Design First With Java

Glenn D. Blank and Sally H. Moritz

Copyright © 2005 (please ask glenn.blank@lehigh.edu for permission to use this material)

Chapter 1: The Big Picture of Software Development

"It is a melancholy of mine own, compounded of many simples, extracted from many objects...."
William Shakespeare, *As You Like It*

1.1 Software development is challenging and cool

Software development is challenging because it's complex. IMHO (web-speak for "in my humble opinion"), many books try to coddle beginners by telling them that programming is simple, just like following a recipe. Just as a recipe describes step by step how to cook an entrée for a meal, so an algorithm describes step by step how a computer should perform a particular task. Let's look at an example, part of a recipe for matzah ball soup:

First, there's a list of ingredients, then the following procedure:
1. In a medium bowl, lightly beat eggs with oil.
2. Add matzo meal, salt and stir until smooth.
3. Stir in water, then let mixture stand for 20 minutes so matzo meal absorbs liquid.
4. Bring salted water to a boil.
5. Carefully slide balls into boiling water.

….

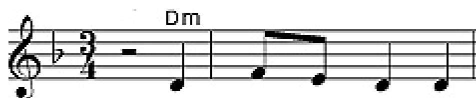
The problem with the analogy is that a recipe is more complicated that than a sequence of numbered steps. First of all, many things are "fuzzy." Exactly what does it mean to "*lightly* beat eggs"? How much salt should be put into the pot before boiling? Actually, the ingredients list, above the procedural part of the recipe, tells us—"a pinch of salt." So, how much is a pinch? Human cooks tolerate *fuzziness*, because it leaves some room for creativity. But computers demand exactness. In addition, the sequence embeds more complex structures. There are many *loops*—such as "stir until smooth," "carefully slide balls," etc.—repeating until some condition has been met. There are *procedures embedded* within this procedure—for example, the cook starts a procedure letting the mixture stand, which ends when a timer indicates that 20 minutes have passed. There is some *parallel processing*—step 2 implies that one is adding matzo meal, salting and stirring more or less at the same time, and step 5 could begin before step 4 completes. There is *sensor* processing—the cook must observe the smoothness of the mix, the boiling of the water, etc. It is unlikely that any robot is capable of such subtle observations. Indeed, the complexity of recipes tells us much about what is challenging and creative about both cooking and software development—programs also have loops, embedded procedures and sensor processing (at least observing where a user points a mouse) and some even have parallel processing. Some people *like* cooking, while others leave the cooking to chefs or Mom! Just think: if something as simple-looking as recipe turn out to be rather complex, consider how complicated it must be to design the software for driving Mars rovers, or real world banking systems, or graphics drawing programs….

The truth is, real world software development is complex. Software systems have rich structures that are not trivial to design correctly, especially in environments that are constantly changing. That's why a great part of the success of the Mars rovers is that the engineers have

been able to reprogram it "on the fly" to meet unanticipated circumstances. That's why some of the most successful, money-making software systems often need fixes, patches, new releases and new versions. That's why software development is at least a trillion dollar industry, the fastest growing industry worldwide for over a generation.

Smart, creative people like doing it. Software development is cool because it's creative. Most software developers enjoy what they do because they like making things that really work, that are useful to other people. Software engineering is how professionals build and maintain large software systems.

Software development is like composing music. Suppose you start with a melody. When you get a melody in your head, you can compose a simple "score" representing the music in a special musical notation. Something like this:



If you can't read musical scores, it's OK. We'll explain, for the sake of exploring the analogy. Even for a simple melody, there is lot of structure. First, there's a *sequence* of notes, starting with the one under the "Dm", expressed in left to right order. That's similar to the sequence of instructions in a recipe or program. But there's more to it than that. The *pitch* of notes is represented in terms of their positions on lines—the five horizontal lines are called the "staff." Since the lowest line on the staff is an E, the first note, below the lowest line, is a D, the second one, just above the E line, is an F. Different shapes of notes represent their relative *timing*. So the first one is a quarter note, while the next two are eighth notes, each of which are half as long as a quarter note. In addition to the individual notes, we must consider the overall rhythm of the whole melody, indicated by the 3 over the 4 on the left. This melody is in "three fourths time," which happens to be the tempo for waltzes. Each measure—between vertical lines on the staff—has three beats, with each beat equal to a quarter note; for example, the second measure has two eighth notes plus two quarter notes, adding up to three beats. Another layer of structure is the "key." To the left of the 3 over 4 is a

Musicians call this notation a "flat". Since it's on the third line from the bottom, it's a B-flat, a pitch halfway between A and B. Since it appears before the melody, it indicates that for the whole melody, whenever we see a B on the staff, we will play a B-flat instead. Musicians recognize that this global B-flat means that this melody is in the "key" of F—but now we're getting into more advanced musical theory. The tempo, key, and other musical qualities are all layered on top of the horizontal structure of the melody. In addition, there's a vertical dimension—the harmony, multiple notes playing together to form chords. The "Dm" above the staff indicates that one can play a D minor chord to accompany this stretch of music. It's amazing how much structure musicians can squeeze into this special notation!

But there's more to music than that what appears on the score. People play the music, either solo or in cooperating groups such as duets, quartets, bands or symphonies. In a rock band, for example, a drummer, guitarist, and bass guitarist have quite different scores. If they are going to play these different scores together well, they have to learn how to communicate with each other, using the score and other cues, to make the music that their fans will hopefully appreciate as music and not noise!

Similarly software development includes algorithms, which at first look like sequences of instructions. But like a recipe, even sequences have rich underlying structures, which make

them not so trivial to compose (not to worry, we'll start by composing simple programs). A programmer translates an algorithm into a program in a particular programming language that a computer can perform, much as a composer translates a melody into a score that a musician (or a computer synthesizer) can perform. Then there are vertical structures, such as programs that depend on other programs to do their job. Finally, there are people who have to work together to create software, use it, and possibly modify it to behave differently or better (such as fixing errors known as "bugs"). Notations such as scores in music, or use cases or class diagrams in software development help people communicate so they work together to solve a problem. Software development always involves at least two people who need to work together to understand and solve a problem: a customer and a developer, and usually many more.

Software development is also like designing and building a house. It's definitely not just *coding*—writing the instructions telling the machine what to do. Please don't be in a hurry to write code. The first challenge of real world software development is *analysis*, understanding what a customer wants. The second challenge is *designing* a solution. The third challenge is *implementing* the design in a programming language code. The fourth challenge is *testing* to make sure the program really does what the customer asked for. Software development also involves a lot of *documentation*. Like scores in music, or blueprints in house constructions, software documents facilitate communication, both during development and while customers are using the software and developers continue to *maintain* it.

Imagine if people started building a house by immediately going out to a site with some nails and lumber. What about digging a foundation? But how would you know what kind of foundation to dig, let alone what kind of rooms to frame, without an architectural plan? The customer and the contractor must first agree on the kind of house to build, an architect has to design it, a bank must agree to finance it. Only after they agree on a good overall design and financing can the contractor begin to hire subcontractors to dig, then frame, install electrical, plumbing, heating systems, etc. Like house construction (let alone skyscraper construction), constructing a software system is complex, creative, and hopefully satisfying.

Let's explore the analogy between house construction and software development further. A developer must understand the *requirements* (exactly what the customer wants the house to look like, or what the customer wants a program to accomplish) and estimate how much it will cost (software development is often riskier than house construction). We call the first and very important phase of software development *requirements analysis*. Then a developer must *design* the house (create an explicit blueprint for the architecture or structure of the house or program). A developer must *implement* the design (a contractor commences digging the foundation, a lead programmer begins figures out how to translate design into code). A developer must *test* the code, making sure it fulfills the design and meets the requirements. Finally, a developer must help other developers *maintain* the software, making those fixes and patches or other improvements that users need and expect. Each step in the process has documents; good software developers document as they go along, not after the fact, because they know that they need to communicate with other software developers, often separated widely in space and time.

In this chapter, we will give you a birds-eye view you can be the developer, performing each of the steps. The following chapters will provide a bit more detail.

Exercise 1.1: Suppose your parents decide to have a new house built for your family. They start the process by going to see a contractor. What do your parents, the contractor, or others need to

do? In one column, list all the tasks that need to be completed, in the order they must be done. In another column, list analogous tasks that software developers would need to do to construct a software system.

1.2 Object Orientation

Computer scientists have come up with a neat way to think about how software can work: as lots of little objects performing behaviors, just like objects behave in the real world. After all, everything in the real world is an object (a house, a car, a person). You learned how to recognize objects when you were an infant, so it comes naturally. Only now, you need to think explicitly about how you recognize objects, in order to discover the objects that a software system will model.

How do you identify individual objects in the real world? Objects may have different characteristics or *attributes*. For example, the person I am thinking about, my daughter, has black hair, her name is Abby, and her social security number is, well, I think I won't tell everyone that…. Objects may also perform different *operations* which we may observe as their *behaviors*; for example, Abby can smile (very cutely), she sings, she brushes her hair, etc. Or take a book: its attributes include its title, number of pages and selling price; its behaviors include showing its title and its author(s), opening to a sequence of pages, etc.

Objects that share the same attributes have different values for those attributes. For example, everyone in your class is a person, but each person has a different name, hair color, social security number, etc. The particular set of values for each object represent its *state*. So the state of the book includes its title, author, and the particular page where it is now open. Another book may have a different title and author and may not be open at all at the moment.

Objects that share the same characteristics belong to the same category, or *class.* For example, each object on my bookshelf belongs the class Book, while each individual in your classroom belongs to the class Person. (Software developers like to spell their class names with an initial upper case character.) Individual objects (each person in the room, for example) are also called *instances* of the class.

Objects and classes come naturally—you learned how to recognize them before you were a year old. Every day, you recognize thousands of objects, many of which you hadn't seen before. It helps that you know how to recognize objects as instances of general classes. So, if something large comes rolling down the road on four wheels, you quickly recognize that it is probably a car. You also have the capability of creating new categories or classes of objects. Maybe that car is a model you've never seen before, so you look for it's model name on the front or rear, or maybe it's not a car at all, but a truck, or a chariot of fire!

Nevertheless, object-orientation is not a simple as that. (I hope you don't mind my harping on things being more complicated than they first appear. Complexity is interesting.) People can cope with the fuzziness of the everyday world much better than computers. For example, one of the attributes we expect an elephant to have is the color grey. But if you happen to encounter an albino elephant (albinos lack pigment so they are white), you would still be able to recognize it as an elephant. Fuzziness or vagueness in natural languages like English is good, because it allows people to communicate about somewhat unanticipated or hard-to-describe things or situations, such as a new style of clothing, or the experience of falling in love, or things that are spiritually discerned. All this experience is way beyond computers, at least for now. Software developers have to figure out how to translate the imprecise world of their customers into the increasingly precise requirements, design, and program code of machines. This takes

skill and diligence. Musicians, cooks, and software developers develop their talent with lots of practice! Eventually, they can create wonderful or useful things—maybe you can, too?

1.3 Use cases

We said that the first step in developing a software system is understanding what the system will do—gathering its requirements. One way to understand a problem is to describe it from the point of view of a *user* using a system that solves the problem. (Users might already be solving the problem by hand or with a less effective program.) A **use case** describes one or more scenarios for using a system, tied together by a common user goal. Each use case gives a step-by-step description of how different "actors" (including the system) might interact to solve a problem. For example, we could describe how a user interacts with an automated teller machine in terms of use cases:

Use case: log in
Actors: customer, ATM, Customer Database
1. Customer submits a card
2. ATM reads customer ID from card
3. ATM sends customer ID to Customer Database, asking for PIN
4. Customer Database returns the correct PIN
5. ATM asks customer to enter 4-digit PIN
6. Customer enters a PIN (personal identification number)
7. ATM compares PIN entered with the correct PIN
8. PINs match
9. ATM displays a menu of possible transactions

Alternative: PINs don't match
8b. ATM tells customer that PIN is invalid; please try again
Go to step 6

This use case has the title "log in." There are three *actors* in this use case: the customer, the ATM and the Customer Database. (Note that actors need not be human; they are individuals who interact with each other in predictable ways.) The use case describes the login *scenario*, step-by- step. It also provides one *alternative* scenario, branching off from step 8, if the PINs don't match.

Exercise 1.2: Can you think of another alternative that could happen in this use case?
Exercise 1.3: A use case may have a step that deserves to be expanded into another use case. Can you find such a step in the above use case?
Exercise 1.4: Develop a use case with at least one alternative for withdrawing cash from the ATM.

What's the big deal with use cases? Because a use case is written from a user's point of view and in a user's language, it can be very useful as a means of communication between an systems analyst and a user. Once the analyst is confident that she understands what the user wants, she can specify the requirements of a system in a way that is precise enough for software developers to design and implement it. Use cases will also provide a way to test the system: among other things, the scenarios described in the use cases should work as advertised.

Let's consider one more example. A customer wants us to create an animated shapes program. For starters, it should be able to draw various geometric shapes—we'll start with circles, squares and triangles—on a canvas (a frame on a computer screen). Now the animation: it should also be able to move these shapes around and change their sizes and colors, dynamically. For example, after creating a picture of a house with a sun overhead, we'd like to show the sun setting—changing colors and sinking below the bottom of the canvas frame. Cool?

Let's describe what we want in terms of use cases. Here's one to start with:

Use case: Create a circle
1. User requests a circle
2. System creates a circle with default diameter, default X,Y coordinates, default color
3. System draws the new circle on the canvas, using default values

"Create a circle" describes the behavior of a *constructor*, which creates an instance (i.e., a single object) of a class. This use case is pretty simple: I couldn't even think of any alternatives. It simply assigns default (i.e., initial) values to a circle's attributes, then draws it on the canvas. Here's another use case:

Use case: change size of a circle
1. User supplies a new diameter
2. System replaces the value of this circle's diameter with user's new value
3. System erases the current display of this circle on the canvas
4. System draws the circle on the canvas again

Exercise 1.5: Write a use case that moves a circle to the right by a default distance, 20 pixels.
Exercise 1.6: Write a use case that moves a circle to the left by a distance given by the user.
Exercise 1.7: Write a use case that changes the color of a circle.
Exercise 1.8: In Knobby's World (from *The Universal Computer*), write a use case describe the behavior of the move button. Note: there is an alternative scenario.
Exercise 1.9: In Knobby's World, write a use case describe the behavior of the read button. Note: there is an alternative scenario.
Exercise 1.10: What parts of the Knobby's World interface involve use cases that are a bit simpler than those of the previous two exercises? Why? What parts involves use cases that are more complicated? How so?

1.4 The Eclipse integrated development environment (IDE)

Nowadays, most developers use tools to help them create software. One useful tool is an *integrated development environment* (IDE), which typically combines an editor for creating and modifying source code of programs, a compiler for translating source code into executable machine code, a debugger to help figure out why a program doesn't work quite as intended, as well as tools to help with design, testing and documentation—all conveniently available via the click of a mouse. Many companies sell and support IDEs and there are also a few that are free, supported by an "open source" community. Eclipse is a free IDE for multiple programming languages (especially Java) and object-oriented design, spearheaded by IBM and widely supported as an open source project. One way that developers can contribute code easily to the Eclipse project is in the form of *plug-ins* that add tools to the IDE. You will learn how to use

two plugins: one for object-oriented analysis and design (UML) and another for object-oriented programming, especially for beginners (DrJava). At this point, if you have your own personal computer connected to the Internet, I recommend that you get started installing Eclipse. Point your browser at www.lehigh.edu/stem/teams/dieruff/ for step-by-step instructions.

1.5 Class diagrams

When you've done the installation and run Eclipse, you should see, on the right, a class diagram for the shapes problem described above, consisting of four classes (in boxes):

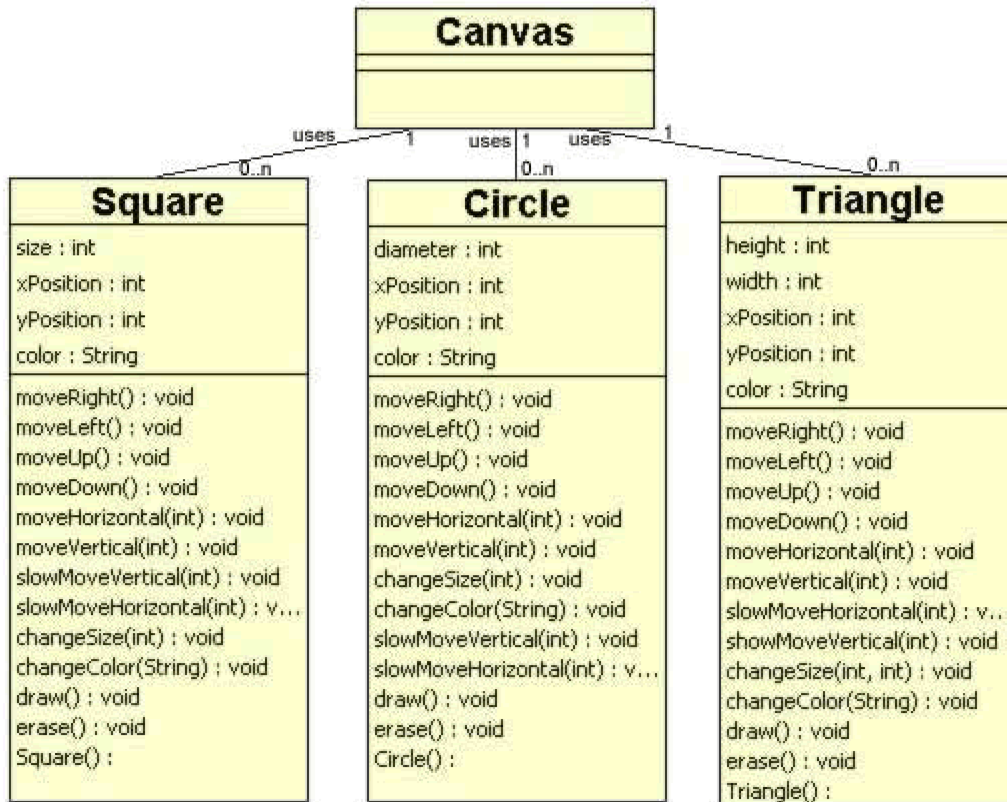


Figure 1.1: UML class diagram for the Shapes project

This diagram represents a high-level design solving this problem, using a notation called *Unified Modeling Language (UML)*, which has become the standard for object-oriented analysis and design. The Circle class (like Triangle and Square) connects with the Canvas class via a simple line. We call this an *association* relationship, the most general kind of relationship connecting two classes. To make the nature of this more specific, we have labeled this association "uses"—i.e., Circle uses Canvas. The "0..n" notation at one end describes its *multiplicity*, meaning that there can be zero or more Circle objects in this association, while the "1" notation at the other end means that there should just be one instance of Canvas when the program is running. In other words, you can have any number of shapes on a unique canvas.

Exercise 1.11: Suppose we wanted to add the ability to draw any polygon on the canvas. How would you modify the high-level class diagram above to make it happen? Describe the shapes,

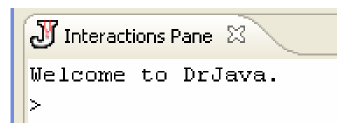
lines and labels that you would add and how they would relate to those already in the diagram. (For now, you can ignore everything in the middle and bottom sections for your new class box.)

Classes have an internal structure. As you can tell from the class diagram, a class has three parts: the class *name* in the top section, *attributes* in the middle section, and *operations* in the bottom section. In this case, there are four attributes, each of which hold *data* of a particular *type*: `color` is a `String`, while diameter, `xPosition` and `yPosition` are of type `int` (integer). As well see, each instance (object) will hold particular values in these attributes. There are many operations, each of which end with `()`. It so happens that all three of our shapes classes have nearly the same operations, such as `moveRight()` or `changeSize()`. The one difference is the operation at the bottom of each class, called the *constructor*, which happens to have the same name as the class, such as `Circle()` for class `Circle`. Each class must have its own constructor.

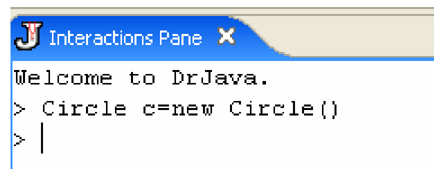
Exercise 1.12: Why does class Triangle have more attributes than Circle or Square?

1.6 Experimenting with DrJava

Let's check it out. At the bottom of Eclipse IDE you'll find DrJava's *Interaction Pane*:



Following the welcome message, you enter Java expressions where it prompts you with >. Try typing an expression shown below, then press the Enter key:



When you enter some code, DrJava's Interactions Pane tab turns blue. But something more interesting happens. Voila! A blue circle appears in another window. (If you don't see the new window, the Eclipse window may be obscuring it—try holding Alt key then pressing the Tab key to bring it on top. You might to drag the new window someplace else on your screen so it won't hide behind the Eclipse window again.) How did this happen? Well, we have to give credit to the folks who created DrJava, making it easy to interact with Java code. The specifics are in the code we entered above. The expression `new Circle()` invokes a constructor, creating an instance of class `Circle`. The code `Circle c=` *assigns* the new instance to a *variable*, `c`, whose type is `Circle`.

Exercise 1.13: Write Java code that will construct a square. Enter this code at the DrJava prompt. What happens?
Exercise 1.14: Create a second instance of `Circle`, naming it `c2`. What did you enter? Why is it important to name it `c2` rather than simply `c`? (You might want to try creating another `Circle` called `c` to see what actually happens.)

Once we've created an instance of a class, we can interact with our new object by invoking operations on it——in Java, the code implementing UML operations are called *methods*. For example, if we enter `c.moveRight()` at the DrJava prompt (>), the blue circle moves. (If you did the previous exercise, why is moving the circle a good idea?) So an operation or method executes some code that may change the state of one or more object. In this case, `c.moveRight()` changes the `xPosition` attribute of `Circle c`, altering its horizontal location. The notation object.method() is how we invoke methods on objects in Java. Since `c` is an object, an instance of class `Circle` which we just created, we can invoke any `Circle` operation on it, to make it change its state in different ways. For example, `c.moveRight()` changes the `yPosition` attribute, altering its vertical location.

Exercise 1.15: After creating a second `Circle`, what do you see in the window? What method might you want to execute to see two separate circles? How do you invoke this method in DrJava? Hint: you need to use object.method() notation.

If we enter `c.changeSize(100)` at the DrJava prompt (>), suddenly the blue circle gets a lot bigger! The value, `100`, between the `(` and `)`, is a *parameter*, which we gave this method to tell it how large we wanted the circle to become.

You can learn more about methods and their parameters with the Eclipse UML editor. Right-click anywhere on the box for class `Square`, then select Properties, then click on the Methods tab. Click on `changeColor` in the methods list and you'll see this dialog box:

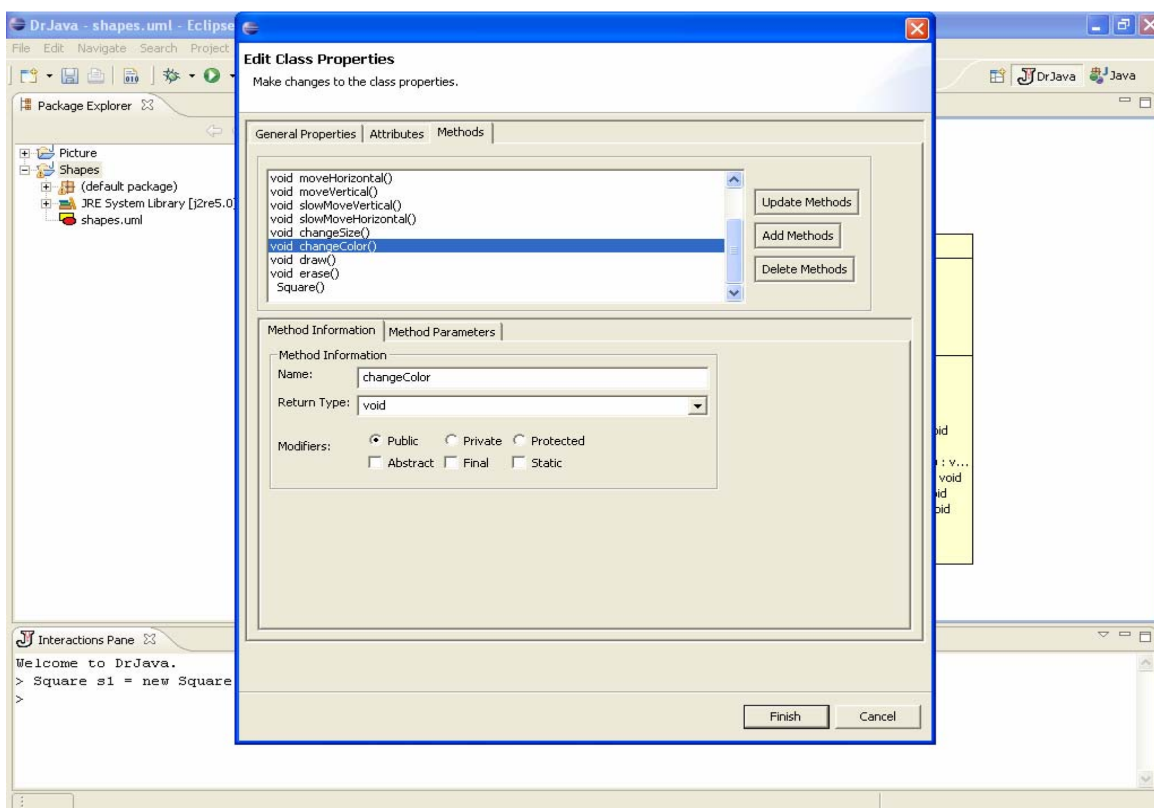


Figure 1.2: Viewing (or editing) the details of a method

The Method Information section shows us the method's name and its return type. The word "void" means that this method returns nothing. (We'll talk about getting data back from a method in a future chapter.) We also see here that "Public" is clicked on. Public means that anyone can call this method for a `Square` object.

If you click on the "Method Parameters" tab, you'll see `changeColor` has one parameter, `String newColor`. In other words, it uses one *formal parameter*, called `newColor`, of type `String`. So, when you say `s1.changeColor("magenta")` , the Java machine will pass "`magenta`" to method `changeColor` as the value of the formal parameter `newColor`.

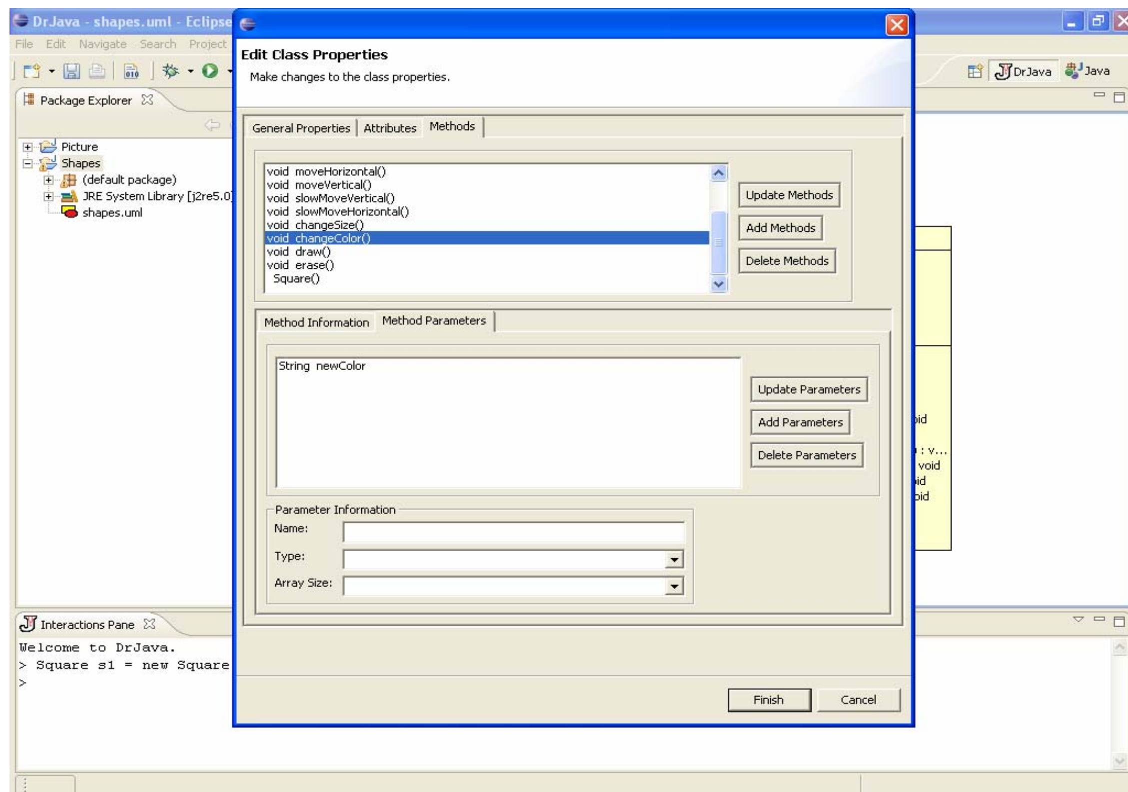


Figure 1.3: Viewing (or editing) a method's parameters

We've now seen two examples of parameters passed to methods: one a series of characters, another a whole number. Notice that the series of characters had to be in double quotes. Numeric data must not be enclosed in quotes.

Let's look at Java data types a bit more. The most common datatypes we will be using are:

> `int` – integers or whole numbers, both positive and negative, such as `0`, `27`, `-932`.
> `double` – real numbers, such as `0.0`, `-12.2`, `27` (or `27.0`).
> `char` – characters, specified between single quotation marks, such as `'a'`, `'%'`, `'7'`.
> `String` – a sequence of characters, specified between double quotes such as "`magenta`", "`U.S.A.`", or "`0.0`". Even "" is a String – the empty, or null, String.

When you pass a parameter to a method, you must make sure you are passing the correct type. If you call `changeColor`, its parameter MUST be in double quotes, because `changeColor` is defined to accept a `String` as a parameter. Trying an expression like:

```
s1.changeColor(12);
```

will cause a *type error*. Forgetting the double quotes, like this:

```
s1.changeColor(blue);
```

will cause a different error. Java will think `blue` is the name of a variable; unless you've defined it elsewhere in the program, you'll get an error that says something like "Undefined class blue."

Suppose you pass 3.14 as a parameter to `moveHorizontal`, which needs an `int` as a parameter. Do you think you'll get an error message? Actually, Java does a *type conversion*. It changes `3.14` into an `int` by chopping the decimal part. The problem with this is even though the statement will execute, it might not do what you intended.

Exercise 1.16: Invoke a method that changes the circle `c`'s size. What did you enter?
Exercise 1.17: Invoke a method that changes the circle `c`'s color. What did you enter? If you got an error messages along the way, what were they and why did they happen?
Exercise 1.18: Invoke a method that moves circle `c`'s location to the left, using two different methods. What did you enter? Hint: the second method takes a parameter that specifies how many pixels to move the circle.
Exercise 1.19: What happens when you try entering the following statements? Explain each one.

```
c.moveDown();
moveDown();
c.moveVertical();
c.moveVertical(100);
```

Exercise 1.20: What's the difference between `moveUp()` and `moveVertical()`? Write Java expressions using each of these methods and describe the behaviors that occur.
Exercise 1.21: Why stop with circles? Create a Triangle. What did you enter? What happens on the screen?
Exercise 1.22: Try changing the size of the Triangle you just created. Did you get an error at first? If so, what did it say? Can you figure out how to make the method work by looking at its code? Or you can right-click on the method, then select Properties from the menu, to get useful information about the method without having to study the code. Do you find what you need here?

1.7 Viewing Java source code in an editor

How do these constructors and methods actually work? To find out, we need to take a look at the source code. If you double-click on operation `Circle()`, the source code implementing it will appear in an editor[1]—a window in place of the UML window within Eclipse. Actually, what you will probably see at first is just the comment:

[1] The Package Explorer provides another way to view source code in the Eclipse editor: for example, double-click on Shapes (or single-click on the little box with a minus sign to the left of Shapes), then double-click on (default package), then double-click on Circle.java—you should now see the source code of Circle.java in the editor.

```
/**
* Create a new Circle at default position with default color.
*/
```

A comment, between /* and */ describe what the following operation does, in English. If you scroll click on the scroll bar on the right side of this window, you'll see the actual Java code[2]:

```
public Circle()
{
      diameter = 30;
                 = 20;
      yPosition = 60;
      color = "blue";
      draw();
}
```

The first thing you notice is that the constructor is `public`, which means you can invoke it outside the class. That's why we were able to invoke the constructor outside the class, in DrJava, to create new instances. The body of the constructor initializes the four attributes of a new `Circle` to default values: its diameter is 30 pixels (picture elements) and its center starts out 20 pixels by 60 pixels from the top-left corner of the canvas, and its color is blue. Then it invokes method `draw()`, so that it will actually appear on the canvas.

While we're looking at source code, let's look at the code implementing a method. Press the key combination Ctrl-F to bring up the Find/Replace dialog box, then enter "moveRight". The editor should find this code:

```
public void moveRight()
{ moveHorizontal(20);
}
```

This method just passes the buck to another method, `moveHorizontal()`. The one difference is that `moveHorizontal()` takes a value between the parentheses—`20`. We call this value a *parameter*—some data that one method passes to another. Here's the source code for `moveHorizontal`:

```
public void moveHorizontal(int distance) {
      erase();
      xPosition += distance;
      draw();
}
```

Method `moveHorizontal()` requires one *parameter*, `distance`, which must be of type `int`. When `moveRight()` calls `moveHorizontal(20)`, it passes the value `20` as an actual value for the formal parameter `distance`. In the body of `moveHorizontal()`, the += operator adds the value of `distance` to the value of the attribute `xPosition`, changing the state of this particular `Circle` object.

[2] Thanks to Michael Kölling of the BlueJ project for letting us borrow the shapes code.

Exercise 1.23: Does the code for the constructor correspond to the use case, Create a circle, shown above? If so, which lines of the use case correspond to which lines of the code?
Exercise 1.24: Does the code for the constructor correspond to the use case, Create a circle, shown above? If so, which lines of the use case correspond to which lines of the code?
Exercise 1.25: Look at the code for `changeSize()` in the source code for class Circle. Does the code for this operation correspond to the use case "change size of a circle" shown above? If so, which lines of the use case correspond to which lines of code?
Exercise 1.26: What's are some differences between the use case notation and the Java code? Also discuss the differences in the purpose and intended actors interpreting these two notations.

1.8 Conclusion: the Big Picture

Remember, the goal of this chapter was just to give you the big picture of software development. In subsequent chapters, you will progressively learn more details, and get more experience using Java. It takes a lot of practice to learn how to play (let alone compose) music well, or to construct the frame of a house (let alone manage the whole construction process or architect the design). So you'll learn software development incrementally, as you learn how to use additional tools of the trade and constructs of the Java language.

Exercise 1.27: Using the shapes classes, you can draw a scene with a house (with a roof, a door, and at least one window), and a sun overhead. First describe your method as a use case. Most of the steps in your use case will involve constructors and methods of Circle, Square and Triangle. Follow the steps in your use case within DrJava and see if your use case works. If not, modify your use case, so that it reflects your actual procedure for designing the scene.
Exercise 1.28: A problem with drawing the clown in DrJava is that you can't easily reproduce it. We've already implemented our own version of the previous exercise in a class. You can see our work by opening the Picture project, using the Package Explorer in the left column of Eclipse, by clicking on the '+' to the left of Picture. Once you've done so, enter the following in DrJava:

```
Picture p = new Picture()
p.draw()
```

Now use the editor to look at the Java source code for class Picture and answer the following:

(a) Why didn't the picture appear when you invoked the constructor?
(b) Method `draw()` initializes a `wall`, `window`, etc. Where are these variables declared?
(c) How does `draw()` use `moveVertical` and `moveHorizontal` to position the `sun`? How could you change the position of the `sun` so it appears to the left of the house?
(d) Suppose you wanted the picture to appear when you invoked the constructor. How could you change the code to make this happen?

Exercise 1.29: Modify `Picture` so that it has two smaller windows and a different colored roof.
Exercise 1.30: Let's make the picture come alive by *animating* a sunset. Modify the `Picture` so that it makes the sun set (changing color and dropping to the horizon).
Exercise 1.31: Using the shapes classes, draw a picture of a clown, wearing a dunce cap. First describe your method as a use case. Most of the steps in your use case will involve constructors and methods of Circle, Square and Triangle. Follow the steps in your use case within DrJava and see if your use case works. If not, modify your use case, so that it reflects your actual procedure for designing the clown.
Exercise 1.32: This time, put all the steps of your use case for drawing a clown in a method, by modifying the `draw()` method of class `Picture`. Extra credit: make the clown smile or wink!