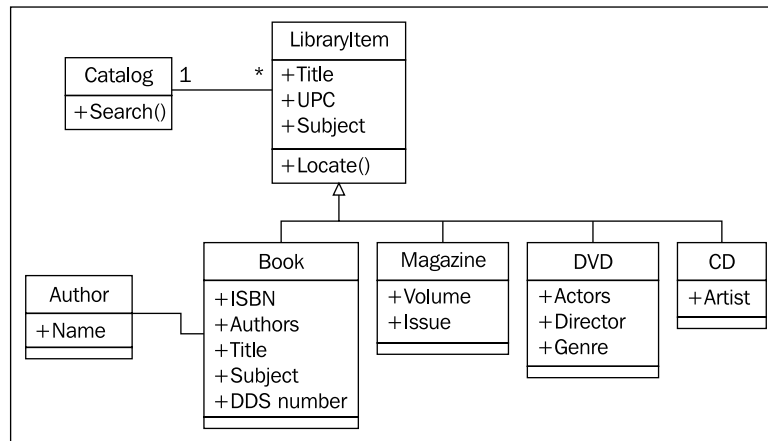
Just looking at this class diagram, it feels like we are doing something wrong. It is bulky and fragile. It may do everything we need, but it feels like it will be hard to maintain or extend. There are too many relationships, and too many classes would be affected by modifications to any one class. It looks like spaghetti and meatballs.

Now that we've explored inheritance as an option, and found it wanting, we might look back at our previous composition-based diagram, where **Contributor** was attached directly to **LibraryItem**. With some thought, we can see that we actually only need to add one more relationship to a brand-new class to identify the type of contributor. This is an important step in object-oriented design. We are now adding a class to the design that is intended to *support* the other objects, rather than modeling any part of the initial requirements. We are **refactoring** the design to facilitate the objects in the system, rather than objects in real life. Refactoring is an essential process in the maintenance of a program or design. The goal of refactoring is to improve the design by moving code around, removing duplicate code or complex relationships in favor of simpler, more elegant designs.

This new class is composed of a **Contributor** and an extra attribute identifying the type of contribution the person has made to the given **LibraryItem**. There can be many such contributions to a particular **LibraryItem**, and one contributor can contribute in the same way to different items. The diagram communicates this design very well:



At first, this composition relationship looks less natural than the inheritance-based relationships. But it has the advantage of allowing us to add new types of contributions without adding a new class to the design. Inheritance is most useful when the subclasses have some kind of specialization. Specialization is creating or changing attributes or behaviors on the subclass to make it somehow different from the parent class. It seems silly to create a bunch of empty classes solely for identifying different types of objects (this attitude is less prevalent among Java and other "everything is an object" programmers, but it is common among more practical Python designers). If we look at the inheritance version of the diagram, we can see a bunch of subclasses that don't actually do anything:



Sometimes it is important to recognize when not to use object-oriented principles. This example of when not to use inheritance is a good reminder that objects are just tools, and not rules.

Exercises

This is a practical book, not a textbook. As such, I'm not about to assign you a bunch of fake object-oriented analysis problems to create designs for. Instead, I want to give you some things to think about that you can apply to your own projects. If you have previous object-oriented experience, you won't need to put much effort into these. But they are useful mental exercises if you've been using Python for a while but never really cared about all that class stuff.

First, think about a recent programming project you've completed. Identify the most prominent object in the design. Try to think of as many attributes for this object as possible. Did it have: Color? Weight? Size? Profit? Cost? Name? ID number? Price? Style? Think about the attribute types. Were they primitives or classes? Were some of those attributes actually behaviors in disguise? Sometimes what looks like data is actually calculated from other data on the object, and you can use a method to do those calculations. What other methods or behaviors did the object have? What objects called those methods. What kinds of relationships did they have to this object?

Now think about an upcoming project. It doesn't matter what the project is; it might be a fun free-time project or a multi-million dollar contract. It doesn't have to be a complete application; it could just be one subsystem. Perform a basic object-oriented analysis. Identify the requirements and the interacting objects. Sketch out a class diagram featuring the very highest level of abstraction on that system. Identify the major interacting objects. Identify minor supporting objects. Go into detail for the attributes and methods of some of the most interesting ones. Take different objects to different levels of abstraction. Look for places you can use inheritance or composition. Look for places you should avoid inheritance.

The goal is not to design a system (although you're certainly welcome to do so if inclination meets both ambition and available time). The goal is to think about object-oriented designs. Focusing on projects that you have worked on or are expecting to work on in the future simply makes it real.

Now visit your favorite search engine and look up some tutorials on UML. There are dozens, so find the one that suits your preferred method of study. Sketch some class diagrams or a sequence diagram for the objects you identified earlier. Don't get too hung up on memorizing the syntax (after all, if it is important, you can always look it up again), just get a feel for the language. Something will stay lodged in your brain, and it can make communicating a bit easier if you can quickly sketch a diagram for your next OOP discussion.

Summary

In this chapter, we took a whirlwind tour through the terminology of the object-oriented paradigm, focusing on object-oriented design. We learned how to separate different objects into a taxonomy of different classes and to describe the attributes and behaviors of those objects via the class interface. In particular, we covered:

- Classes and objects
- Abstraction, encapsulation, and information hiding
- Designing a public interface
- Object relations: association, composition, and inheritance
- Basic UML syntax for fun and communication

In the next chapter, we'll explore how to implement classes and methods in Python.