Chapter 1: Introducing Object Oriented Problem Solving

Pedagogic Motivations

- Introduce a problem-solving process by analogy with writing processes.
- Introduce the concept of an object as a bundle of properties and capabilities.
- Demonstrate how to create and interact with an object.
- Introduce object oriented models as systems of interacting objects.
- Engage students in thinking about problem-solving in terms of objects.

Preliminaries

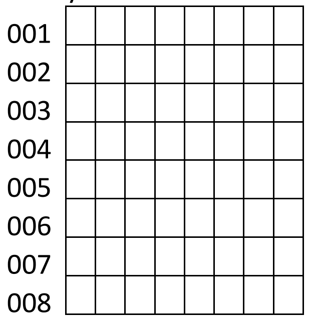
This text introduces you to the field of computer science. The computer science field is very broad, offering many different areas of specialization. Whatever your interests are, whether they lie primarily in hardware, software or even theory, the foundation this text provides will serve you well in your future studies.

Hardware

A computer is, at its core, a fairly simple machine. It is able to follow simple instructions to process data. It consists of two main components, a processor and memory. The processor follows instructions, while the memory stores both the instructions that the processor follows, and the data that the processor works with.[1]

The memory of a computer consists of a sequence of individual memory cells. Each memory cell can store one of two values. Physically these two values are different voltage levels, high and low. We can think of these voltage levels as representing any two values; common values are **true** and **false**, or **0** (zero) and **1** (one). Thinking of a memory cell as storing a 0 or 1, it is usual to say that it stores a bit of information. The word "bit" is a contraction of "**b**inary dig**it**". In our usual decimal (base 10) number system, we use ten digits (0, 1, 2, 3, 4, 5, 6, 7, 8 and 9) to form numbers. In the binary (base 2) number system, only two digits are used (0 and 1).

The memory cells in a computer are organized into groups of eight memory cells. Each group of eight bits is called a byte. In a typical computer bytes are arranged into a linear sequence, in which each byte has its own address, much like houses on a street have their own distinct address. We can visualize a small part of a computer's memory like this:



[1] Equivalence of program and data.

In this picture, each row of the table represents a byte. Each row has eight cells, each one representing a bit. Each bit can store either a 0 or a 1. The numbers along the left edge of table represent the addresses of the bytes. This table shows just eight bytes; a typical computer these days has one megabyte or more of memory. One megabyte of computer memory[2] consists of 1,048,576 bytes!

A byte consists of a string of eight bits. Each of these bits can be 0 or 1. There are 2^8=256 distinct strings of eight bits. Here are a few:

1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1
1	0	0	0	0	0	1	0
1	0	0	0	0	0	1	1
1	0	0	0	0	1	0	0

These bit strings can be used to represent different things. If we choose to think of them as numbers, we can interpret these bit strings as numbers in base 2. If so, these five strings represent the numbers 65, 66, 67, 68 and 69. But we can interpret these bit strings in other ways too. If we interpret them as so-called ASCII characters, they represent the characters 'A', 'B', 'C', 'D' and 'E'. We could, if we wanted, choose to interpret these bits strings as kinds of pets, say dog, cat, bird, fish and reptile. What is important for us to keep in mind is that while the hardware of a computer uses high and low voltages in its computations, we can interpret patterns of bits in whatever way is useful to us.

Software

Computer processors understand very simple instructions. For example, a typical instruction causes the processor to move the contents of a byte from one memory location to another, or adds the contents of two bytes, interpreting their contents as binary numbers. Writing useful pieces of software, like a word processor, a web browser, an MP3 player, or a payroll system, cannot be done effectively by giving instructions to a computer at such a low level. Instead, software is written in so-called high-level programming languages. High-level programming languages allow complex sequences of processor instructions to be expressed succinctly.

In this text you will learn how to approach writing computer programs in a modern, high-level programming language. The computer programs in this text are expressed in a language called Java. The skills you will learn are not specific to this programming language, but can be applied to any programming language you use in your career.

Problem Solving

Every day we are confronted with many problems that we must solve. Most of the problems we face, such as what to wear to school or what to eat for breakfast, are fairly easy for us to solve. Sometimes the problems are more challenging for us: think of the starred problems in a calculus or physics textbook. However difficult a problem appears to be, it can be solved by applying a systematic approach. Most of the time we do not think too much about how we solve a particular problem. Although we learn many different ways of solving problems when young, we internalize them and use them as adults, often without really thinking consciously about what we do.

[2] See "mebibyte".

When faced with trying to solve a problem with a computer, a common lament is “I don’t know where to start.” This text teaches a process for writing computer programs to solve problems, starting from a description of the problem.

Computers: General Purpose Problem Solvers

It is widely believed by computer scientists that a computer program can solve any problem whose solution can be expressed as a finite sequence of instructions.[3] This is known as the Church-Turing thesis, named after two mathematicians, Alonzo Church and Alan Turing. This is an important result because it means that any problem that can be solved in a mechanistic way can be solved by a computer which has been programmed appropriately. We mention this now because it tells us that what you will learn in this book is not relevant only to computer science, but more broadly to solving all manner of real-world problems.

A very important result in computer science is that there are problems which no computer (or other mechanistic process) can solve. This is the notion of uncomputability. We will not explain how to prove this here, but it this tells us that no matter how powerful a computer we have, there will always be problems we cannot solve.[4]

These results together motivate learning how to express solutions to problems as computer programs. If a problem is solvable, a solution can be expressed as a computer program. Computers are already used to solve many real-world problems. You probably use more computers each day than you realize. Some computers are very visible, like laptop computers. Other computers are less visible; cell phones and digital cameras are computers running programs. Many appliances, like dishwashers, ranges, and refrigerators, are controlled by computers running programs. Cars contain many different computers, each running a different program. Some cars can even park themselves, and there are vacuum cleaners that can vacuum by themselves. When you take out money at an ATM or use pay-at-the-pump at the gas station, you are using a computer running a program. As computing technology becomes smaller and less expensive, it is replacing mechanical or biological (human) systems throughout society.

In short, we want to learn how to solve problems with computers because computers are general-purpose problem solving machines. Given an appropriate set of sensors and actuators, they can solve problems in the physical world. They can also be used to simulate the world when acting in the world is too expensive (crash testing a new car design) or dangerous (testing the design of a new nuclear reactor), or impractical (simulating the worldwide effects of global warming).

[3] This is a very informal and loose statement of the Church-Turing thesis.

[4] An example of a problem which cannot be solved in the general case is called the halting problem. This problem is to determine whether or not a program will terminate on all input. It can be shown that there can be no program which will take as its input any program and answers “yes” if it will halt on all input, and “no” otherwise.

Another example is the problem of determining whether two computer programs are equivalent: whether they compute the same output on the same input. It can be shown that there can be no program which will take as its inputs any two programs and answers “yes” if it the two programs are equivalent, and “no” otherwise.

Problem Solving Process

An important part of the problem solving process you will learn in this book is the idea that to solve a complex problem we can first come up with a rough solution, and then refine it several times until we arrive at a more polished and complete solution. In fact, this book teaches the problem solving process in this way: this first chapter presents the process in a fairly rough way, while later chapters refine the process.

To help give some context for the process, we first talk about an important problem solving tool, abstraction. Next we demonstrate a problem solving process that you are probably already familiar with, to show how abstraction is used to make solving large problems more manageable. Finally, we make the first presentation of our object oriented problem solving process.

Object Oriented Problem Solving

This book teaches problem solving with computers in an object oriented way. There are many ways in which computers can be programmed to solve problems. The object oriented way is just one of many. It is the approach used in this book because it is a very powerful and flexible way of thinking of problem solving. The object oriented approach makes it (relatively) easier to think about the problems we want to solve in terms that make sense to people, rather than in ways which make sense to computers. This is not to say that object oriented approaches can solve more or different problems than other approaches: the Church-Turing thesis claims that this isn't possible. Rather, our object oriented solutions will likely have a more direct connection with the original problems than other solutions would have. This makes them easier to express and to understand.

Simplification through abstraction

The word "abstract" strikes fear into many people because there is a perception that things which are abstract are difficult. Where this comes from we don't know for certain, but perhaps it's from abstract art, whose meaning can be difficult to grasp. Whatever the root of this perception, the process of abstraction does not make things more difficult, but rather it makes things simpler. Abstraction allows you to ignore irrelevant details. This means that the more abstract something is the simpler it is.

In fact, abstraction is a technique that you probably use every day. It is a very human way of dealing with complexity. For example, think about how you might describe how the human body works. You would probably not start by describing how it works at the cellular level. Instead, you would start be describing major systems of the body (e.g. the digestive system, the cardiovascular system, the nervous system, and others). Your first approximation of a model of the human body might therefore be something like this:

- Digestive system
- Cardiovascular system
- Nervous system
- etc.

At this high level you might describe some of the ways in which these systems interact with each other. For example, the nervous system indicates hunger when blood sugar levels fall too low. When a person eats, their digestive system converts food into nutrients which are picked up by the bloodstream, which causes the nervous system to signal that the body is satisfied.

Each system in this high-level view can be refined all the way down to the cellular level. By refining from the abstract (simplified) view to the concrete (detailed) view, the details of each subsystem are presented in context, and independently of the details of the other subsystems. This makes the whole system easier to make sense of. We will want to do something similar when understanding and building computer programs.

A familiar problem solving process

Let us now take a look at a problem solving process, one which you are likely familiar with, which relies to abstraction break a large problem into manageable subproblems.

At some point in your schooling you probably had to write an essay or report. You have probably been taught that a good way to help you develop and organize an essay is to start with an outline of the essay, and refine it in several steps until the various sections of the essay can be written.

As an example, consider using this approach to organize an essay about the effect of technology on democracy. A top-level outline might look as follows:

I. Introduction

II. What is democracy?

III. What is technology?

IV. How does technology interact with democracy?

V. Effects of technology/democracy interaction

VI. Conclusions

This top-level organization is not very detailed, but it does two very important things. First, it highlights the main components of the essay (e.g. it tells us that both democracy and technology need to be defined for the purposes of this essay). Second, it gives an order for the sections of the essay. For example, it shows that both democracy and technology must be defined before their interaction can be discussed.

This outline needs to be refined before the essay can be written. Each section can be refined, independently of the others. As an example, after some more refinement the essay outline might well look as follows:

I. Introduction

II. What is democracy?

- A. One person, one vote
- B. Secret ballot
- C. Freedom of expression

III. What is technology?

- A. Encryption
- B. Audit trails
- C. Electronic eavesdropping
- D. User interfaces

        E. The Internet and the WWW
    IV. How does technology interact with democracy?
        A. On-line voting
        B. Verifying identity
        C. Privacy
        D. Anonymity
    V. Effects of technology/democracy interaction
        A. Greater voter participation in elections
        B. Is the secret ballot secret?
            i. Vote buying
        C. Election integrity
            i. Losing votes
            ii. Phantom votes
            iii. Tampering with election results
    VI. Conclusions

This outline clearly contains more detail. Refining it further may well force a re-ordering of topics. Eventually the essay can be written. A well-written essay introduces ideas in a logical order and draws well-supported conclusions. A well-written essay clearly doesn't appear out of thin air – it is carefully constructed step by step.

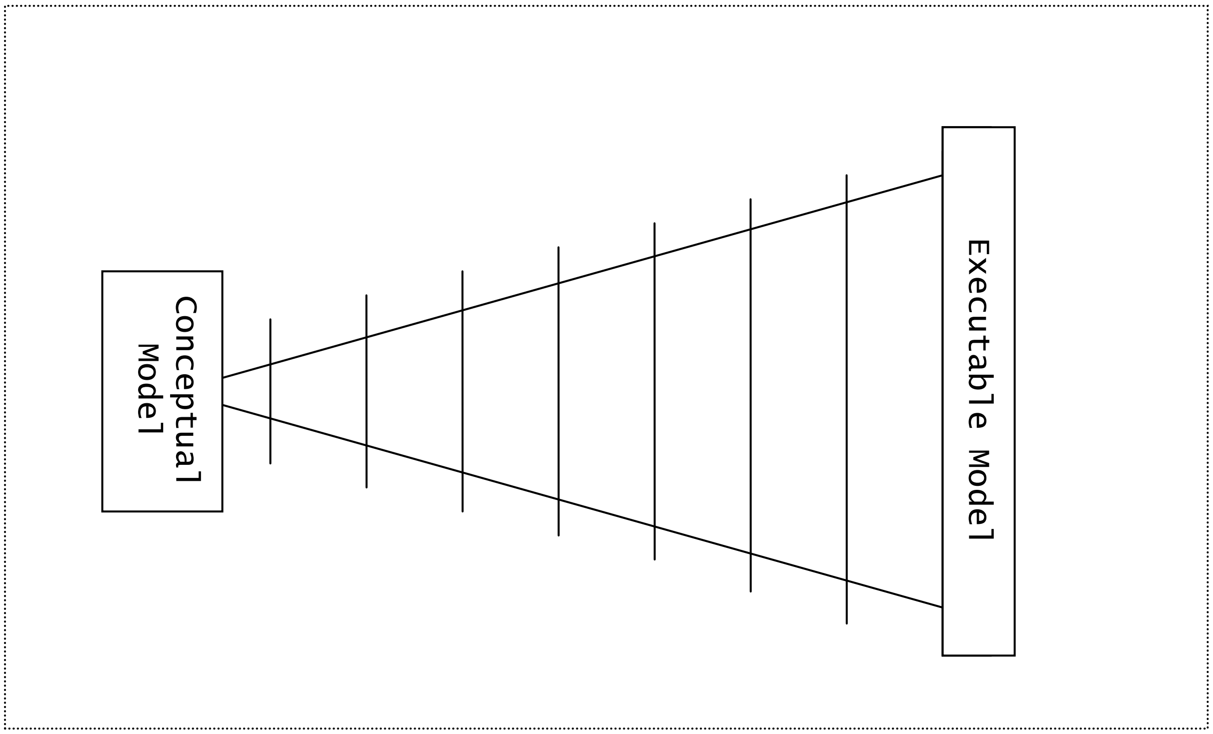
Of course, there are other ways to write essays. This outlining technique may not be helpful for all kinds of writing. When writing fiction, other ways of iteratively refining the structure of a piece of writing may be helpful. Major plot points can be outlined and ordered while characters can be listed and developed in parallel. The point is that having some process to follow is crucial to being able to solve large problems.

An object-oriented problem solving process

Now we are in a position to introduce our object oriented problem solving process. Using this process to solve a problem you end up with a solution to the problem expressed as a computer program. A computer program is a set of instructions that a computer can follow. By following the instructions and interacting with the world around it, a computer can mechanistically solve the problem.

The process makes use of a type of outline, just like the essay process. The outline for a computer program is bit different from the outline for an essay. To help emphasize this we call it a model rather than an outline. The first model that we produce we call a conceptual model, because it mirrors how we conceptualize the problem. As we refine the conceptual model it becomes closer and closer to what we call an executable model (since it is expressed in a precise enough way that it can be executed by a computer).

The following diagram is intended to show the progression from an abstract conceptual model to a concrete executable model. The model "fans out" to indicate that there is more detail in the executable model than there is in the conceptual model.



The Conceptual Model

What is a conceptual model? The conceptual model is a high-level outline of the main components of the solution to our problem. How do we arrive at this conceptual model? The conceptual model is a model of the problem domain. We therefore arrive at a conceptual model by constructing a model of the problem domain. This section explains what the problem domain is, and what sorts of things go into a model of the problem domain (i.e. the conceptual model).

The Problem Domain

When solving any sort of problem the first thing one should do is figure out the domain of the problem. The domain is the set of things which are relevant in some way to the problem.

Example 1

Suppose your problem is that you want to invite some friends for lunch. You need to figure out who to invite and what to serve them. What is the domain of this problem? The problem domain presumably includes the set of your friends, and their food likes/dislikes, their food allergies, what food you have in the house, and so on.

Things which are not relevant to solving the problem do not belong in the domain. For instance, things like the age of the roof of your house, or the type of roof shingles (cedar or asphalt), are not relevant and therefore do not belong in the problem domain.

Example 2

Consider the problem of modeling a human being for the purpose of simulating the effect of medications on them. It is probably relevant to this problem to include in the model the weight, age, height and gender of the human being, and so these characteristics should be included in the domain. The person's hair style, clothes and musical tastes are irrelevant (for the purpose indicated), and so don't need to be included in the domain.

Part of the art of being a good problem-solver is being able to pick out what the domain of a problem really is. In many so-called "textbook" examples the domain of a problem is given to you, or is very easy to pick out. In real-world examples, this is not always the case.

When faced with a novel problem to solve, think about what the domain should be. Do not worry about getting it exactly right the first time: problem solving is an iterative process. A process is iterative if it is repeated many times. In the problem solving context we continually refine our ideas and approaches as we learn more about a problem. In other words, since problem solving is a process of repeatedly refining a potential solution, it is not critical that our first attempt is 100% correct.[5]

It is interesting to note that this is also the way that the scientific method works: scientific knowledge progresses via a process of iterative refinement through experimentation (data gathering) and data analysis (examination of the experimental results). You can think of determining the domain of a problem as an experiment. You hypothesize what you believe is a good domain, and then test it (try to solve the original problem). If the domain is not good enough, the experimental results force you to revise your hypothesis.

The Conceptual Model: A Model of the Problem Domain

The conceptual model consists of objects which represent the items you've identified as being part of the problem domain, these objects' properties and capabilities, and their relationships with other objects in the domain. Here we briefly introduce these terms. Our understanding of them will deepen as we proceed through the book.

Objects

Each of the things identified as being part of the problem domain must somehow be represented in our computer program. Because we are learning object oriented programming, each element from the problem domain will be represented by an object. Exactly how to express this representation in our

[5] You can think of a problem solving process a little bit like golf, if you're familiar with the game. In golf you try to hit a golf ball from a starting position into a little hole. The hole is typically several hundred yards away from the starting position. The ideal is to hit a hole-in-one: to get the ball into the hole with one hit. This almost never happens, and is mostly pure luck when it does happen. The reality is to keep moving the ball closer to the hole with each hit. Not every hit moves the ball directly towards the hole; sometimes the ball goes from the starting position towards the hole in a bit of a zig-zag pattern. Problem solving in general is often like this. Rarely can you go directly from a problem to a solution in one step – it is as unlikely as a hole-in-one. More typically solving a problem is a multi-step process, which doesn't always move you directly toward the solution.

The process of getting the golf ball into the hole is an example is a program. If the ball is already in the hole, we're done. If not, hit the ball towards the hole, and repeat. An important question to ask it whether this program, when followed, will terminate (i.e. can we guarantee that the ball eventually gets into the hole?) Can you think of a condition which, if imposed, will guarantee termination?

executable models is something we will learn later on. For now, we need to remember that each element of the problem domain is represented by an object.

Properties and Capabilities

Each object in our computer program consists of properties and capabilities. Properties are things that objects have, whereas capabilities are things that objects can do.

A property of an object is a feature that describes the object. For example, your weight is one of your properties. A property value is the value of a property. For example, your weight property might have the value 150 lbs. Of course, property values can change. When you were born your weight might have been about 8 lbs.

The state of an object is the set of all its properties and their values at a given point in time. The set of properties of an object is fixed, but as we have just seen, its property values can change. Thus, the state of an object changes whenever the value of one of its properties changes.

Example 3

Suppose a room has as a property the color of its walls. If on Tuesday the room has green walls which are on Wednesday painted yellow, then the state of the object on Tuesday is {wall color = green} and on Thursday it is {wall color = yellow}.

Example 4

Suppose a taxi cab has among of its properties the driver who drives the cab, and the people/customers who ride in the cab. At the beginning of the day, Moe the cab driver gets into his cab ready to drive people around town. At this point the state of the taxi cab is {driver = Moe, customers = none}. On the corner of 5th and Main Street, he spots his friend Binky hailing for a cab. After Moe picks up Binky the state of the taxi cab is {driver = Moe, customers = Binky}. Now suppose that after Moe drops Binky off, he picks up Fred and Barney who are going to the park for the afternoon. At this point the state of the taxi cab will be {driver = Moe, customers = Fred, Barney}.

Capabilities are the actions that an object is capable of performing. One way to discover the capabilities of an object is to think of the capabilities that it can perform for other objects when asked. We call capabilities which can be performed for other objects services.

Example 5

Consider a restaurant and the people who work or patronize it. In this example a server takes a patron's order, communicates it to the chef (who prepares the requested meal), and brings the completed meal to the patron, in exchange for payment. These are capabilities of a server – services that they provide to other objects in their domain. Notice that the chef is an object which the patron does not interact with directly. The server and chef form a subsystem; the patron need not be aware of *this internal structure. The server functions as a façade*[6] *for the subsystem, delegating the task of actually preparing the food to the chef.*

Example 6

Consider the capabilities of a warehousing operation (not the physical building). We may want to include such capabilities as adding a new item to warehouse's inventory, removing an item from the warehouse's inventory, and reporting how many of a particular items there are in the warehouse.

When identifying the capabilities for an object, we are looking for actions that an object can perform. At times, the capabilities will represent interactions between two objects. Other times, the action will be internal to the object.

Relationships

Each object in our domain can be related to other objects in a variety of ways. The essential reason for these relationships is to allow objects to communicate with one another: to let one object invoke the capabilities of another object. It is important to identify these relationships when building the conceptual model. Until we know more about the different sorts of relationships that can be established, we will identify a relationship (of an unspecified type) whenever two objects need to communicate with each other.

Unified Modeling Language (UML)

We express our conceptual model using a standard notation called the Unified Modeling Language, most commonly referred to as the UML. UML is a very large and rich language, and in this book we will introduce a very small part of it, just what we need.

The UML which we will use is expressed diagrammatically. We will have more to say about UML as we go along. We will learn how elements of the UML can be translated into a high-level computer programming language. There are many tools available which can translate from UML to programming language, and some which can translate in the reverse direction too. We will use one such tool, called Green (green.sourceforge.net).

[6] "Façade" is an architectural term. It refers to the front of a building, what people see when they look at a building from the street. Movie sets used in old Westerns often consisted of a main street with boardwalks and only façades: there was no actual building behind the façade. We can also say of people that they put up a façade if they are not showing their true behavior or personality. In software terms a façade is an object which serves as the point of contact for a subsystem (it is the only object from the subsystem visible to objects outside the subsystem; it is also sometimes called an interface object). The façade object receives messages from objects outside the subsystem and forwards them to some appropriate object within the subsystem to actually respond to the message. This forwarding of a message is called delegation.

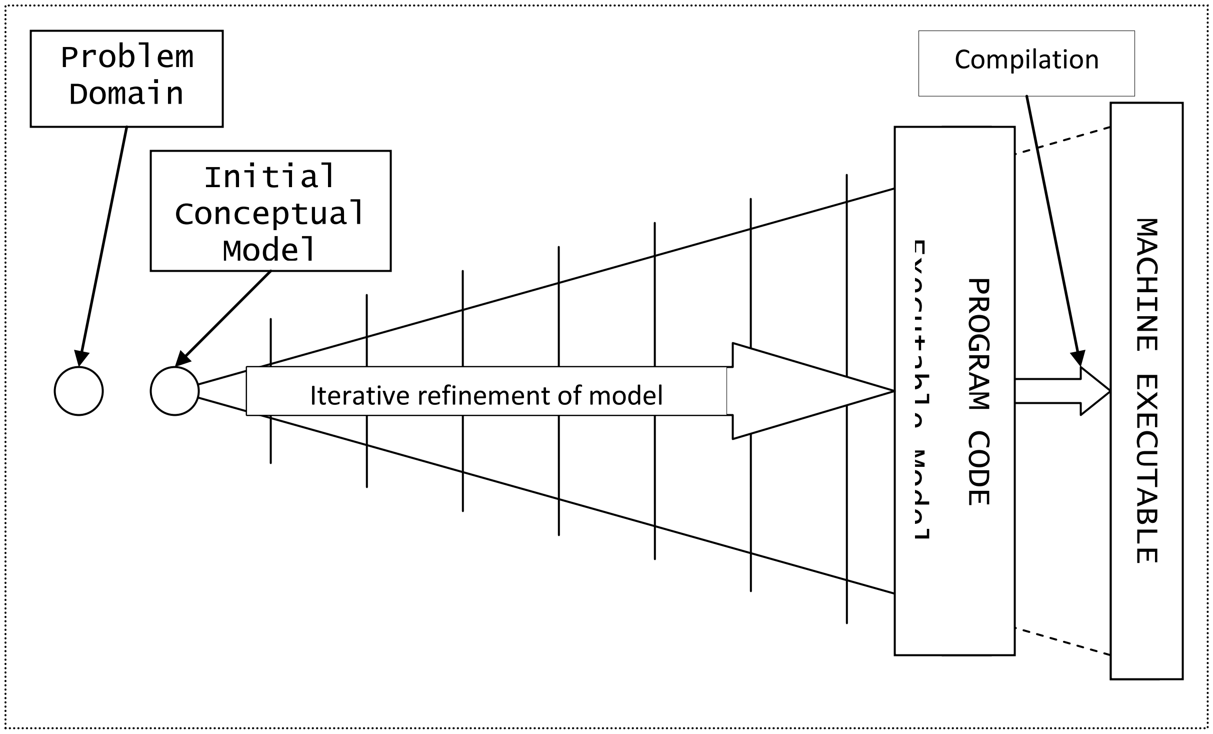
The executable model

Our initial conceptual model is a first approximation to a solution to a problem. This model, expressed using UML, is not an executable computer program. The initial conceptual model must be refined to the point where a computer can directly execute it.

We will take an executable model to be a model expressed in a high-level computer language. The computer programs that we show in this book are written in a high-level programming language called Java. There are many other programming languages, and there is nothing particularly special about Java. We had to make a choice of programming language to express our computer programs in, and Java is the language we settled on. The basic principles you learn in this book transfer to other programming languages.

While we think of programs written in high-level programming languages as executable, technically they are not. It turns out that programs expressed in high-level languages must be expressed in a low-level machine language before being executable directly by a computer. Luckily there are computer programs which can translate high-level computer programs into machine language computer programs. A program which performs this kind of translation is called a compiler. This demonstrates another significant result in computer science, namely the equivalence of programs and data (the inputs to programs). In the case of a compiler, its data (the input to the compiler) is itself a computer program. In fact, the output of the compiler is another computer program! Alan Turing noted that it is possible to write what is called a Universal Turing Machine. A Turing Machine (TM) is a very simple computer. By the Church-Turing thesis any program that can be expressed in any programming language can be carried out by a Turing Machine. Despite its simplicity, it is possible to write a TM which can take as its input another TM, and the input to that second TM, and have it simulate that TM on its input. Such a TM is called a Universal TM.

With this in mind, our picture of our problem solving process can be refined to become:



At this point it might be helpful to get a better sense of what we are aiming for when writing an object oriented program. The basic building block of an object oriented program is the object. A running object oriented program is a collection of objects. But it is not only a collection of objects. It is a system of interacting objects. Objects have capabilities which other objects in the system can invoke: in this way they interact with each other.

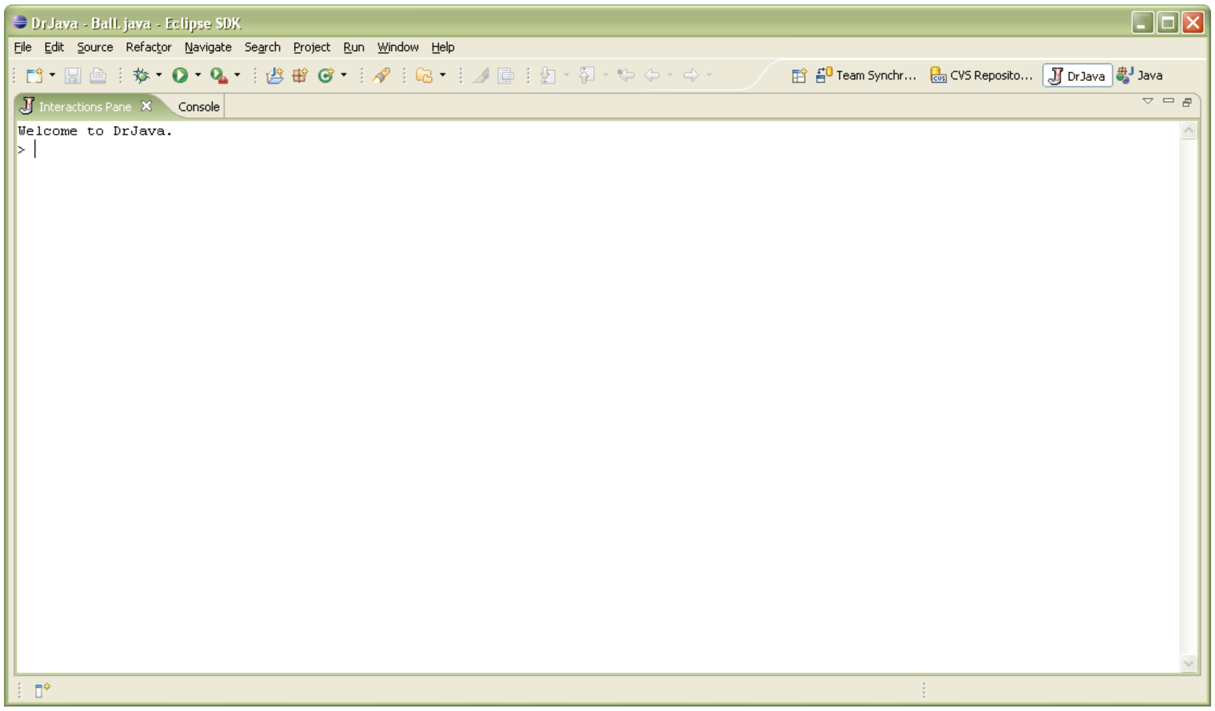
The history of programming language design can be thought of as striving to shorten the distance between our initial conceptual model and the executable model, the program code which can be mechanistically translated into the machine language. Another way of thinking of this is that as programming languages come to support better abstractions, and compilers are able to perform better translations, the executable model moves further from the machine executable form of the program.

Creating and Interacting with Objects

Before we go too much further considering objects only in theory, it is time we showed an example of an object. We recommend that you use this text together with a good development tool. While there are many out there, we use a tool called Eclipse (www.eclipse.org). Eclipse is an integrated development environment (IDE), meaning that it brings together (integrates) many useful tools and features for writing software into one application. We also use two plug-ins to Eclipse, plug-ins which enhance its functionality in pedagogically useful ways. These plug-ins are Green (green.sourceforge.net) and DrJava (drjava.sourceforge.net). If you download DrJava yourself, be sure to download the Eclipse plug-in, and not the free-standing tool. Eclipse and these plug-ins are freely available.

The DrJava plug-in to Eclipse provides what it calls an interactions pane. In the interactions pane you can type bits and pieces of Java programs, and have them executed directly, without having to write a complete Java program. We will take advantage of this to demonstrate how objects are created and how to invoke objects' capabilities.

The diagram below shows what the interactions pane looks like initially:


DrJava - Ball.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Team Synchr... CVS Reposito... DrJava Java
Interactions Pane
Console

```
Welcome to DrJava.
> 
```

The interactions pane displays a welcome message ("Welcome to DrJava") and a prompt ("> "). Anything you type at the prompt will be taken as Java, and performed right away. In the following two subsections we show how to create some objects and invoke capabilities of one of the objects.

Creating an object

So that we are not doing things in complete isolation, let us set up a scenario for what we are about to do. Imagine that we're studying the behavior of ant colonies. The problem domain consists of the soil (what the colony is physically constructed from) and the ants. As such we must model the colony and the ants in it. For the purposes of showing how to create objects and invoke their capabilities, we have already written the required computer program. The soil is part of a terrarium object. The ant is its own object.

To create a terrarium, we write the following expression in the interactions pane:

new chapter1.Terrarium()

An expression is something which can be evaluated to produce a value. For example, the mathematical expression 3+4, when evaluated, produces the value 7. The linguistic expression the capital of the United States, when evaluated, produces the value Washington D.C.

The value of the Java expression **new chapter1.Terrarium()** is an object reference. Let's explain this. When this expression is evaluated, a new Terrarium object is created. This object is stored somewhere in the computer's memory. The object occupies some amount of memory (determined by the objects properties), and its representation will be stored in a contiguous block of memory of just the right size. The address of the first byte of this block of memory is returned as the value of the **new**

chapter1.Terrarium() expression; this value is called an object reference. Sometimes it is called just a reference, or even a pointer.

Let us look at the parts of this expression. From our perspective it has three distinct parts:

new

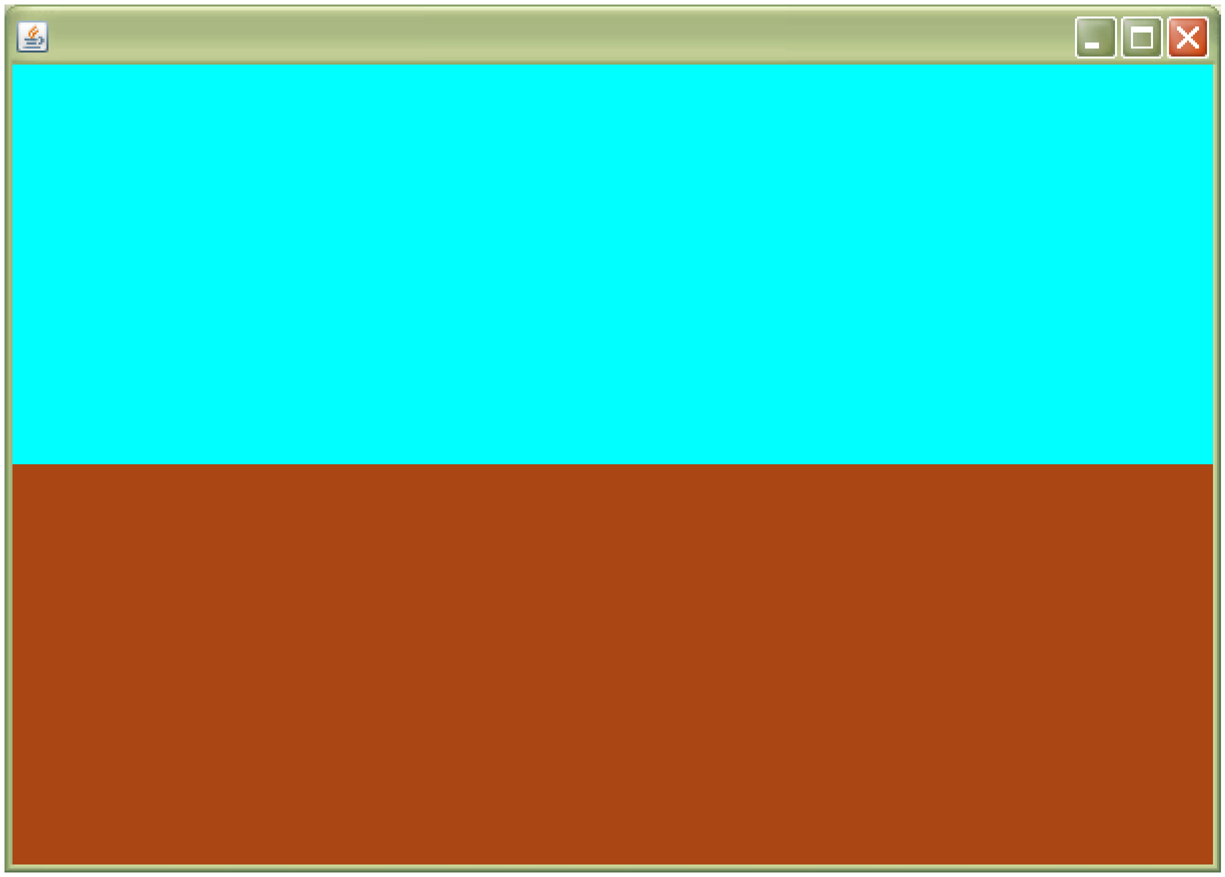
chapter1.Terrarium

()

Although we will return later on to explain these pieces in more detail, a brief explanation is appropriate now. The first part, **new**, says that we want to create a new object. The second part, **chapter1.Terrarium**, says which kind of object we want to create. Finally, **()** indicates that we want a default (sometimes called "plain vanilla") object created.

At this point in our exploration of the material it is not important to understand exactly why new objects are created in this way. The point is simply to demonstrate the creation of an object. The details of what this means will be left until the next chapter.

When a new Terrarium object is created, a window opens on the screen:



This window is created as a result of the creating the Terrarium object – this is part of its pre-programmed behavior. The DrJava interactions pane displays some information, which we will ignore for the time being.

Interacting with an object (i.e. using capabilities)

Next, we will create an Ant object and place the Ant in the Terrarium. To create an Ant object we write:

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

However, this does not appear to do much: nothing changes in the Terrarium window. The DrJava interactions pane does display some new information, and in fact an Ant object was created. However, there is no connection between this Ant object the Terrarium object. In our goofy example, the ant needs to be placed into the terrarium. Once in the terrarium, it will appear on-screen.

To add the Ant object to the Terrarium object we type:

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

This is a lot of typing! What does it do? We know that

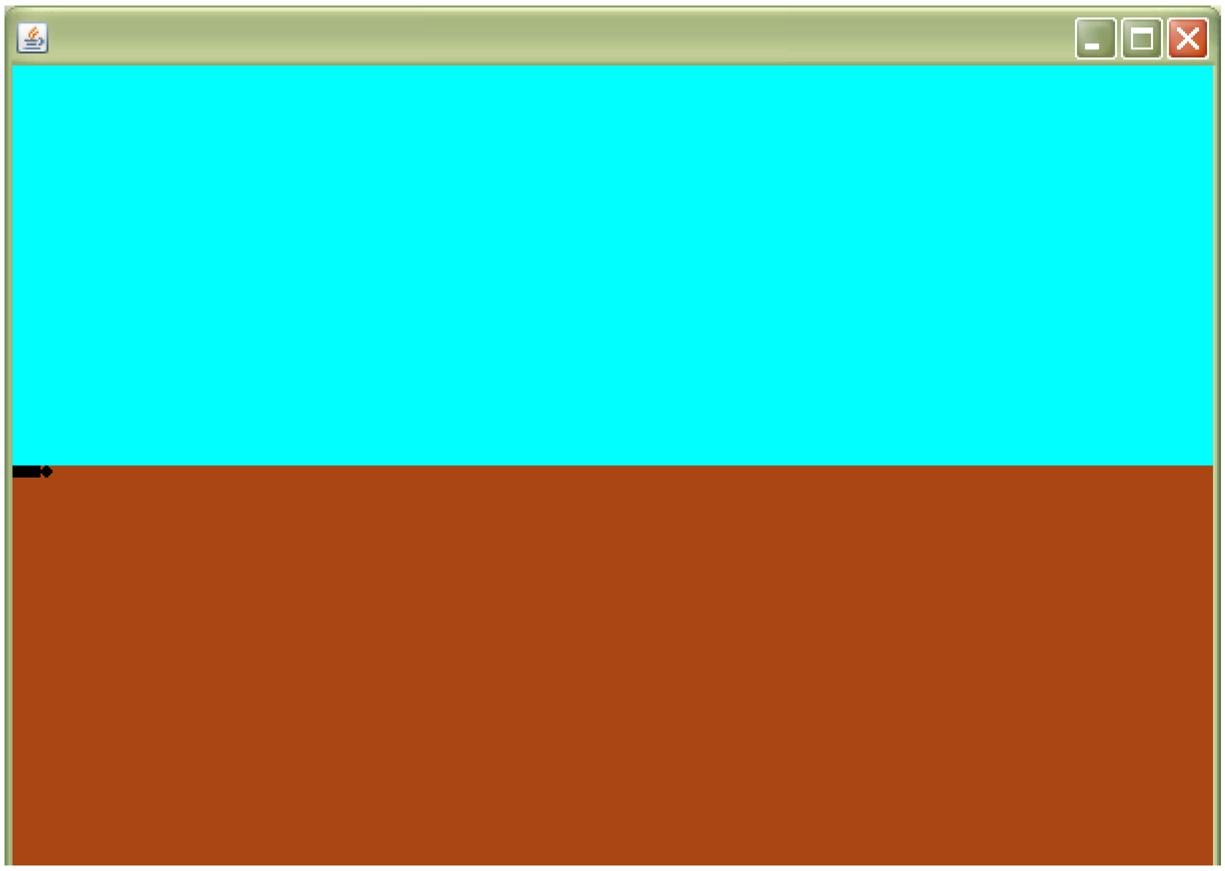
```
new chapter1.Terrarium()
```

creates a new Terrarium object. Similarly,

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

creates a new Ant object. Remember that **new** says we want to create a new object, **chapter1.Ant** says what kind of object we want to create, and **()** says we want to create a default (plain vanilla) object. The **.add(…)** part of the expression is something new. It says to use one of the capabilities of the terrarium. We say that add is a method which we are invoking on the Terrarium object, and we are sending along the new Ant object as an argument in the method call. Notice that **new chapter1.Ant()** appears inside a set of parentheses, which are part of the method call.

On screen we see something like this:



The Ant object is represented by the skinny black image between the blue area (the sky) and the brown area (the soil).

This Ant just sits there. We can get it moving by calling its start method, a method which takes no arguments (so the parentheses that are part of the method call to start are empty:

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

Now a Terrarium window opens with an Ant that moves – we have invoked a capability of the Ant object!

Summary

We started this chapter discussing some basic problem solving strategies that are useful when trying to solve many real-world problems. We have looked at how to model the domain of a particular problem and that important decisions must be made about what information is eventually included in our model of the domain. In computer science, we build executable models to solve our problems, and in this book, we are focusing on object oriented models, which consist of a set of objects that interact with one another to solve the problem. Objects in our model will be identified from our problem description. After we have identified the objects that are important for solving our problem, we define each object in terms of its properties (features) and capabilities (actions).