Chapter 2
Conceptual and Executable Models: A First Look

Pedagogic Motivations

- In this chapter, we take the ideas of identifying objects in a problem domain and begin to put them into practice by creating our first conceptual model.
- Design and design issues are picked up with the introduction of notation for expressing design (UML) and our first relationship (instantiation dependency).
- We use DrJava to instantiate our top-level objects to avoid the main method and instead focus on the idea of the class' constructor creating an object.
- All of the classes in this text are written with an explicit constructor. We do not rely on the constructor that Java inserts for our classes. We also do not rely on the default package that Java provides for us; we insist that all of our Java classes are in a named package.
- Since all of our classes are organized in some named package, and at times we need to refer to classes outside of a package, we adopt the convention to use fully-qualified class names rather than imports. This further reinforces the concept of packages as an organizational tool.

Introduction

In the last chapter, we learned that a computer is a general purpose problem solver. For this reason it is interesting to learn how to instruct a computer to solve problems (i.e. how to write computer programs). We also sketched a problem solving process for analyzing problems and building computer programs to solve them. We promised there that we would refine the process and fill in the missing details throughout the rest of the book. In this chapter we begin to make good on that promise.

Last chapter we discussed the need to identify the domain of a problem (the objects in that domain). Identifying the domain is simply the first step in creating a useful software system. Once we have decided how to model a system, we need to take the critical step of actually creating an executable model in a programming language (Java in our case). In this chapter we explain, in more detail, just how to do that.

Making a model more explicit – UML

We started out creating our models in English. We described the domain, the objects in the domain, their properties and capabilities, all in English. This approach is fine if you simply want to describe how to solve a problem to someone. However, our goal is to create executable models: actual working computer programs. In order to do this, we need to move

away from the natural language we are comfortable with and towards the programming language the computer understands. For us, the first step in this process is to convert our natural language models into another language called *UML*.

UML stands for Unified Modeling Language. It provides a way for people to visualize the structure of a system through the use of diagrams. UML is a fairly large language and we will not study all of it in this book. Instead, we will focus on part of the language that will help us better visualize the models we are creating – *class diagrams*. When we are creating our UML diagrams,[1] we will draw one box for each type of object[2] we have identified in our problem domain. Remember that the essential components of objects are their properties, capabilities and relationships with other objects. UML diagrams allow us to express all of these things.

We begin by showing how we specify capabilities and relationships between objects of objects in UML diagrams. Later we will learn how to represent properties of objects.

What do the Boxes Represent?

The goal of this chapter is to realize our conceptual models in executable form (for us, as computer programs expressed in the programming language Java). We have spent some time thinking about the objects that should be part of our model, as well as the properties and capabilities those objects should have. When we move into actually creating the model (the program), we need a way to express to the computer what the objects are. More accurately, we need to formally define what the objects are and what they will do.

In UML, we use one box to represent each ***type of*** object, but the boxes do not represent objects themselves. The boxes in UML represent types. UML recognizes different sorts of types. The first is called a *class*. **Classes are the formal definitions of what properties we want our objects to have, what capabilities we want them to be able to perform, and the relationships they have with other types of objects.** Objects in the problem domain are represented by objects in our programs; the objects in our programs are created from their formal class definitions.

So, we need to identify the objects from our problem domain using the problem description, and then define them formally (as a class). After the class has been defined, we use those classes to create the actual things that will do the work (the objects). It seems like we are doing twice the work we should need to get the task accomplished. However, there are actually quite a few tasks that are analogous to this one in the real world.

Example 1

Suppose you are trying to build a building. Pick your favorite kind, a house, skyscraper, or a really cool shed for the backyard. In order to do this,

[1] For the remainder of the text, when we refer to UML diagrams, we are referring to UML class diagrams.

[2] There is a difference between an object and the type of an object. To make clear the distinction between object and type, consider the famous dogs Lassie, Spot, and Clifford. Using our terminology, Lassie is an object, Spot is an object and Clifford is an object. Each of these three objects belongs to the type dog. The type of an object tells us something about its typical properties, capabilities and relationships with other (types of) objects. We will explore these ideas and their consequences in more detail throughout the book.

you first need to plan. You need to decide what you want your building to look like and what types of rooms and other things the building should have.

After some initial planning, it is a good idea to bring in an architect. Architects help you draw up formal plans for your building. They can also help illuminate some problems with the initial plans you made. For example, the second floor of a house should not be twice as big as the first floor (it just will not work). After you have those plans, you can actually go off, build your building, and begin using it.

However, after the building is built, do not the blueprints for the building seem like a waste? Not really, because if you ever had to go back and change something about the building you would have the original plans to start with. Moreover, you might know someone who built a totally awesome shed in their backyard and then had a friend come over and see it. That friend liked the shed so much; they wanted one just like it. For the friend, building their shed was much easier because all they had to do was take the plans and make another instance of that shed in their backyard.

The process we just described is similar to what we are doing when we create a program. We plan it out (design it), create the model (first in UML and then in Java code), and then run the code to do the actual work (e.g. - build the building). We retain the blueprint (the class diagram and Java code) to do the work again if needed or if we want to change the program and run it again.

When you write code, you can think of yourself as the customer,[3] architect and builder rolled into one. You do the design, the blueprints, and write the code all yourself. This isn't the way it always works: often these roles are played by different people. However, when you are starting out it is helpful to see the whole process of constructing software.

Modeling in UML: Drawing a Class Box

In UML, each class box is separated into three sections. The first section gives the name of the class.[4] The second section gives the properties of the class. The third section gives what the capabilities of the class.[5]

[3] More often, there is actually a customer you are building code for. It can be a real paying customer, if you are in the software development business, or the instructor for your class if you are working on your assignments.
[4] The "C" in the green circle indicates that this is a box for a class. We will see later on that our UML diagrams can contain other kinds of boxes, which will have different annotations. These annotations are not a formal part of UML, but correspond to the symbols that Eclipse uses for these elements.
[5] A UML diagram can give as much (or as little) information about the properties and capabilities as deemed necessary by those using the diagram. Some diagrams may list all properties and capabilities in each class box, but others may not. The UML tool Green lists all properties and all capabilities.

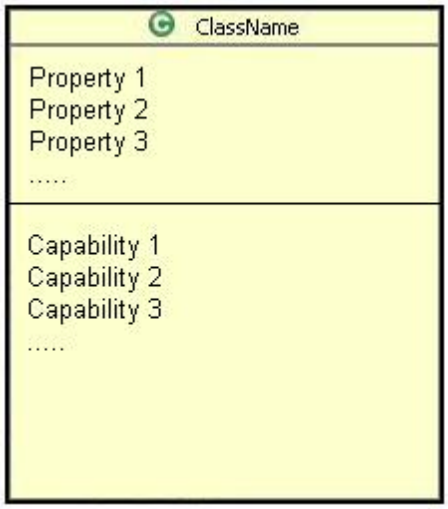


Figure 1 UML Class Box

There are many software tools out there that can help a software developer create UML diagrams electronically so that they can be shared by all the people working on a project. Many of these tools also provide a way to quickly translate between the UML diagram and the actual source code that it represents. Your instructor might show you a UML editor to use while you work on projects in this class. Throughout the book, we will often show UML diagrams of programs along with or even instead of the actual program code[6]. It is oftentimes easier to visualize how a system is put together using the UML diagram as opposed to the program's source code.

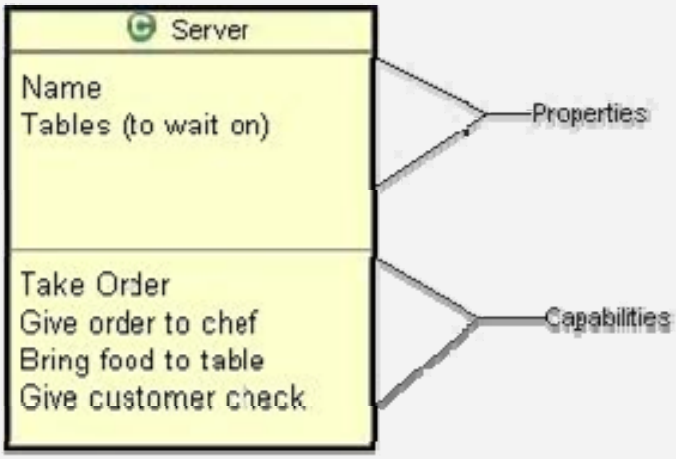
Let us now look at some example problems which describe the types of things you will see in a UML diagram. The diagrams in these examples are not completely correct UML. At this point we want to emphasize what information belongs in the diagram. We will learn how to express that information in correct UML notation as we continue through this chapter and the rest of the book.

Example 2

In our restaurant example, we talked about servers, a chef, and other people who were in the restaurant. If we wanted to draw the UML diagram[7] for the server, it would look something like this:

[6] The UML diagrams in this book were created with a tool called Green (http://green.sourceforge.net). Green is a free, open-source plug-in for the Eclipse IDE (http://www.eclipse.org).

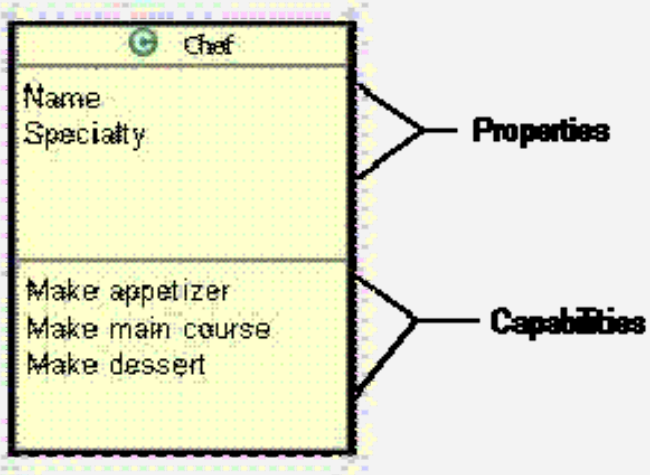
[7] If you have used UML class diagrams before, you will note that in these first examples, we are not using the formal notation for properties and capabilities in our UML class boxes. As we go through the next few chapters, we will see the formal notation really used in UML and our UML diagrams will slowly begin to look like standard UML diagrams.



The Server has a name as a property as well as the set of tables that they are responsible for waiting on. The server's capabilities at the restaurant include taking an order from the patrons of the restaurant, giving the orders to the chef, bringing prepared food to the table and giving the customer the check for the food they have ordered.

Example 3

If we wanted to draw the UML diagram for the Chef, it might look something like this:



The Chef has a name as a property as well as a specialty (what the chef cooks best). Chefs make all the food for the restaurant, so they have the capabilities to make an appetizer, or a main course, or a dessert.

Example 4

In our restaurant example, we have been so busy worrying about the things that make up the operations and people in a restaurant, we need to remember that we need an actual restaurant class as well. Our restaurant class in UML might look like this:

This Restaurant class is important in that it encompasses the other classes in our model. For example, both the Server class and the Chef class are somehow related to the Restaurant class.

UML diagrams help us visualize the properties and capabilities for each type of object in our system. In each class box we create, we give the names, properties, and capabilities for each class. Recall from our discussion last chapter that objects are a mechanism for encapsulation. When we see the drawings of our class boxes, we can physically see this encapsulation as all of the properties and capabilities are enclosed in the class box, keeping them together as one computational unit.

At the end of the last chapter we introduced a small scenario, consisting of an ant and terrarium. We will revisit this scenario in this chapter, and use it to demonstrate how the UML can be used to visualize the structure of a system, and also how a UML tool like Green can make a connection between a UML diagram and an executable model. However, before doing this we need to cover a bit of background.

How and where do you write a program?

In chapter 1, we began identifying our problem domain and the objects within that domain. We now know that to get an executable model on the computer, we will have to create formal definitions for our objects (classes). We are using UML to help us organize our ideas about what our classes need to look like. When we create our UML diagrams, we are

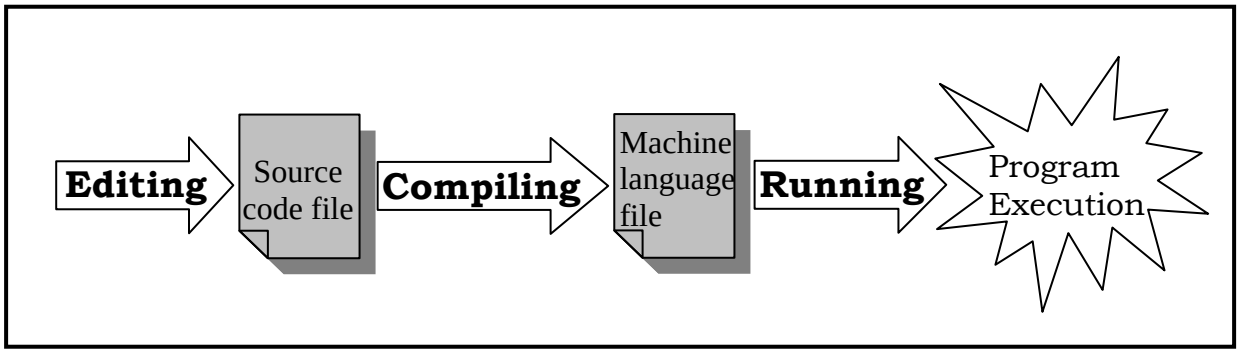
showing through the class box what properties and capabilities will be defined for a particular class. The next step is to learn how to take UML diagrams and turn them into a computer program. We will specifically learn how to write a computer program using the Java programming language, but many of the steps we will learn apply to writing a computer program in any programming language.

In the Introduction, we learned what a programming language is, and that in this text we will use the Java programming language to communicate our instructions to a computer. Most people do not think about how they communicate with a computer because computers have become such a daily part of our lives. When we type letters on the keyboard, press the "Enter" key, click a mouse button, or select an option from a menu, we are issuing commands to a computer. The computer needs to know how to respond to these commands, and that is where a computer program comes in. The program decides what a left mouse click on the menu named "Save" means for that program. This piece of the process is one that computer users do not usually see. This is the part that the software developer is uniquely responsible for.

In order to get a computer to perform the actions we need to help us solve the problem, we need to give it instructions using a programming language (Java). A program written in a programming language is often called *source code*. Source code consists of the actual instructions that we want the computer to execute.

Where do we write the source code? In chapter 1 we saw that we could type Java source code into the DrJava interactions pane and have the computer act on our instructions. This method works great for writing a small number of instructions. For writing larger programs this approach quickly becomes impractical. Instead, source code is written into a file that is stored on the computer.

We have also already learned that programs expressed in a high-level computer programming language cannot be directly carried out by a computer. Programs must be expressed in a very low-level language, called machine language, before the hardware of a computer can carry out the instructions it contains. A computer program called a compiler translates programs expressed in high-level programming languages into equivalent programs expressed in low-level languages. In other words, a compiler takes as its input a computer program and produces as its output another computer program.



The process of editing, compiling and running a computer program

Since we know that we are dealing with Java, we can provide a little more detail. Since we are expressing our programs in the Java programming language, then the source code file will contain Java source code. You might wonder how the computer knows that there is Java source code in a particular file, as opposed to something else (perhaps an essay, a spreadsheet, a song, a picture or a video)? A computer determines the contents of a source code file in the same way it recognizes which files are text documents and which are images: it looks at the file extension. For Java source code files, the extension is `.java`. However, to get a file with a `.java` extension we must use an *editor* that will allow us to write Java source code. An editor is a computer program that allows us to create files and insert information (text) into them. There are many special-purpose editors that will allow us to create files of different types. For example, Word is a document editor (a "word processor") whereas Paint is an image editor. There are general purpose text editors, such as pico, Notepad. These have the advantage (and disadvantage) of being very simple.

Programmers have found that they are more productive when they use editors suited to writing source code in a programming language. Programmers tend to use program editors (sometimes called development environments). One such program editor is called Emacs. Emacs is free (see http://www.gnu.org/software/emacs/ and http://www.xemacs.org/) and is customizable to provide support for writing programs in many different programming languages.

More recently *integrated development environments*, or *IDE*s, have become common tools for people writing computer programs. An IDE typically provides more support within the editor for doing software development tasks, than a run-of-the-mill program editor. Some examples of this type of program are Eclipse, JBuilder, and NetBeans. Your instructor will need to tell you which type of program (editor or IDE) you will be using and how to use that program to create a file that contains Java source code. We use Eclipse, and so the examples we show will be of Eclipse. However, any editor or IDE can serve just as well.

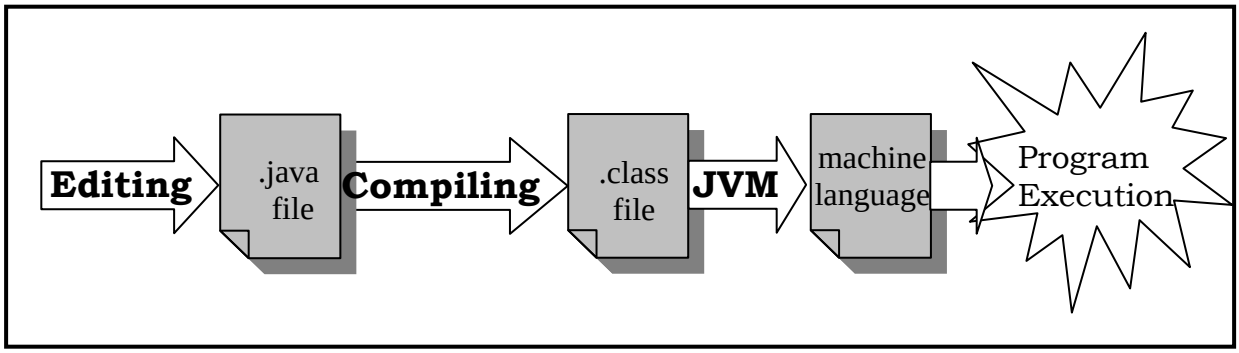
Once you have typed your source code into a file and given it a name with the `.java` extension, your job as a programmer is not yet complete. Remember that the source code that you have written is a set of instructions to the computer and the computer needs to be able to read and understand those instructions. As we said before, computers do not understand English, or even Java source code. In order to get a program written in Java converted into a language that the computer understands, we use a special translating program called a *compiler*. The compiler for Java is called `javac`. Depending on the type of environment you are using to develop your code, the way that you run the compiler is different. Your instructor will need to tell you how you need to run the compiler on your source code.

However, no matter how this process happens, at the end of it the compiler will generate another file with a `.class` extension. This file will have the same name as your source code file, but the extension and the contents will be different. This file contains Java *bytecode*. The bytecode in this file is comprehensible only by the computer. You can look at the contents of a .class file by opening it up in your editor. You should do so at least once so you can see for yourself what it contains. Chances are pretty good that you will not be able to understand the contents. Do not worry though; the computer understands it just fine.

The bytecode can be read by the computer, but we need to tell the computer that it should read the bytecode and execute the instructions it contains. This is the last step of the process

of writing a program – actually telling the computer to run the program. To run a Java program from the bytecode we run another program, called `java`, on the bytecode.

When we run the program called `java`, the bytecode is interpreted by the *Java Virtual Machine* (JVM). The JVM acts as an intermediary between the bytecode instructions and the actual underlying computer. The JVM reads the bytecode and then talks to the internals of the computer to actually execute the instructions. The use of the JVM has led to the ability of Java programs to run on many different types of computers even if they use different operating systems.



The process of creating a working Java program

Edit-Compile-Run Cycle

The process that we have just talked about [editing to create a .java file, compiling to create a .class file, and using the JVM to run a Java program] is referred to as the *edit-compile-run cycle*. The most interesting thing about this process is that it truly is a cycle. After editing, compiling, and running a program, it is often the case that it does not function 100% correctly, or perhaps more to the point, it is not complete yet (think incremental development). Therefore, the developer needs to go back and write some more code (or fix code that was not written correctly the first time) re-compile and re-run the program. In fact, the best software is usually developed incrementally with this cycle in mind. The developer builds a little bit of a program, compiles it, and runs it to check the first little bit of functionality and ensure that it is working correctly. Then she or he adds more functionality, compiles and runs again. This process continues until the program is completely built and functions as required.

If you are using an IDE such as Eclipse to write your programs, you do not always need to explicitly run the compiler to translate your source code into executable code. This is because many IDEs continually compile your source code for you, as you type it. This has the advantage that you get immediate feedback on some kinds of errors (called syntax errors) as you type. Syntax errors are akin to spelling and grammar mistakes in writing essays.

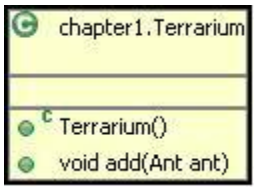
Our first conceptual model

Recall the example from the end of last chapter, which consisted of a Terrarium and an Ant. When working with this example we used the interactions pane of DrJava to create an

instance of each of these classes and make a connection between the Ant and the Terrarium. We did this by writing the following:

```
new chapter1.Terrarium().add(new chapter1.Ant())
```

Let's use Green to show us the class box for the Terrarium class:



Like the examples we saw earlier in the chapter, this class box is divided into three parts. The top part contains the name of the class, which is formally chapter1.Terrarium. The middle part lists the properties of objects belonging to this class. Since this part is empty, there are no properties defined in this class. Finally, the bottom part shows capabilities.

A few things about this diagram are notable. We have already pointed out that the "C" in a green circle in the name area indicates that this box represents a class. In the capabilities section we see smaller green circles. Such a green circle indicates a capability that anyone can access.[8] The green "C" above and to the right of the green circle next to the "Terrarium()" capability indicates that this capability is special – it marks the class' *constructor*. A constructor is a special capability used only when an object is first created. In fact, when writing an expression like,

```
new chapter1.Terrarium()
```

we are in fact referring to the constructor of the class whose full name is chapter1.Terrarium when we write chapter1.Terrarium().

The second capability shown is the "add" capability, which we used to make the connection between the Ant object and the Terrarium object when we wrote:

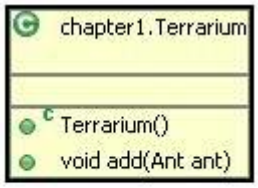
```
new chapter1.Terrarium().add(new chapter1.Ant())
```

If we look at the class box for the class chapter1.Ant we see something similar:

[8] In Java terminology, which we will explain in more detail later, the green circle indicates a *public method*. A "method" corresponds to a capability, and "public" means accessible to anyone.

The constructor of this class is Ant(). The class provides two capabilities aside from the constructor, start() and stop(). Again, there are more details in this diagram, which we will explain a little later on in the chapter.

If we look at both of these classes in a single UML diagram, here's what it looks like:



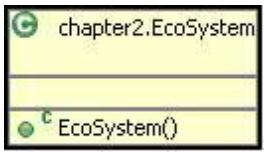
While both class boxes are shown, there is no apparent connection between the two classes. But think about what we did when we wrote:

```
new chapter1.Terrarium().add(new chapter1.Ant())
```

In this expression we create two objects. Where should this happen in our model? The idea we want to explore now is that we always want to have an object representing our system of interacting objects as a whole. We call this object a *top level* object. The top level object of a system has the responsibility for creating, either directly or indirectly, the objects which together make up the system.

To model a top level object, we draw a UML class box. This box will be labeled with the name `chapter2.EcoSystem`. When we draw such a class box using a tool like Green, not only are we creating a UML diagram, we are also creating the Java source code which

corresponds to that diagram.[9] The following sections explain the structure of a Java source code file in general, and the structure of the source code of this top level object in particular:



Syntax & Semantics

In the edit-compile-run cycle, the bulk of the work comes in the edit phase when you actually compose the Java source code. Remember that although we show our examples in Java, there are many programming languages available.

We know that programming languages are different than natural (human) languages, but in some ways they are similar. When someone is trying to learn a new natural language, they spend a lot of time learning about the words that make up the language and translating them into words familiar to them in their own language, essentially focusing on the meanings of the words. However, they also spend time learning about the grammatical structure of the language and the way sentences should be constructed so that they can speak and/or write in the correct way. For example, the following two sentences, though they consist of the same words, mean two quite different things:

The dog chased the cat.

The cat chased the dog.

Therefore, when you learn a new natural language, you must focus on the grammar (structure or syntax) of the language and also the words and their meanings in context (semantics).

When you study a programming language, you need to pay attention to these same issues: *syntax* (grammar) and *semantics* (meaning). What is different about a programming language is that you have the meanest, nastiest grammarian working with you at all times: the compiler. In programming, we am much more careful about grammar than we would when talking to our friends or even writing an email. Did you notice that the grammar in the previous sentence was wrong? If you did, good for you! If not, be warned that the compiler will never let a mistake like that go by without yelling at you about it. No matter how big or small, a grammatical/syntax error in your program will always be caught by the compiler. When the compiler catches a mistake, it does not compile your program into Java bytecode. You must fix the mistake, after which you can and try again to compile your program. Once your program compiles successfully, you can run it.

[9] Green is what is called a live round-tripping tool. This means that when you use Green to draw a diagram, it immediately creates the source code which goes along with the diagram (this part is sometimes called *code generation*). When you write source code, Green will update any corresponding diagram (this part is sometimes called *reverse engineering*). A tool which can make a round trip from code to diagram to code (or diagram to code to diagram) is called a *round-tripping* tool. Because changes in the code happen as you draw the diagram, and likewise the diagram updates as you change he code, the tool is referred to as a *live* round-tripping tool.

This syntactic strictness can be most frustrating when learning a programming language. We as humans are used to poor grammar and can often process what someone with poor grammar is trying to say to us. However, the computer does not have this ability. The syntax must be perfect for the computer to understand what we want it to do. That is one reason the compiler is there – to help us create grammatically correct programs so that it can translate them into the language the computer can understand.

There is no program analogous to the compiler for the semantics of a language. If we do not understand what some of our program code does, or the code we write does the wrong thing once run, there is no way for the computer to tell us that ahead of time. For example, if we say "The dog chased the cat" when we meant to say "The cat chased the dog", the computer won't complain. Since both of these sentences are (grammatically) well-formed, the compiler will accept either one as correct. The computer cannot read our minds to determine what we *meant* to say. It takes what we *do* say quite literally. We need to take the time as we are writing our programs to understand what the program is doing and make sure that the code we write will do what we want it to do when we execute the program.

We will spend some of our time learning the syntax of Java so that we do not have to fight with the compiler as much, and we will spend some of our time learning the semantics of Java so that we know what our code actually accomplishes after we write it. But most of all, we will focus on how to model real-world problems and design effective solutions for them.

Basic Structure of a Java Source Code File

The first piece of syntax we will learn about is the fundamental structure of a Java source code file. In each of the Java source code files we create, we will put the definition of exactly one class. Therefore, we will have one file for each type of object we have identified from our problem description; equivalently, we will have one file for each one of the UML class boxes we have created. Since our programs will consist of many different types of objects, our programs will actually consist of many Java files, each containing one class definition. Green, our UML tool, is more than a tool for drawing diagrams. It is able to generate (some of) the Java code which underlies the things we draw. For example, if we create the class box for the class `chapter2.TopLevel`, it creates the following Java code:

```
Error! Not a valid filename.
```

Let us now take a look at the basic structure of a class file. It consists of two elements:

```
Package Declaration
Class Definition
```

We will talk about what goes inside the package declaration in the next section. The class definition is further broken down into two elements of its own:


```
Class Header
Class Body
```

Inside the class body is where we define the properties and capabilities of a class. A more complete picture of the structure of a Java source code file looks like this:

```
Package Declaration
Class Header
Class Body made up of:
            Definitions of Properties
            Definitions of Capabilities
```

Before begin to put Java syntax into our class definition, let us introduce a convention that we will use to make describing Java syntax a little easier. To introduce the convention we will take a little detour into the world of business letters.

Business letters follow a fairly strict format. We might describe the format (the syntax) of a standard business letter as follows:

Address
Date

Name
Address

To whom it may concern:

Text of letter

Yours very truly,
Name

This provides a template from which you can create your own business letters. In this example, the parts of the template which are fixed are written in normal typeface, whereas the parts which need to be filled in (the placeholders in the template) are shown in *italics*. Note that some of the names of the templates are duplicated. These template names (such as *Address* and *Name*) tell us what type of thing gets substituted in the template. However, we must know that the first occurrence of address gets substituted with the sender's address, while the second occurrence is for the recipient's address. Likewise, the first name is the recipient's name, while the second is the sender's name. A completed letter using this template might look as follows:

123 Main Street
Buffalo, NY 14260

June 3

Rainbow Painting
124 Main Street
Buffalo, NY 14260

To whom it may concern:

I am writing to thank you for the marvelous job you did painting our house.

Yours very truly,
Ardelia Jones

In our explanation of how to write well-formed (syntactically correct) Java programs we will use the same convention: things written in italics are placeholders for information which you as the program writer can fill in, whereas things written in regular font must be written exactly as shown.

Another convention we will adopt when showing syntax is how we will show certain special characters that you will need to type in your programs. There are quite a few special characters that you will see in your Java code and when a certain character needs to appear in the code, we will give you its name as well as the symbol you will need to write in the code. This symbol will be surrounded by the single quotes, like the percent sign shown here '%'. However, if you need to type the percent sign in your code, you must not insert the single quotes around it. You just need to type the percent sign at the appropriate place in the code.

Now let us continue our explanation of the structure of Java source code files.

Package Declaration

The first line of a Java source code file is the *package declaration*, indicating which *package* the class belongs to. A package is a way for us to group together classes that have a common purpose. When we put classes in the same package, we are telling Java that they belong together. Classes in the same package are stored in the same directory on the file system inside the computer.

The actual syntax for a package declaration looks like this:

```
package identifier;
```

`package` is a special word to Java. It is called a *keyword*. Most programming languages have words like this. These words are kept special by the language to mean specific things. Whenever anyone uses the language, these words mean exactly the same thing and no one can use them to mean anything different (the compiler enforces this for us). The keyword `package` is always followed by the name of the package in which that file belongs. In the definition above, we see the word identifier (in italics) where the name of the

package should go. Recall that words given in italics are words that we will replace with other words in our actual Java files. We use the word identifier in the package definition as a placeholder for the name of the package. The word identifier has a special meaning in the programming world.

Identifiers are names that programmers pick out. As the programmer, you have the freedom to pick out the names for some of the different parts of the code you write. However, there are a few rules that we must follow when picking out identifiers. These rules that we discuss are enforced by the compiler. That is, the compiler will not allow us to continue in our edit-compile-run cycle without sticking to these rules. The rules for naming identifiers are pretty simple:

- Identifiers must begin with a letter or underscore '_' (Remember: don't include the quotes if using the underscore.)
- The first character of an identifier can then be followed by zero or more letters, digits, or underscores.
- Other than letters, digits, or underscores, no other characters can be part of an identifier. That means no spaces, or other special characters (like the '`*`' or '`;`').
- Keywords[10] are not allowed to be identifiers.

As long as we follow these rules, the compiler will let us name our packages anything we want. You should note that the Java language is case-sensitive. That means an upper case letter and a lower case letter are different to Java. So, the identifiers `hello` and `Hello` do not mean the same thing to Java. Therefore, it is important to watch the capitalization of words in your program. Another thing that Java looks for is spelling. This is not what you might think. Java is not spell-checked. If you type a word incorrectly as an identifier, Java will not tell you or try to auto-correct it for you. Java simply believes that you would like the name of the identifier to be whatever you type. However, Java will remember the way you type the identifier and expect you to type it the same way each time it is used. Java does not expect you to be a good speller, but you must be a consistent speller.

The compiler enforces several rules that we must abide by. However, the compiler does not regulate everything a developer does. Nonetheless, one of the things we must keep in mind as developers of software is that the code we write will most likely be looked at and read by other people. We should strive when we name things to make sure that the names make sense in the context of the program we are trying to build. Remember back to the modeling we discussed in the last chapter – we should name things so that they correspond to and describe what we are trying to model.

Another thing that we must be aware of is that many times we are not writing code for ourselves, but rather for someone else (like the company you work for, or the instructor that teaches the class). Therefore, there may be rules that you need to conform to that are not compiler enforced, but are enforced by the community of programmers around you (at your company or within the class). These types of rules are sometimes called stylistic guidelines or *naming conventions*. Sometimes companies have huge documents devoted to these guidelines or coding standards for a company that all company-written code must follow.

The reason that coding standards are put into place is to enhance readability. In other words, code will have the same style regardless of who wrote it, which makes it easier to read

[10] A list of Java keywords is available at the end of this text.

the code and understand what it does, than if everyone wrote code using a different style. One way to think of this is that it is easier to read a book written by several people if they all use a word processor to write their text, rather than writing it out in their own handwriting. We will follow several stylistic guidelines throughout this book. For example, the name of every package we write will begin with a lower case letter, and all of the letters in the name of the package will be lower case. Sometimes we might want our package names to be more than one word long. This is fine to do, but we must remember that there are no spaces allowed in identifiers, so all the words will be compressed together. Also, with the naming convention for packages being all lower case letters, the package name with multiple words could look something like this: `ourfirstpackagename`.

The package declaration ends with a semicolon '`;`'. The semicolon will play an important role in your life as a Java programmer. In this case, it tells us that the package declaration is finished. Later in this chapter, we will see other parts of code that contain a semicolon and will see that the semicolon plays a different role depending on what part of code we see it in.

Nested Packages

We can also define a nested package structure for our classes. This means we can have packages inside other packages to further help us in grouping like classes together. For example, all the code for each of the chapters in this book appear in packages that correspond to the name of the chapter. Code from this chapter is in a package named `chapter2`. However, within this chapter, there may be a few different examples that are used. We might develop some code for a restaurant example and some other code for a car wash example, and we would like to keep the code for these examples separate from other. To do this we could put the example code for the restaurant in a package named `chapter2.restaurant` and code for the car wash example in a package named `chapter2.carwash`. Notice the period '`.`' in the name of the package. The period (more commonly called "dot" in this context) is an indication to Java that the packages are nested. Inside a package named `chapter2` there is a package named `restaurant` as well as a package named `carwash`.

Packages (and nested packages) correspond to the directories[11] (and subdirectories) the files are stored in on the filesystem of the machine. On our filesystem, there is a directory named `chapter2` and inside that are directories named `restaurant` and `carwash`. Inside each of these directories is where the Java source code files that belong to these packages are located.

When writing and looking at Java code, you will see a mix of classes that belong to nested and non-nested packages. Either way, the class belongs to a package and it is the package declaration that you will see as the first line of the Java source code file.

Returning now to our EcoSystem example,

```
Error! Not a valid filename.
```

[11] Different operating systems refer to directories by different names. In some operating systems, such as Windows and MacOS, directories are referred to as *folders*.

we see that the package for this code is called `chapter2`.

Class Header

The class header is the beginning of the definition of the class. The class header looks like this:

```
public class identifier
```

`public` is a keyword in Java. `public` is called an *access control modifier*. Access control modifiers express a programming entity's[12] ability to be recognized and available to other programming entities. Inside a program there are some things that we want to make `public`, meaning other parts of the program have access to them, and there are some things we do not want to make `public` ; there are other access control modifiers for those. For a class definition, we want to let other parts of the code have access to it. If we did not allow access to the class, then there would be no way to set up the communication between objects that is needed to get the work of the program done.

`class` is another keyword in Java. The keyword `class` simply tells the computer that what follows is the body of a class. If you want to define a class (which we will want to do often), you need to use this keyword.

As we said before, identifiers cannot be keywords. We must name all of our classes with unique names. Java will not allow two classes to have the same name. Furthermore, it is easier for us programmers to keep track of what classes represent if we have a meaningful and unique name for each one.

Names of classes must conform to the rules for identifiers talked about earlier. Remember, the rules are compiler-enforced. Since classes are different from packages, the style we will use for the names of classes will be different than the style for the names of packages. All class names will begin with a capital letter. If we decide that we need multiple words to express the name of the class, we will capitalize each new word in the name. An example is `OurFirstClassName`. Even though there are no spaces in between the words, we can tell that there are four distinct words.

The name of a class also gives us one more important piece of information about our code. It determines the name of the file that the class is stored in. We already said that all of our class files have to have a `.java` extension, but each file also needs a name. We now know that the name of the file needs to be the same as the name of the class stored within it.

Example 5

[12] A class is the only programming entity we know about right now. However, there are other programming entities that we will learn about throughout the course of this text. The rules for access control apply to them as well.

Looking at the example of the EcoSystem class, the code for that class is stored in a file named `EcoSystem.java`. When we run the compiler on that file, it creates the class' bytecode in a file named `EcoSystem.class`.

Class Body

The body of the class is the second part of the definition of a class. The syntax for a class body looks like this:

```
{
  Properties Defined
  Capabilities Defined
}
```

The class body holds the guts of the class, the part where the really fun stuff happens. Note the opening brace '`{`' and the closing brace '`}`'. The braces '`{}`' represent the start and end of the body of the class respectively. We will see the braces quite a bit as we are learning about Java. Java uses the braces as a grouping mechanism, showing which parts of code belong together. The precise placement of the braces is subject to programmer preference. Of course the opening brace '`{`' must come before the segment of code it is enclosing, and the closing brace '`}`' must come after. However, different brace placement styles are used; in this text we follow the style which places the opening brace at the end of the line before the block of code the brace delimits, and the closing brace on a line by itself. The lines of code between the braces are indented. You can see this style of brace placement and indentation in the code example just below.

You will notice quite a few pairs of braces inside class definitions, each of which groups together different pieces of code. Braces must always occur in pairs. If you are missing a '`}`' or a '`{`', the compiler will not compile the program. Putting it all together, we get this basic picture of Java files:

```
package identifier ;

public class identifier {
      Properties defined
      Capabilities defined
}
```

Comparing this to the Java source code for the EcoSystem class,

```
Error! Not a valid filename.
```

we see that not only is the package for this code called `chapter2`, the class is named `EcoSystem`, and the body of the class definition contains the following two lines:

```
Error! Not a valid filename.
```

To properly explain these two lines of code we need to discuss how objects are created.

How do we get objects?

Remember that we want to model our domain using objects. However, for our Java code, we have to formally define the types of objects we want to use by writing classes. Now what we need is a way in our code to actually create objects.

Java provides the keyword `new` for just this task. `new` tells the computer to create an instance of a particular class definition: an object is created in memory. Another way of saying "create a new new instance of a class" is "instantiate a class". How does Java know which class to create an instance of? We tell Java this by using `new` together with a constructor for the class. Recall that we did this in the previous chapter when we wrote,

```
new chapter1.Terrarium()
```

We can say that this expression creates an instance of the chapter1.Terrarium class, or that it instantiates the chapter1.Terrarium class.

A constructor for a class shares its name with the class, so even though we are actually giving the name of the constructor after the keyword `new`, it is also the name of a class. The syntax for a *statement*[13] to create an object looks like this:

```
new ConstructorName();
```

A statement is an instruction that can be performed. Statements end with a semicolon. We will see many different types of statements in our Java files. It is common to put each statement on its own separate line, but that is not necessary for the compiler. However, it is much easier for human eyes to read if you put each statement on its own line, and we consider it good practice to do so.

[13] This code to create an object is more than a statement. It consists of an expression, `new ConstructorName()`, followed by a semicolon. When we created objects in the DrJava interactions pane we typed expressions, not statements. As a result DrJava responded by printing a representation of the value of the expression. If you had typed the statement corresponding to the expressions we wrote, i.e. `new chapter1.Terrarium();` then DrJava would not have printed a value, since statements, unlike expressions, do not have values. We will learn more about expressions in chapter 3.

In this statement, we see the keyword `new` followed by the name of the constructor. The name of the constructor is followed by a set of parentheses '`(`' and '`)`'. The parentheses are an *actual parameter list*. Actual parameter lists have a special meaning in Java that we will discuss in greater detail in chapter 3. For right now, you need to know that the parentheses are needed whenever we are working with capabilities. The constructor is a capability, so we see the parentheses.

Constructors

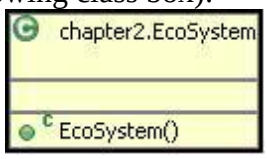
A constructor is a special capability of an object that should initialize the state of an object. When we see the name of a constructor after the keyword `new`, we are actually calling (or invoking) this special capability. Invoking a capability causes the capability to be performed. Therefore, when we invoke a constructor, we initialize the state of the object. Constructors can only be invoked when an object is created (i.e. together with `new`).

A constructor is something that must be part of the source code that we put in our Java file. It is also listed in our UML diagrams. Typically we will list a constructor as the first capability, as its purpose is to set up the initial state of the object:

```
package identifier;

public class identifier {
            Properties defined
            Constructor defined
            Other capabilities defined
}
```

We have seen a constructor already, both in UML (the EcoSystem() capability in the following class box):



UML class box showing constructor

and in code:

```
Error! Not a valid filename.
```

The constructor definition in this code is,

```
Error! Not a valid filename.
```

which we can now properly explain.

Constructor definition

Inside the source code for a class, a constructor definition has two parts:

```
Constructor header
Constructor body
```

A constructor header consists of:[14]

```
public identifier ()
```

We have seen the keyword `public` already; the use of `public` in the constructor definition means the same as it did when used in class headers, namely that there is public access to the constructor. The name of the constructor is an identifier, but as we talked about previously, the name of the constructor must be the same as the name of the class. This is a rule, so it is enforced by the compiler.

A constructor body consists of:

```
{
      List of statements to be executed when constructor is invoked
}
```

The constructor body once again is surrounded by opening '`{`' and closing braces '`}`'. Inside the constructor we write Java code for what the constructor of the class should do. We said that constructors are responsible for setting up the initial state of the object. One of the steps of setting up the initial state of the object is to give initial values to all the properties of an object. We will see how to do this for the first time in a later chapter. The first constructor we look at will be empty inside the braces.

Putting it all together, we have:

```
public identifier () {
      List of statements to be executed when constructor is invoked
}
```

The constructor of our EcoSystem class does not yet contain any statements. Recall that we want the constructor of this class to create one instance of each of the two classes `chapter1.Terrarium` and `chapter1.Ant`.

[14] This is a simplification for now. We will see the full use of parameter lists in chapter 3, which can change the basic header of a constructor.

Note that there are two steps to actually getting an object in our program. We first must write a class definition, and then we must create an instance of that class by using the keyword `new` and calling the class' constructor.

Using Classes from Other Packages

Java is good at finding classes that are defined in the same package. Java has some difficulty if one class tries to use a class from a different package. If we want one class to create an instance of a class that is in a different package, we need to identify that class by its fully qualified name: this is the full name of the package to which the class belongs, followed by a dot and the name of the class itself.

Example 6

Last chapter we created instances of the `Terrarium` and `Ant` classes in the DrJava interactions pane. These classes are both defined in the `chapter1` package. We referred to the constructors of these classes by their full names:

```
new chapter1.Terrarium()
new chapter1.Ant()
```

You can think of this like calling people on the telephone. If you are calling a number that is in your area code, you do not need to specify the area code when you dial the phone. If you are calling a number outside of your area code, you must give the area code explicitly before the phone number.

Therefore, if you are going to use a class from within the same package, you can refer to it simply by its name. If you are referring to a class from another package, you must tell Java not only its name, but what package it belongs to.

The Instantiation Dependency Relationship

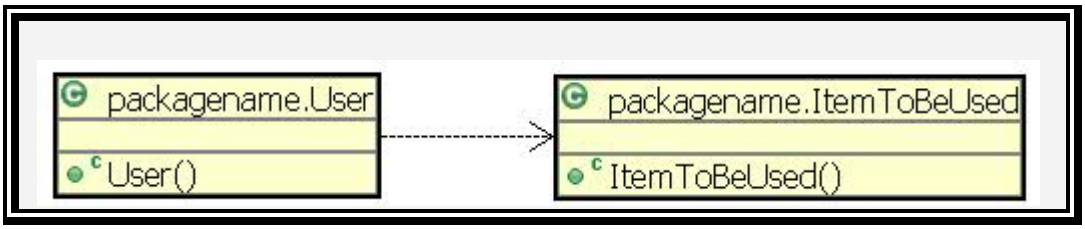
Whenever we talk about relationships amongst the classes in our code, we talk about them in many ways. We use their formal names (for example, *instantiation dependency*) and their informal names (instantiation dependency is sometimes called the *uses a* relationship). We will see them pictorially in our UML diagrams as connections, or arcs, between the class boxes. We will also see the relationship inside our source code by recognizing certain patterns within the code. By the end of our time studying relationships, you should be able to recognize any of the representations of the relationship and understand which relationship it is and what that relationship would look like in the other forms.

As you learn to use languages other than Java, the discussion of modeling and relationships will not go away. What will change is how the relationships are expressed in a different programming language. However, you can still use all the same types of relationships to model your solutions.

The instantiation dependency relationship exists between two classes when one of them creates an instance of the other.

Instantiation Dependency Relationship in UML

Inside a UML diagram, two classes[15] that are in the instantiation dependency relationship have a dashed arc with an arrow head between them. The arrow points from the "user" class to the class it is using. In general, the relationship between the two classes looks like this:



Instantiation Dependency Relationship in Code

If a `User` uses an `ItemToBeUsed`, we will see that there is a line of code that creates an instance of an `ItemToBeUsed` inside the `User`'s constructor. Recall that the code for `User` will be in a file named `User.java`.

```
package packagename;

public class User {
      public User () {
            new ItemToBeUsed();
      }
}
```

How is the code for the `ItemToBeUsed` class affected by this relationship? Actually, this relationship does not change the code for `ItemToBeUsed` at all. Since, for this example, we do not have much information about `ItemToBeUsed`, the definition for that class is shown as pretty empty. Remember that this class is different from the one above, so it will be in a different file. This class will be in a file called `ItemToBeUsed.java`.

```
package packagename;

public class ItemToBeUsed {
            public ItemToBeUsed () {
            }
}
```

We will see this situation, where the code of one of the classes is affected by the relationship and the code of the other is not, with all our relationships. Since the two classes in instantiation dependency play different roles in the relationship, the relationship is called a *directional* relationship. All of the relationships we will study are directional. Relationships

[15] Because this relationship involves two classes, we call it a binary relationship. Binary means involving two components. Thus, we can speak of a binary star (two stars which orbit each other) or a binary digit (a 0 or 1).

that are directional affect one of the classes in the relationship, but not the other. It is convenient to be able to refer to the roles that classes play in a relationship unambiguously. We do this by referring to one class (the affected one) as the source class of the relationship, and the other class (the unaffected one) as the target class of the relationship. In the instantiation dependency relationship we call the class that instantiates (creates) the other the source, and the class that is being instantiated the target.

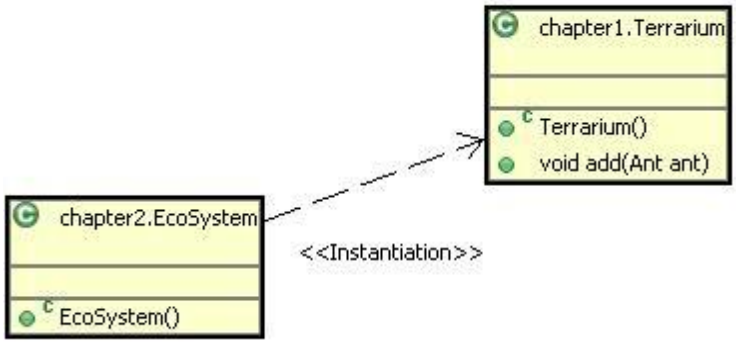
> ***Exercise***
>
> Draw the UML and write the code for the instantiation dependency relationship between a class named `Carpenter` and a class named `Hammer`, so that a `Carpenter` uses a `Hammer` and that both classes are in a package called `construction`.

Extending our conceptual model

Earlier this chapter we introduced the idea of a top level object by saying,

> The top level object of a system has the responsibility for creating, either directly or indirectly, the objects which together make up the system.

Now that we know about the instantiation dependency relationship we can use it to indicate the relationship that we wish to model between the chapter2.EcoSystem class and the chapter1.Terrarium class. In Green, we can draw the relationship from the source class (chapter2.EcoSystem) to the target class (chapter1.Terrarium):



If we look at the code for the chapter2.EcoSystem class, we see that it has been modified to include, in the constructor, the code to instantiate the chapter1.Terrarium class:

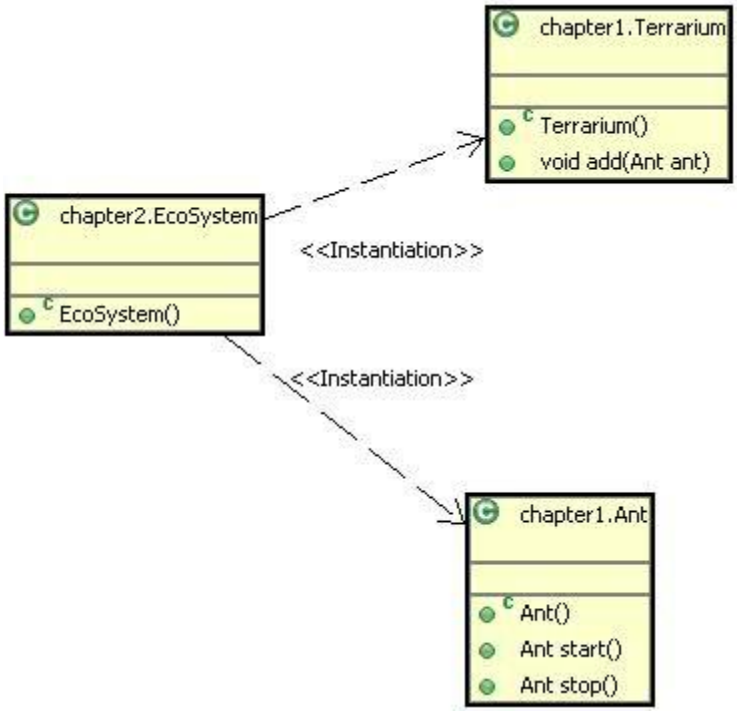
```
package chapter2;

public class EcoSystem {
      public EcoSystem() {
            new chapter1.Terrarium();
      }
}
```

The effect of this change to the code is that now, whenever the chapter2.EcoSystem class is instantiated a new chapter1.Terrarium object will be created as well.

Completing our conceptual model

Continuing on with this example, let us add an instantiation dependency relationship between the two classes chapter2.EcoSystem and chapter1.Ant as well:



The change in the code in the constructor of chapter2.EcoSystem is similar to what we saw earlier:

```
package chapter2;

public class EcoSystem {
      public EcoSystem() {
            new chapter1.Terrarium();
            new chapter1.Ant();
      }
}
```

Green has added a line of code to the constructor which instantiates the chapter1.Ant class.

When we set out working on this example we wanted to create a top level object which would take responsibility not only for creating instances of chapter1.Terrarium and

chapter1.Ant, but also invoke capabilities to make the Ant appear in the Terrarium, and also to begin to move around. In the DrJava interactions pane we had written:

```
new chapter1.Terrarium().add(new chapter1.Ant().start())
```

Note that code generated by Green is not exactly what we want. Green in particular, and UML class diagrams in general, do not show every detail of a program. A UML diagram is an abstraction, and as such it allows us to ignore certain details, details which are irrelevant to us when we want to consider the main players in a system and their relationships. Green does the best it can when generating code, but very often we need to modify the code that Green creates for us to get it to do what we want.

With this in mind, we can change the code in the constructor of chapter2.EcoSystem as follows:

```
package chapter2;

public class EcoSystem {
      public EcoSystem() {
            new chapter1.Terrarium().add(new chapter1.Ant().start());
      }
}
```

Now, if we create an instance of chapter2.EcoSystem everything that needs to happen to set up the system and get it running will happen, because we recorded all the relevant steps in the constructor of this class. In other words, we have now written a program to:

- Create a chapter1.Ant object
- Make the chapter1.Ant object begin moving about
- Create a chapter1.Terrarium object
- Place the chapter1.Ant object into the chapter1.Terrarium object

This sequence of instructions is a program, and it will be carried out whenever the chapter2.EcoSystem class is instantiated.

Summary

In this chapter, we began to translate the models of our problem domain into executable models, i.e. Java code. This is a multi-step process. We began by translating the English descriptions of our objects into UML class diagrams and learned that a class is a formal specification of an object. UML class diagrams help us to visualize the objects in our problem domain and formalize our design. In a UML class diagram we list the properties and capabilities for each of the types of objects in a UML class box.

Once we have our UML started, we can begin to translate the UML into Java source code. Java source code is written in files with a `.java` extension. Each Java file we write contains the definition of one class. The name of the class defined in the file and the name of the file are the same. Each class belongs to a package, which corresponds to a directory in

which the file is stored. Each class needs a constructor, which is a special capability that sets the initial state of the object when it is created. To create an object, we use the keyword `new` and call the constructor of the class we want to instantiate.

We have learned to create a top level object representing a system as a whole. The top level object's role is to create, either directly or indirectly, the objects which together make up the system. Commonly, this creation is achieved through the use of the instantiation dependency relationship.

