Imagine that you have just returned home from some fantastic and strenuous journey. You unlock the door to your house with one of the many keys on your key ring, walk inside and grab a glass from the cupboard. You place your keys back in your pocket and then walk into the kitchen and turn the light on. You take the glass and walk over to the fridge, placing your cup under the ice dispenser, filling the glass with ice. You then press the button for the water dispenser, filling the remaining space in the glass with water. Finally, you walk with your glass of ice water into the living room, sit down on your sofa, kick your feet up on the coffee table, and then relax and enjoy a nice drink.

To perform this simple task, you have had to interact with a number of different objects, including your pants pocket, the keyring containing the key for your front door, the doorknob, the kitchen light switch, the cupboard's handle, and the buttons on your fridge. There isn't anything remarkable about your interactions with these objects, in fact, the whole sequence sounds like something a trained monkey could do.

However, if you look just below the surface, you'll see that each of these objects has been carefully designed to make interacting with it as simple as possible. You do not need to think about the tumblers of a lock to successfully make use of a key, nor do you need to think about the hinges of a cupboard to open it. The same goes for the electrical wiring and water lines for your house. Since you don't even need to think about how these primitive systems work, you certainly don't need to contemplate the inner workings of a mechanical ice dispenser to be able to push a button and watch ice pop out.

Just as mechanical design can be used to simplify interactions with physical objects, software can be designed in the same manner. Programming languages that provide the building blocks that carry this analogy into the digital sphere are known as object-oriented languages. Among object-oriented languages, Ruby stands out as being particularly rich. We can catch a glimpse of why this is the case by exploring a simple example.

If we adapt our previous thought experiment, we can imagine a home that rather uses a digital keypad instead of a key for unlocking its doors. In such a system, you would key in password that would be verified by a program before the door could be unlocked via an electrical signal. While I've simplified greatly, the program for such a system might look something like what you see below if it were written in Ruby.

```ruby
####################################
# Combination-lock Implementation #
####################################

combo_lock = Object.new

def combo_lock.set_password(new_password)
  @stored_password = new_password unless locked?
end

def combo_lock.lock
  @locked = true if @stored_password
end

def combo_lock.unlock(entered_password)
  @locked = false if entered_password == @stored_password
```

```
end

def combo_lock.locked?
  @locked
end

###############################
# Combination-lock Interface #
###############################

combo_lock.set_password("1337")
combo_lock.lock

p combo_lock.locked? #=> true

combo_lock.unlock("1336")

p combo_lock.locked? #=> true

combo_lock.unlock("1337")

p combo_lock.locked? #=> false
```

In this example, we've used a freshly created Ruby object as a building block and imbued it with the behaviors of a combination lock by defining certain methods on it. Each method performs a procedure, possibly doing some manipulation of the object's internal data, such as its locked status or stored password. However, looking at the implementation is a bit like looking at the tumblers of a mechanical lock: it tells you a lot about how it works, but isn't particularly good at illustrating how the object ought to be used. Conversely, if we look at the code that interfaces with the combo_lock object, we are able to see a whole lot about how it's meant to be used without catching even a glimpse of how it is implemented.

While we're only scratching the surface with this trivial example, the benefits of being able to separate the concept of an interface from its implementation are vast. For example, we could swap out our implementation that stored passwords in plain text with one that uses cryptographic hashes without ever having to change the interface code, as illustrated in the example below.

```
############################
# Crypto-combination lock #
############################

require "digest/sha1"

combo_lock = Object.new

def combo_lock.set_password(new_password)
```

```ruby
  hashed_password  = Digest::SHA1.hexdigest(new_password)
  @stored_password =  hashed_password unless locked?
end

def combo_lock.lock
  @locked = true if @stored_password
end

def combo_lock.unlock(entered_password)
  hashed_password = Digest::SHA1.hexdigest(entered_password)
  @locked         = false if hashed_password == @stored_password
end

def combo_lock.locked?
  @locked
end
```

While building a single logical machine is easy, the interesting work is in composing behaviors of many objects in a system to model complex scenarios. Because the examples we've looked at so far use only the most primitive tools Ruby has to offer, they may seem quite boring. However, as we explore more elaborate problems, we'll see Ruby's object system get a lot more interesting.

Ruby is intensely human-centric language that is designed to allow us to model our software in ways that closely mirror how we model things in the physical world. This book challenges you to gain an intuitive understanding of what that means by exploring its consequences in realistic scenarios.