# CLEAN RUBY

by Jim Gay

# Introduction

Our simple ideas are often far more difficult to implement than we first expect. Over time we develop new ideas, change our understanding, discover unexpected consequences, and more. We grow programs with new features in a way that makes us curse our "monolith." Often we are the victims of our own devices.

It's easy to blame the bigness of our application and formulate plans to make things better. "If I only had time to refactor this," you might be thinking as you browse your massively large classes. But how do we get there? How is it that we start with something simple and the best of intentions, but we end up with code bloated with unnecessary indirection and complicated responsibilities?

Despite studying and following common design patterns in your program you continue to lose the ability to maintain a clear mental model of your growing codebase. Jumping in at any point in the code requires digging around in your classes before you're reminded of exactly how every piece fits together. Does this sound familiar?

You can change this. You can make your program more intentional and your intentions more obvious. *Clean Ruby* is an introduction to a way of thinking about object orientation that's a bit unlike your current understanding. This book focuses on teaching you ways to remove distractions and unnecessary hierarchy from your business logic.

I have four goals with this book; that you will:

1. Learn to understand the difference between your program's *form* and *function*;
2. Describe your application's *use cases* in code;

3. Write code that feels familiar, even to new developers; and
4. Dramatically simplify and speed up your automated tests.

*Clean Ruby* comes from my own experience in working with teams to simplify our code and apply a new way of thinking about object-oriented programs with DCI (Data, Context and Interaction). DCI is the brainchild of Trygve Reenskaug[1] and while you may or may not know the Reenskaug name, you know and likely practice an approach to object-oriented programming that he created: MVC.

The MVC paradigm (Model, View, Controller) is a way to ensure that programmers, like you and me, write code that supports the end user's actual understanding of a program and its functions. This approach became popular because it allows us to very clearly separate concerns and encapsulate required logic into individual objects, and distinctly map the ways of the world in our programs.

What you, I, and many others—including Reenskaug—have found is that we still tend to write code that's not quite as revealing about its purpose as it should be. After spending years considering problems that we find in our object-oriented code, Reenskaug has formulated an approach that puts the business use case in the driver's seat of your programs. DCI is a complement to your MVC structure and is an approach that isn't just an example of writing tests in a BDD (Behavior Driven Development[2]) style; it's all about writing **code** that describes the behavior.

James "Cope" Coplien[3] has been a major partner in developing DCI along with Reenskaug, and pushing the ideas into the OOP community. Cope is a leader in Agile and Lean software development communities. Among many things, Cope is the co-author of *Organizational Patterns of Agile Software Development,* and he and Gertrud Bjørnvig[4] lay out a clear picture of DCI in *Lean Architecture*[5], an excellent book on the hows and whys of the thinking that begat DCI.

Many programmers begin serious programming with object orientation. There are now so many "best practices" and patterns for OOP that it's hard to avoid writing a

---

[1] http://heim.ifi.uio.no/~trygver/

[2] [2] http://en.wikipedia.org/wiki/Behavior_Driven_Development

[3] http://en.wikipedia.org/wiki/Jim_Coplien

[4] http://scrumfoundation.com/about/gertrud-bjornvig

[5] http://leanarchitecture.com

program without them. Many of us still don't spend time considering the purpose of OO and how it can be used to benefit our applications, so it's well worth our time to consider object modeling perspectives from early languages like Simula.

## What This Is

Clean Ruby is a guide for understanding and applying thinking that will allow you to build systems. It's focused on discussing ideas to communicate the semantics of working software.

There's plenty of thinking and understanding that needs to occur before you have any chance of writing a coherent, maintainable program. *Clean Ruby* first will lay a foundation exploring why we approach programming from an object-oriented perspective and what we can learn from user interface design.

User interface design can teach us a lot about programming. When we create applications, we interact with the files, classes, method names, and more in the same way we interact with a graphical user interface. In this book we'll explore aspects of interface design that can help us have a better intuition about our application architecture.

## What This Is Not

To get the most out of *Clean Ruby* you should have experience writing Ruby. This isn't a tutorial that will get you up to speed on the language. This isn't a tutorial on how to use any of the popular frameworks like Ruby on Rails either. So be prepared.

# Lost In Translation

## *Modeling a Business*

"We only need a simple registration form," your client says. You pause, knowing well enough that use of the words "only" and "simple" implies that they'd like your work to be fast and cheap. It also means they likely haven't thought things through.

It's moments like this that often lead us to either love or hate our jobs.

Mental models are an important feature both in object-oriented programming and the ideas in this book. While the phrase *mental model* seems easy enough to grasp by itself as a representation of something in our minds, it's helpful to consider deeper aspects of it.

In *Mental Models*[6], one of the first books to introduce the idea, a few of its main points are discussed by Donald Norman (Gentner 7-14). Norman points out that through observation of test subjects he discovered that mental models are:

- incomplete
- difficult to mentally "run"
- unstable
- unscientific
- ungenerous (favoring manual labor over cognitive load)

---

[6] Gentner, D., and Albert L. Stevens eds. Mental Models, Lawrence Erlbaum Associates, Inc., 1983. Print.

These characteristics of our mental models show us that it's no surprise that business owners (or any person in charge of determining a business process) will often approach developers with an incomplete idea. A "simple registration form" may represent an understanding of a process but that understanding may not be complete. It may reflect someone's incorrect assumptions about both the needs of the software and the needs of the business process itself.

> "Do you want immediate membership or will we need to confirm, for example, by clicking on a link in an email generated after the visitor submits the form?"
>
> *"Why would we do that?"*
>
> "One reason would be to prevent accounts from being registered and used to immediately to add spam messages to your system."
>
> *"Oh. I hadn't thought of that. Do we need that?"*

Our mental models of relevant business processes can be (and often are) incomplete. There may be reasons why one aspect or another goes unnoticed until later in the development of your software, but that's not always because someone is inexperienced or incompetent.

Norman points out that if we represent any system as *t*, then an end user's mental model of the system is *M(t)*. That is, it's a function of the mental model; there's a separation between the actual system and the understanding of it. Worse yet, we can formulate our own understanding of the end user's mental model as *C(M(t))*. We need to cut through layers of indirection in an attempt to understand an understanding of a system.

## Our First Bugs

One of the greatest challenges in writing software is communication. What does each variable, method, and class do? Even in answering that question, how can we describe how it all fits together to form the whole?

We can learn from research like Norman's that mental models are both difficult to communicate and often misunderstood. This is where our first bugs enter the program. These bugs aren't of the kind that come from typos or `nil` return values leading to unexpected results. Our first bugs often come from our expected results being incorrect or our mental models being incomplete.

Over time, these incorrect expectations and incomplete mental models lead us to anticipate something: *change*. Change is why so many have found value in the Agile and Lean approaches to software development. Change seems to come more often than anything else in software development, and to deliver working software the creators of the Agile Manifesto[7] specifically highlight "responding to change" as more valuable than following a plan. Responding to change is more valuable than following a plan because we base our plans on mental models of a system and if our mental models are incomplete, then so will be our plans.

Going forward with this knowledge about the limitations of our mental models we can confidently say: we will be wrong. Our first attempts at understanding will be wrong and we'll go back to the business owner for more details, more discussion and more thinking. But this response is a good thing; it means we take the time to get it right. Unfortunately, more often than not, these "final" discussions aren't, and we eventually become frustrated or disappointed when it ends up that "final" isn't really final.

To avoid frustration and to prevent these bugs from complicating our code, making it even harder to change, we can separate our business processes from our domain models. We should prepare our application's architecture to respond well to the changes in our processes and the changes in our understanding of the models we're building. We should keep in mind some words of wisdom from Reenskaug and Coplien:

> A key, longstanding hallmark of a good program is that it separates what is stable from what changes in the interest of good maintenance.[8]

Change is expected, so plan for it; build for it.

---

[7] http://agilemanifesto.org

[8] http://www.artima.com/articles/dci_vision.html

# Object Orientation

## Where We Are

Every Ruby programmer knows about object orientation. In Ruby, almost everything is an object that can be given properties and actions. The ruby-lang.org website highlights this feature in particular under "Seeing Everything as an Object"[9] teaching newcomers that even numbers are treated as objects.

Object-oriented Programming (OOP) has been a dominant style of programming in recent decades. Although this style is pervasive, the definition of "object-oriented" seems to have a number of interpretations. Smalltalk gave us a definition of objects:

> An Object is an entity that has identity and that encapsulates state and behavior.[10]

Alan Kay, who coined the term "object-oriented programming" later said:

---

[9] http://www.ruby-lang.org/en/about/

[10] http://heim.ifi.uio.no/~trygver/2011/DCI-Glossary.pdf

> I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages[.]
>
> OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.[11]

The SOLID[12] principles give us a guide for building Object-oriented (OO) systems in a way that helps us balance control of responsibilities and flexibility. In particular, as Jim Coplien has pointed out to me in discussions about object orientation, the Liskov Substitution Principle (LSP) "is a quite broadly accepted definition in the OO world."

LSP is a principle that, when followed, ensures that an operation expecting an instance of one class should accept a substitute instance of one of it's subclasses and continue to function without the need to change the program. The "Treaty of Orlando"[13] solidified requirements of OO systems that they implement sharing mechanisms for object behavior via either delegation or inheritance.

It would be helpful to have a simple definition which combines these ideas and to which we can anchor our discussion. Let's pick one.

> Object-oriented programs execute algorithms by sending messages among objects, where the objects are an encapsulation of data and functions that perform operations.

Perhaps you might choose to word this differently, but it's of little consequence. We're saying that we represent things in a program with objects and those objects know how to perform actions. And, these objects talk to each other by sending messages.

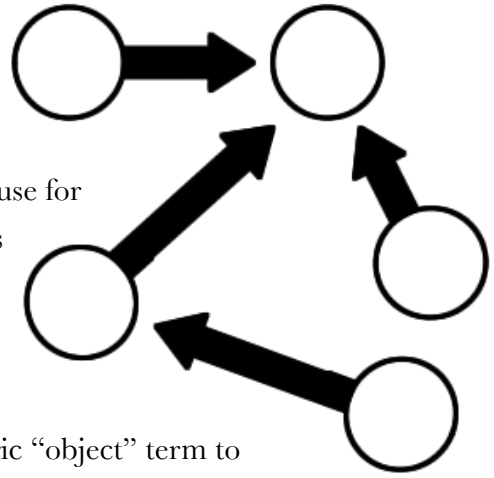
---

[11] http://www.purl.org/stefan_ram/pub/doc_kay_oop_en

[12] http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

[13] [Stein et al., 1989] Stein, L. A., H. Lieberman, and D. Ungar. "A Shared View of Sharing: The Treaty of Orlando." Object-Oriented Concepts, Databases and Applications. Ed. W. Kim and F. H. Lochovsky. Reading (MA), USA: ACM Press/Addison-Wesley, 1989. 31–48.

We can extrapolate this simple idea into a graphic view of the objects of our system communicating by sending messages: objects (circles) sending messages to each other (arrows between them).



## What are objects?

The objects that we use may be called "account," "article," "user," or any other word we use for these things. This is important because using words like this **reveals** to us the **intentions** of what these objects represent. We tend to focus on nouns with OO design.

In our world of programming we use the generic "object" term to refer to so many things, but when we get into the details of our applications we have objects with more specific and meaningful names. Most often our program's objects are given certain behavior as well: behavior that often reflects the typical functions of real world objects. We model what we know so we can better reason about how it functions and better communicate with each other through our software.

In a typical Ruby application we initialize objects through a relevant class.

```
class Account
   def initialize(cents)
      @cents = cents
   end
end
account = Account.new(500000)
=> #<Account:0x101a776f8 @cents=500000>
```

Here, we've set up an `Account` class to store an amount of cents when creating the new object. This is just basic Ruby code and it allows us to initialize an object that represents an account containing an amount of money.

## Doing more

Along come requirements from the business that says the account holder needs to add money to an account and we dutifully add that ability to the class:

```
class Account
  def add_money(cents)
    @cents += cents
  end
end
```

Now, of course, with objects that manage money you're likely to need to transfer from one to the other and the business owners are quick to remind you of that need. This class is simple enough, so we can just add a method to do that:

```
class Account
  def transfer_to(account, amount)
    @cents -= amount
    account.add_money(amount)
  end
end
```

When we have two `Account` objects, one of which is sending a message to the other, our mental model is simple to diagram:


In the above diagram we see one account in green sending a message to another account in orange. The arrow in the middle represents the message being sent.

We just added the `transfer_to` method to the `Account` class and we are able to move money between accounts. But there's more to what we've done than just that: we've added behavior.

This is where we begin to complicate our code, but it isn't obvious. You're probably wondering how in the world I could take a two method class and tell you that this is the start of something undesirable. This is exactly what we need when designing programs

around the way the real world works, isn't it? It *feels* fine, allowing us to initialize objects that encapsulate data and functions, which is, after all, how we defined object orientation.

## Restricted OO

Here's the distinction: Our `Account` is no longer just a representation of something that contains money. It now has the innate ability to add and transfer money as well.

At the outset this may seem completely normal, perfectly OK, and just plain trivial. After all, our definition of OO says that our objects encapsulate data and functions; this is exactly what we've created; this is OO.

We've gone from a class that represents a thing, to a class that represents a thing as well as defining its actions. Notice here that I referred to the code example being about the "class" and not the "object." In traditional OO Ruby programs, we initialize objects through classes and as our applications grow, we add any necessary behavior for those objects to its class. We've just implemented this feature and have complected the two requirements of the system with what we can call **Restricted OO**.

We'll better define Restricted OO in a minute, but first we need to agree that we've combined behavior and state with our `Account` class, which still fits our working definition. We built a class to represent objects that store information (how much money) as well as perform actions (transfer money). We've built our class with responsibilities in mind.

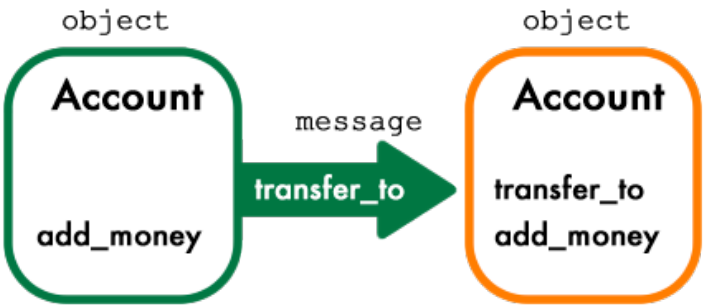
Why call this *Restricted* OO? Well, let's step back and first just call this OO, since that's what most people would call it now. With OO, the perspective the programmer takes is from within the object itself.

This traditional approach allows you to see the state of the object as defined by its class, the public methods that may be called on the objects (or the messages that may be sent to it), the private and protected methods that will be called to implement some aspect of the called public methods, and the attributes that will be changed by these methods.

The problem with a perspective like this is that we need to understand so much more about the class of an object while knowing so little about the actual execution of the program. With this paradigm, we know at some point these methods may be called and the object's state may change, but there's nothing deterministic about this. We're still left guessing about the program's actual behavior and are often forced to go spend time reading documentation files, wikis, or user acceptance test scripts that pull us further away

from where we want to be: in the program code. Our understanding of each object carries with it all the baggage of the entire set of potentially unrelated behaviors regardless of the necessity of the features they provide. Going back to our example, this means that our source and destination accounts can transfer money to another even though we only intend for one to do so.

By making a diagram of these objects and what actions they can perform, their roles are less clear than what our mental model of OOP tends to be:

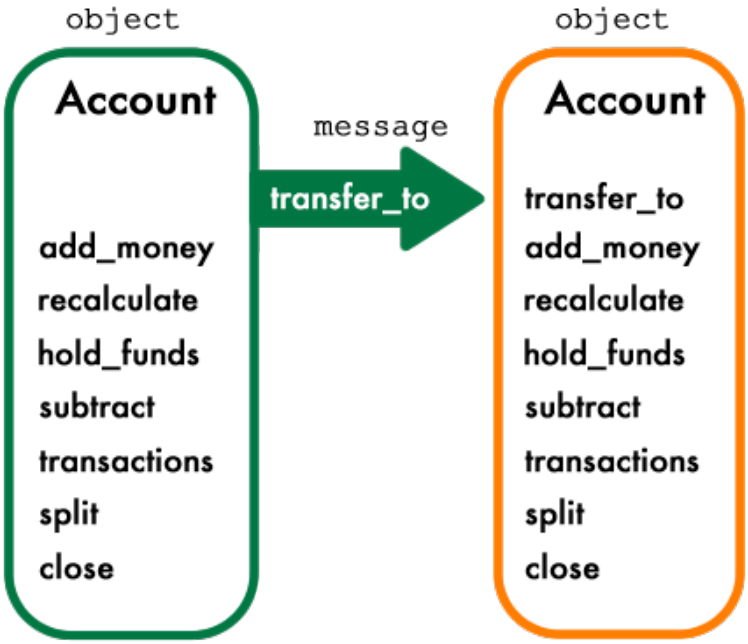



In this diagram we see the two objects and their available functions.

With few responsibilities in our classes this perspective appears to be reasonable. If we increase the number of methods per object (which is extremely likely as our program grows) and if we multiply the number of objects in this interaction then our simple diagram of interconnected objects becomes much more difficult to decipher.

Let's think about a visual example of this `Account` class as it grows. Currently we have `add_money` and `transfer_to` methods. As an example, we'll add more methods that you might find on a class like this. By merely adding a few more methods, our

graphical depiction has grown significantly:




While visual references are often great tools, they can exaggerate problems very easily, so let's shift to some running code examples for a more concrete approach that you'll experience with your own projects. Here's an example of running code in IRB:

```
> account = Account.new(100)
 => #<Account:0x0000010085ac90 @cents=100>
```

We've initialized an account with a `@cents` value. Now that we have our account, let's see what we can do:

```
> account.public_methods(false)
 => [:add_money, :transfer_to]
```

Ruby allows us to pass `false` into the `public_methods` call to return only methods that are defined on the objects immediate ancestor and not any inherited methods. This makes understanding our object much easier. Were we to print all public methods (omitting the flag we set here) we'd see a large list of methods that are mostly unimportant to our current needs.

Unfortunately and as we've seen in our diagram, as we add methods to our class, our list of options for every object from that class grows.

```
> account.public_methods(false)
 =>
[:add_money, :transfer_to, :recalculate, :hold_funds, :subtract,
:transactions, :split, :close]
```

If we only care about transfers, these extra features are noise, serving only to increase the cognitive effort require to understand our code. It's more mental strain to understand this account object. Although this list of methods is small, I'm willing to bet that you've had a class that didn't stop at seven methods.

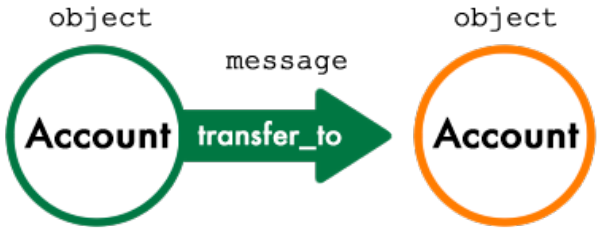
We can call this "Restricted OO" because our goal of readable code is supported only by the restrictions that we place on our architecture. Because our perspective is from within an individual class, we restrict our architecture to that view having no broad concrete understanding of how and when our domain models interact. With this, and to quote Coplien, "a programmer can understand the program behavior only in general terms."[14]

We can only understand the code in general terms because there is no clear indication of what objects are present and of what methods they require at any given point. This is because our individual objects grow with our requirements. As a result, we are forced to understand increasingly more about our system in its entirety to make sense of specific scenarios.

How can we fix this? Can we drop the baggage of the class and move back to the perspective that we really prefer to have?

## Full OO

If we were to be fully aware of the behavior of our objects during the execution of the program (that is at runtime, not at compile time as defined in our classes), we could see the world our program creates and see the world the way we think of OO.




[14] https://groups.google.com/forum/#!msg/object-composition/umY_w1rXBEw/hyAF-jPgFn4J

With Full OO we observe our objects from a perspective of being among them, not within them; we don't merely understand objects through their classes. We can see exactly what object sends what message to what other object and we can clearly see and specify where those abilities are assigned.

> A very important goal for DCI is to give the user an illusion of working directly with his or her mental model when working with the computer. Object orientation is the key to the DCI way of achieving this goal.[15]

This mental model we seek is the network of interacting objects. It is the representation of how objects interact with what other objects and when. DCI elevates our perceived network of objects to a readable, executable algorithm. Virtually every definition of "object oriented programming" considers such a network of objects to be its goal.

Shortly we'll be looking at the way DCI makes this happen, but first let's check to make sure we're on the right track by asking: What did the first OO languages intend?

## Where We've Been

Simula 67[16] was designed to represent real world objects in computer code and was the beginning of OOP as we know it. Simula 67 is the first language to introduce the concepts of objects, classes, subclasses, coroutines and more.

The first thing to remember about the first programming language to introduce the world to OOP is that it was designed to *simulate* real world objects. In "The Development of the Simula Languages," Nygaard and Dahl point out that in their experience:

> [I]t became evident that no useful and consistent set of concepts existed in terms of which the structure and interaction in these complex systems could be understood and described.[17]

What were the complex systems they needed to understand? Resonance absorption calculations related to the construction of Norway's first nuclear reactor. This isn't something that you can easily model in miniature and not worry about any nuclear

---

[15] http://folk.uio.no/trygver/2011/DCI-Glossary.pdf

[16] http://en.wikipedia.org/wiki/Simula

[17] Nygaard, K., Ole-Jahn, D. 1981The development of the Simula languages. History of Programming Languages. Association for Computing Machinery, Inc.

radiation from your experiment. At the time, nothing existed for them to better understand their problem and potential solutions, so they simulated it by making a computer world that represented the real world.

Next, let's look to an important description from *An Introduction to Programming in Simula* by Rob Pooley:

> A computer program is a series of instructions which contains all the information necessary for a computer to perform some task.

This quote isn't particular to Simula, but it's an attempt to describe what a *program* is. What's valuable about this is that it describes a program as a "series of instructions." This series implies a step-by-step approach to getting work done.

By following steps, we can stop and evaluate where we are. At any point, we can see how much we have done and how much is left to do. We can understand each step as a unit of work. But how often can you read your code like a series of instructions?

It's pedantic to evaluate your code this way, isn't it? After all, you've followed common patterns and have neatly encapsulated the various responsibilities of your system. Nobody ever sits down to read code as if it's a series of instructions anyway.

## Where We're Going

Nobody reads code as if it's a series of instructions, but couldn't they? Shouldn't they?

We can, and we should.

Step-by-step instructions are often how we think about our business process. Step-by-step is how we evaluate what happens in our business when things work, when things fail, and when things just work differently.

There's no better argument for that than the ever-growing popularity of the Behavior Driven Development (BDD) [18] approach to programming.

In BDD you drive your application design by defining the behavior that you expect. The goal of this approach is not in writing tests, but in writing better code and, more specifically, writing an executable specification.

DCI encourages a focus on representing the real world scenarios that our applications support. This turns out to be incredibly similar to what BDD encourages. The benefit to

---

[18] http://dannorth.net/introducing-bdd/

using an approach like this in executable code is that you're not removed from your program like you are with testing. With DCI you're describing the interactions between objects with the actual objects that interact. In contrast, when you write tests to describe the same thing you're in a completely separate layer: the tests.

This isn't an argument to say that BDD is not valuable, but I want to point out that DCI can help us achieve the same goals that we set to achieve with these other approaches to application design. Starting with tests helps to drive us toward creating object classes that we can understand individually, but DCI pushes us to write software that we better understand in terms of object collaboration and runtime execution.

Later, in **Objects In Context**, we'll explore how to better reveal your intent and describe the program's behavior from within the program itself, but first let's take a closer look at what our objects *are* versus what they *do*.

# Being and Doing

As I write this I'm watching my two oldest boys squirm and climb around a playground with two mountainous structures made of interconnected rope. To them, this is so many things: a boat, a spaceship, a mountain, and who knows what else. They carry out their play as actors of great adventure. I, on the other hand, have the misfortune of seeing through the magic to just the poles, rope, and grommets that hold it all together.

For better or worse, I see these things for what they are: raw materials. This is what the playground *is*.

Your software is a network of objects sending messages to each other for the purpose of fulfilling the requirements of your use cases. The objects in your system tend to represent things in the real world or working mental concepts.

Traditionally we create classes of things and define what their behavior is as we discussed earlier. When we discover that an `Account` needs to be able to transfer money, for example, we find the file that defines the `Account` class and add our new behavior to the class. As we've seen, one problem with this traditional approach is that we tend to grow our classes like weeds adding distraction as our requirements grow.

What we need to do is merely define what things *are,* then focus our efforts on programming what they *do*.

Let's make an example class to facilitate this discussion. Before we do, however, it's important to note that discussing example code in a way that makes it relevant is a difficult task. We could build a large class with many methods and fill up the next few

pages, but by spending time on code we don't want, solely for the illustration of a large and complicated class we'd be getting further away from our goal of understanding use cases. So as we discuss these examples, if you need to, just close your eyes and imagine scrolling through hundreds of lines of methods to get to the one you want.

Let's get started.

In our application we will have end users that need to be activated before being granted access. Here's some code to support that model:

```
class User
  attr_accessor :activation_code, :activation_expires_at,
                :activated_at
  def activate
    activation_code = nil
    activation_expires_at = nil
    activated_at = Time.now
    save
  end
end
```

This class will initialize objects that can be activated by clearing an activation code and expiration date, setting an activation date, then saving the changes. A method such as `activate` is common and illustrates well the problem of unnecessarily complicated classes.

The ability for a user to be activated is only important when that activation needs to occur and at no other time. Over time, we will find ourselves adding more features like this to a class that represents many aspects of the behavior of the system. We complicate our class and make understanding more difficult with every method added. So why do we burden ourselves with this overhead in the `User` class?

The actual activation is something that a user will do, not what it is. Our `User` class is a representation of data and combining what the user is with what it does complicates our mental model. While simple models may support combining data and behavior in one class, doing so can lead to difficulty in responding to change later. The classes that we begin to define should represent data or model the behavior of the data object, **but not both**. This code better represents what the user is:

```
class User
 attr_accessor :activation_code, :activation_expires_at,
                  :activated_at
end
```

And our activation code can be put into a module to be used at the appropriate time:

```
module Activator
  def activate
    activation_code = nil
    activation_expires_at = nil
    activated_at = Time.now
    save
  end
end
```

These two things, the *being* and *doing*, need to come together at some point, but: why are we doing this?

A common approach to refactoring complex classes is to follow this same paradigm of identifying related methods and breaking them out from the main class using modules. Although this may clean up the number lines of code in a given class definition, often developers are just moving code around in their filesystem and their classes end up just as bloated as ever by including these modules in the class at load time. This copy-paste refactoring is well intentioned, but ultimately accomplishes little.

For example the following implementation still has all of the problems we discussed in Chapter 2 with initialized objects having unnecessary abilities:

```
class User
  include Activator
end
```

What we will see with this approach is that our objects still carry with them the ability to activate, regardless of our need for it:

```
> user = User.new
 => #<User:0x1003b1478>
> user.public_methods(false)
 => ["activate"]
```

Keep in mind that we've omitted method definitions we don't care about in this example. If our class included methods and modules for activation, email, registration,

making friends, eating dinner, and more, we would have a lot to sort through. The more we add to this class, the more difficult it becomes to understand its capabilities and limitations.

To avoid this, our class should stand alone:

```
class User
end
```

This example class is extremely simple, but it's only an example. Don't stop here and think that we're forming an argument for empty classes like this `User` class. We'll explore other design considerations later, but first let's look at how an instance of this class gets activated.

## Managing Change

Let's expand our example class to include code you might find in a typical Rails application.

```
class User < ActiveRecord::Base
  validates :name, :presence => true
  validates :email, :presence => true, :uniqueness => true
  validates :password, :confirmation => true
end
```

This class is unlikely to change. It represents users of our program for whom we store a name, email address, and password. Users are users and in general it's not what they are that drives our software development, it's what they do. So this class and our representation of users is unlikely to change significantly.

In fact, if we properly follow the Single Responsibility Principle[19] (SRP), this `User` class should only ever change when our representation of our user changes. New methods and new responsibilities that we create or discover as we develop our system probably do not belong here.

Our mode of activation, however, may change over time. Perhaps at first we merely activate new users automatically. Then, over time, we decide we need to email an activation link. Next we decide we'd prefer to collect payment first. And on and on.

---

[19] http://en.wikipedia.org/wiki/Single_responsibility_principle

By making these changes outside of the `User` class we have a clearer separation of concerns, we can better target a unit test to only a specific piece of functionality, and we immediately gain the ability to reuse this module elsewhere. But, most importantly, we can focus on changing only the code that is directly related to the change in requirements. Of course, we can still change that code if it all lived within the main `User` class, but then we would be mixing the *being* with the *doing*, where changing one inherently changes the other, and we'd be throwing away the values from SRP.

Since the activation of a `User` is a separate responsibility we could create an `Activator` module or class to handle it. We'll initialize another object to handle the responsibility of activation so for now, let's do this as a class:

```
class Activator
  def initialize(user)
    @user = user
  end

  def activate
    @user.activation_code = nil
    @user.activation_expires_at = nil
    @user.activated_at = Time.now
    @user.save
  end
end

activator = Activator.new(user)
activator.activate
```

This class lives on it's own and clearly represents the activation of a user. Over time, as the needs of our application grow we'll know to look to the `Activator` class to make changes.

On the other hand, if we elevate every procedural action to a class like this we'll be adding a lot of new language into our code in the form of classes. Looking at our files we'll have little indication that `activator.rb` has any significant relationship to `user.rb`. Perhaps that's a good thing, or maybe we want things a little clearer and decide to name this class `UserActivator` in `user_activator.rb`. Only the needs of your project and its domain can determine that, but let's take a step back for a moment.

## Implicitness, Explicitness, and Encapsulation

The OO purists might stop right here and claim that we've broken encapsulation. This example shows that we have an `activator` reaching into the `user` object to alter its state.

This `Activator` code assumes that the `user` instance will provide methods that allow the alteration of its attributes. The methods `activation_code=`, `activation_expires_at=`, `activated_at=`, and `save` are all required to exist on all `User` instances by the `Activator`. By doing this, we've made an implicit contract.

Implicit structure is often good. Class methods such as `attr_accessor` allow you to imply the existence of getter and setter methods for the given attribute name. But explicit structure is obvious. Explicit code reveals your intentions far better than implicit code because it shows you exactly what it means. Often the balance between these styles has to do with the preferences of developers writing the code, the complexity of the task, and the common idioms of the language.

What's missing here is a reliable notion of what contract we need to enforce. While any object could implement the required methods, a unit test can help enforce the requirements of the `Activator` class. We'll leave that only as a point of discussion for now, but we'll get back to discussing contracts and tests later in the book.

# Simplifying the Code

We've been thinking about this separation of data and interaction but this feels heavy handed. Imagine every procedure required that we create a new class to handle it. All of these bits and pieces seem like even more to worry about, not less. What problems are we trying to solve here? And why exactly would we go about all this setup just to keep our data objects separate from the interactions?

What we care about is that our code is written well enough that it is easily understood and easily tested. So let's think about our typical path through managing our code with Rails.

If you follow the general approach to building web applications in Rails, then you've likely been down the road of seeing controller methods handling a lot of responsibilities. Often we have them find records, call methods on them, send emails, log details, notify external systems, all within a single action. For example, the following code might be similar to something you've either seen, written, or refactored. Let's take a simple signup

form. Visitors come to your site, they enter their credentials, and receive a welcome email while the administrators also receive an email about their new member:

```
class UsersController < ApplicationController
  def create
    @user = User.create!(params[:user])
    Notifier.send_welcome(@user.id)
    Notifier.notify_admin_of_new_member(@user.id)
  rescue ActiveRecord::RecordInvalid
    render 'new'
  end
end
```

In this code we see a `User` being created and two emails sent out for notification related to that event. If there's an exception during the process of creating the `User`, then we render the `new` page with a simple display of the errors.

As you develop your application you come to realize that you need this exact same functionality elsewhere in your system. Because you've chosen to put this business logic in your controller you're now faced with the possibility of duplicating your code in another controller and causing you more maintenance headaches down the road.

The answer, as you well know, is to solve this problem with skinny controllers and fat models. All of this code related to sending notifications for this event can be pushed into our User model, making our controller skinny (with fewer responsibilities) and our model fat (with more responsibilities). You might decide that callbacks from `ActiveRecord::Callbacks` are the simplest way to go to get what you need. In a matter of minutes you've got code similar to this:

```
class User < ActiveRecord::Base
  after_create :send_welcome, :notify_admin_of_new_member

  private

  def send_welcome
    Notifier.send_welcome(self.id)
  end

  def notify_admin_of_new_member
    Notifier.notify_admin_of_new_member(self.id)
  end
end
```

You can safely go back to your controller and clean things up:

```
class UsersController < ApplicationController
  def create
    @user = User.create!(params[:user])
  rescue ActiveRecord::RecordInvalid
    render 'new'
  end
end
```

Pat yourself on the back, great developer, you have made a skinny controller and a fat model. Your code is DRY! Tell all the world of your refactoring prowess!

You wrote tests for all of this, of course, but I'm leaving them out here for the demonstration.

## WRONG TURN

Later in your development you realize that the administrators prefer to create users themselves. This leads you down the obvious road of creating another controller for almost the same purpose:

```
class Admin::UsersController < ApplicationController
  def create
    @user = User.create!(params[:user])
  rescue ActiveRecord::RecordInvalid
    render 'new'
  end
end
```

This is easy stuff! You've now got another endpoint where only your authorized administrators can go, likely at `/admin/users` I imagine, and you have a nice little interface for them too. During development you realize, of course, that you don't really need to notify administrators that a new user has been created when the administrator did the creating in the first place.

This problem should be simple enough to solve since it all lives in your well tested model. Where was that functionality again? Oh, right; this should go in `notify_admin_of_new_member` which happens after the object is created.

But now we're working with an alternate scenario. Since the method `notify_admin_of_new_member` is called in an `after_create` callback we're stuck with some unpleasant ideas about how to adjust for this new scenario.

How about just setting a flag on the new user like this:

```
class User < ActiveRecord::Base
  attr_accessor :dont_send_to_admin?
  def notify_admin_of_new_member
    Notifier.notify_admin_of_new_member(self.id) unless
dont_send_to_admin?
  end
end
```

This option is awful. In order to understand the behavior of this program you'd not only need to know about the callback, but you'd need to know how to alter it by setting `dont_send_to_admin?` to `true`. It's ugly English too. Perhaps instead we can say `Notifier.notify_admin_of_new_member(self.id) unless ignore_admin?`, but even then it's not beautiful prose and is difficult to quickly understand. No, this setting of flags to skip parts of the procedure is just too prone to cause confusion.

We could create an alternate class method where we delineate the behavior and luckily enough `ActiveSupport::Callbacks` gives us the `skip_callback` method. What about just skipping the callback in a new method:

```
class User < ActiveRecord::Base
  def self.create_without_admin_notification!(params)
    self.skip_callback :create, :after, :notify_admin_of_new_member
    self.create!(params)
    # Turn the callback back on
    self.set_callback :create, :after, :notify_admin_of_new_member
  end
end
```

I feel terrible even suggesting that this is a reasonable solution.

Not only would you need to control the setting of a callback in two places (its original declaration and now in this method) but you'd also be adding another method to load into your head whenever you needed to understand the behavior of this class in the future. When should you use `create_without_admin_notification!` and when shouldn't you? There's an answer that only spending time diving deeper into the code will reveal. Callbacks are often a developers first foray into spaghetti code. Sadly a common first thought of many developers is not to refactor, but to just skip those callbacks they don't currently need.

This need to understand increasingly more things is a problem. When we concentrate on our programming, we should consider how much we need to keep in our heads to understand the task at hand. Often the limited use cases are easy to understand when we write them, or when they are explained to us, but when those use cases are executed in a world where there's much more to understand we easily lose focus. Sometimes our code can feel like Times Square with many new and old things vying for our attention. How many times have you started out writing code to implement a new feature only to be distracted by a misunderstanding, lack of knowledge, or lines of ugly code screaming out at you in mercy to be refactored?

The examples in this chapter are just some of the ways that we solve our problems by making more down the road. Setting flags and adding more methods to a global namespace are just two ways that we increase the cognitive load on ourselves just to understand something that, especially in our example, should be extremely simple. If this

example is difficult to keep simple, what will it mean for our application as its list of features grows?

Despite knowing our needs for business logic will change and have exceptional cases, we still tend to interweave *being* and *doing*. The more we combine these two things into one, the more knowledge is required to understand the one thing. Our knowledge deficit always grows as our features grow, but with this class-oriented approach our deficit grows at a greater rate. This creates a gap in our ability to quickly and effectively continue development.

By loading our data classes with behavior for all aspects of their use we increase the demand on our brains to understand and we create gaps in our understanding. The more we need to understand, the harder it is to understand. What we really want to have is a much more even and high level of understanding of when, where, and how the methods in our program are used.

# Objects in Context

The answer to our woes of requiring so much knowledge to understand simple things is to merely *control our perspective*. DCI is an approach that removes the distraction of unrelated code. Our runtime objects don't have those extra and distracting abilities, likewise our written classes aren't cluttered with extra features, and our brains aren't burdened with understanding them outside of the context in which they are needed.

We've looked at some tangled ways we write code for our needs and how that will drastically increase our cognitive load. By lumping our responsibilities in to a single class and attempting to skip the parts we don't want for a particular part of our program, we split our focus for related parts of code, and combined responsibilities for unrelated parts. This is the opposite of what we want for easy to understand code.

We've seen how we can separate *being* and *doing*, but that alone isn't enough to make a useful tool. We need to fully understand when, where, and how to bring these parts together. We need to worry about codifying our business logic in a way the reflects the working whole, rather than a way that defines the individual parts.

Typically we organize what happens in our Rails applications in controllers. A web request hits the application server, the Rails router determines which controller is responsible, and the controller does its thing.

Let's look at our earlier example of transferring money between accounts to see how we can apply the idea of a DCI context.

We'll set our router to map a POST to our `TransfersController`:

```
AccountManager::Application.routes.draw do
  resources :accounts do
    resources :transfers, :only => [:index, :create]
  end
end
```

This sets us up with paths to manage accounts and their transfers. What we're currently concerned with is the request to initialize a transfer. We can setup our `TransfersController` to handle this request:

```
class TransfersController < ApplicationController
  def create
    account = Account.find(params[:account_id])
    account.transfer_to(params[:destination_id], params[:amount])
    redirect_to account_transfers_path(account)
  end
end
```

This first example is a typical Rails controller action. It's written assuming that the initialized `account` can call the `transfer_to` method. We've discussed some reasons why we'd not want `account` to have that ability inherently, so let's alter this example to add it only now that we need it.

```
class TransfersController < ApplicationController
  def create
    source = Account.find(params[:account_id])
    destination = Account.find(params[:destination_id])
    source.extend(Transferrer)
    source.transfer_to(destination, params[:amount])
    redirect_to account_transfers_path(source)
  end
end
```

This example assumes that you've got a `Transferrer` module that defines `transfer_to`, but we can leave the implementation up to our imagination for now.

Your typical test for this controller in Rails might look something like this:

```
class TransfersControllerTest < ActionController::TestCase
  test "POST#create" do
    source = Account.new # with a balance of 900 by default
    destination = Account.new # with a balance of 900 by default

    Account.expects(:find).with('1').returns(source)
    Account.expects(:find).with('2').returns(destination)

    post :create, :account_id => '1',
                  :destination_id => '2',
                  :amount => '500'
    assert_equal 400, source.balance
    assert_equal 1300, destination.balance

    assert_redirected_to account_transfers_path(source)
  end
end
```

This example code uses the "mocha" gem[20] to provide for some simple mocking.

This controller test checks that a post to the `create` action will find the accounts with the given `account_id` and `destination_id`, transfer the `amount` between them and then redirect to the list of transfers.

You might argue that we're doing this backwards. I should have shown you the test first to drive the design and then show the code. We'll get to that, but I want to start with something that I've found to be typical. Many developers write code and tests this way so it's likely that you've done so as well, but our focus is on setting up what's typical, not discussing how we got here.

We've already discussed a problem with leaving logic like this in a controller: if we need the same behavior again, we might find ourselves duplicating code. The example we discussed previously with our `User` moved our repeatable code into the model by using ActiveRecord's callbacks. We can't use that same idea here because we're not performing any action on the account that would trigger a callback in a way that wouldn't be messy. This example is simplistic and there's always the YAGNI argument: You Ain't Gonna

[20] https://rubygems.org/gems/mocha

Need It[21]. But will our application architecture help us or hurt us when we finally do need to reuse this code?

In any application where we can transfer money between accounts, we could also provide the option to pay bills; this can be just like our transfer example. What do we do when we want to do a transfer of money to multiple accounts at once? Can we reuse the code in this controller the way it is? Perhaps we could if we automated sending requests to it, but even the thought of that should immediately cause you to wince at the unnecessary complexity. Imagine explaining that solution to your colleagues.

## Clearer Perspective

This is where the ideas in DCI can help improve our architecture. In our example we have two accounts where one acts as the source and the other acts as the destination for the transfer of funds. As we saw in the chapter **Being and Doing**, we can separate the responsibilities of representing what the system is from what it does. Our `User` objects control our data, and our `Activator` was responsible for maintaining the behavior.

The **context** is the object responsible for organizing the data and roles for carrying out your use cases. It's a powerful model for encapsulating your business logic because it gives you a clear way to organize the interactions of objects in your system.

We'll review an example of a DCI context, but first consider common approaches to organizing logic. Just in the examples we've discussed so far we've seen application logic in controllers, models, and even split into modules or classes to be added to or used with data models later.

Where is the business logic? In some cases it's in our controllers, in others it's in our models, and still in other cases it's in modules and classes too.

While we might build a working and well-tested system like this, we'll find that as our program grows, we'll have more difficulty understanding how the pieces fit together. The more pieces we add in the absence of a clear structure guiding us about how they fit together, the harder it will be to understand exactly how the system functions.

Often as a program grows the team of developers grows with it. Some new developers are added, some leave and are replaced by others. How can you, as the owner of your code, hand over a program and make it easy to understand? Does a successful hand-off

---

[21] If English isn't your primary language this might look strange to you. Both "ain't" and "gonna" are slang terms. A proper expansion would be "You Aren't Going to Need It."

require a long explanation? Do you rely on a "read the tests" approach or are you sending new programmers to go "read the documentation?"

*Wait a minute! Are you arguing against documentation or even tests!?*

No, I'm not, but these additional artifacts require more of our limited attention spans. Going from tests to code or documentation to code changes your locus of attention. Your locus of attention is the single point on which you are able to concentrate your mind. You're human and you can only focus on one thing at a time.

As you change your focus from code to documentation, you might need to remember what line of code you were reviewing, or what it meant. You might also need to recall other parts of a method or class, and perhaps need to recall what some other part of the documentation said or what was happening in the tests before and after that line is run.

By pointing to tests and documentation to understand a system we are adding more complexity. What we want is less. The goal is to have less for us to know in order to better understand the system. It should make sense without a deep dive because that would be faster. Our locus of attention should change as little as possible. We should be able to rely on our intuition and immediate needs to get going.

*Will DCI make this magical codebase with no tests and no documentation happen?*

No. You should still write tests, you should still write documentation where you need it, but if your goal is to understand your program, your program code should be your best source of that understanding. Make it so.

In my experience, the power of the DCI context object will get you closer to this intention-revealing code far better than any other method, so let's take a look.

## Describing the System

Let's start off with a description of DCI by Coplien:

> The "D" is for data modeling: what we know as traditional objects, though bereft of knowledge about scenarios. It captures the static structure of objects[...]
>
> The "C" is for context: the mapping of roles onto objects on a per-use-case basis[...]
>
> The "I" is for interaction: an algorithm of a stateless role, written in terms only of other roles, that defines in readable terms what the system does, and how it does it.[22]

Let's start by focusing on context's "mapping of roles onto objects." Previously we've explored the separation of data objects and their responsibilities but we pulled them together in our controller. This is a logical place for that to occur in a Rails application, but as we saw it doesn't lend itself to reuse, and if you begin to add this feature to your controllers you'll find they become bloated with responsibility. We can limit the controllers to handling requests and responses, and put our business logic into contexts.

```
class MoneyTransfer
   def execute(amount)
      @source.transfer_to(@destination, amount)
   end
end
```

This sample code shows us the behavior that we want; a source is transferring some amount to a destination. What's left out of this is the setup about how source and destination are pulled into the algorithm and given their responsibilities:

```
class MoneyTransfer
   def initialize(source, destination)
      @source, @destination = source, destination
   end
end
```

In this context we now have the concept of roles. Although we will pass accounts to this context when it's initialized, we want to refer to them in terms relevant to our context. We call these **Methodless Roles**.

Our concern with DCI is with the interactions between and among roles. Objects in your program play different roles at different times during its execution. It doesn't really

---

[22] James Coplien - https://sites.google.com/a/gertrudandcope.com/www/thedciarchitecture

matter to the use case if the `@source` is a `SavingsAccount` object or a `CheckingAccount` object or something else altogether. A Methodless Role is the role an object plays in the execution of an algorithm. It is a marker signifying an actor performing a part in a play.

Here we have two accounts. These two objects aren't called `account1` and `account2` for a very specific reason: those names have no meaning relevant to our business logic. Our actual use for these objects is that one will be the source of money, the other will be the recipient of a transfer, so a good selection of Methodless Roles are `source` and `destination`.

Choosing these names is important in the communication about what this code is doing. Were we to use placeholder names with no relevance, we'd be forcing the future readers of the code to slow down merely to understand what each item is. Good names grease the wheels of understanding.

In an excellent and well named book called "Clean Code," Robert C. Martin succinctly explains:

> Choosing descriptive names will clarify the design of the module in your mind and help you to improve it. It is not at all uncommon that hunting for a good name results in a favorable restructuring of the code.[23]

Naming has everything to do with communication. The Methodless Roles communicate the purpose of each object with a descriptive name and the purpose of your code is not only to create working software, but, most importantly, to communicate to you and other people about its purpose.

Programs can easily be written for computers to understand, and the litmus test is that it works properly. If, however, a person is ever to read, write, or edit the code, then human communication becomes significantly more important rather than merely working properly.

Back in our context, we're still missing something. We're missing the ability to do anything. If our accounts are merely representations of funds with no inherent ability to transfer money, this context will do nothing but raise an error when we get to the `transfer_to` method call. Let's alter the context a bit to be sure we've gotten this right:

---

[23] Martin, Robert C. *Clean code: a handbook of agile software craftmanship*. Upper Saddle River, NJ: Prentice Hall, Copyright 2009. informit.com/ph

```
class MoneyTransfer
   def initialize(source, destination)
      @source = source
      @destination = destination
      assign_transferrer(@source)
   end

   def execute(amount)
      @source.transfer_to(@destination, amount)
   end

   private

   def assign_transferrer(account)
      account.extend(Transferrer)
   end

   module Transferrer
      def transfer_to(destination, amount)
         self.balance -= amount
         destination.balance += amount
      end
   end
end
```

A complete example would include transactions for the transfer procedure as well as other needs like logging but we're keeping it simple and only manipulating a value in this case.

We have just added what we call a **Methodful Role**. While we had our objects playing roles in this context as `source` and `destination`, these roles had no special functions other than to clarify the purpose of your logic. The `Transferrer` is the Methodful Role played which, as its type implies, adds methods to the data object.

## Implementing Your System

We want to be sure that we know how to use this new context and we want to test the before and after states of the accounts to be sure that this transfer is handled properly:

```
class MoneyTransferTest < ActiveSupport::TestCase

  test "initializes with a source and destination" do
    assert MoneyTransfer.new(Account.new, Account.new)
  end

  test "initialization errors without 2 objects" do
    assert_raises(ArgumentError) {
      MoneyTransfer.new(Account.new)
    }
  end

  test "#execute subtracts the given amount from the source and
        adds it to the destination" do
    source = Account.new
    source.balance = 2000
    destination = Account.new
    destination.balance = 0
    transfer = MoneyTransfer.new(source, destination)

    transfer.execute(99)

    assert_equal 1901, source.balance
    assert_equal 99, destination.balance
  end

end
```

Now we have something useful and reusable, but what does our controller look like?

```
class TransfersController < ApplicationController
  respond_to :html
  def create
    source = Account.find(params[:account_id])
    destination = Account.find(params[:destination_id])
    transfer = MoneyTransfer.new(source, destination)
    transfer.execute(params[:amount])
    respond_with source do |format|
      format.html{ redirect_to account_transfers_path(source) }
    end
  end
end
```

Although we've put our business logic in a central and reusable container, this controller action is still quite complicated. We can simplify its responsibilities. I'd much prefer that it be as easy to write as this:

```
class TransfersController < ApplicationController
  respond_to :html
  def create
    transfer = MoneyTransfer.new(params[:account_id],
                                 params[:destination_id])
    account = transfer.execute(params[:amount])
    respond_with account do |format|
      format.html { redirect_to account_transfers_path(account) }
    end
  end
end
```

Let's alter our test of the transfer context to reflect this:

```
class MoneyTransferTest < ActiveSupport::TestCase

  def setup
    @source = Account.new
    @destination = Account.new

    @source.balance = 2000
    @destination.balance = 0

    Account.expects(:find).with(1).returns(@source)
    Account.expects(:find).with(2).returns(@destination)

    @transfer = MoneyTransfer.new(1, 2)
  end

  test "#execute subtracts the given amount from the source and
        adds it to the destination" do
    @transfer.execute(99)

    assert_equal 1901, @source.balance
    assert_equal 99, @destination.balance
  end

  test "execute returns the source account" do
    assert_equal @source, @transfer.execute(99)
  end

end
```

Our tests will fail without altering our context with the ability to find the appropriate objects, so we'll add that ability now:

```
class MoneyTransfer
   def initialize(source_id, destination_id)
     @source = find_account(source_id)
     @destination = find_account(destination_id)
     assign_transferrer(@source)
   end

   private

   def find_account(id)
     Account.find(id)
  end
end
```

Here we've changed the behavior as we intended. Our controller will pass along the appropriate argument and the context takes care of finding the relevant objects and assigning them their roles. Finally, the controller tells our context to `execute` and returns the response.

## All Set Up

We've managed to put our business logic in a place that's reusable and can easily be tested, and we've implemented a system that doesn't rely on inherent behavior from our classes. The value here is that we have a clear definition of the behavior and we won't need to go diving into class definitions to figure it out.

When explaining our implementation to our business owners or to new developers on our team, we can point here and say, "this is how it works." Instead of answering "what can Accounts do?" we can answer "what does the system do?"

With our focus on the program's duties we can more quickly react to changes. Our reaction to changes in business logic is faster because we've put the business logic in plain view in an object that describes our interacting data objects. The `MoneyTransfer` context in this example centralizes our understanding of the system.

Will we do this for every procedure that our application needs to perform? Not quite. We're not implementing our entire system with procedures, but we do have a better way to organize our procedural flow. Beyond this small example we have a powerful tool to organize the *doing* part of our program outside of its *being*.

There's a lot more to implementing business logic than this. To better manage our flow we need a way to organize related scenarios, and we'll explore that in the next chapter.

# Gazelles And Gazebos

> "A user story is to a use case as a gazelle is to a gazebo"
> —Alistair Cockburn[24]

Many projects are begun by gathering user stories[25] to describe the end user behavior when completing tasks. User stories, which describe an end user's behavior, are often helpful in describing tasks which a developer may translate into code, but use cases are a fundamentally different thing.

A use case helps us to describe multiple scenarios for a business requirement. When writing a use case you'll say that you have a specific preconditions, specific actors, and of course specific goals.

A short summary to help you remember comes from a comment on Cockburn's website by Ted Husted:

> Another answer is that you can fit a gazelle inside a gazebo just as you can fit a user story inside of a use case. And in both cases, the inverse would be cruel.

A use case will describe at least two exit points for a business process: the success and the failure. Often there are many variations and multiple exit points in related scenarios.

---

[24] http://alistair.cockburn.us/A+user+story+is+to+a+use+case+as+a+gazelle+is+to+a+gazebo

[25] http://en.wikipedia.org/wiki/User_story

We can look to Cockburn[26] again for the full understanding of use cases. Here are the main points about what a use case should do and some points about how we can apply this to our executable code. In Cockburn's words (with my comments afterward) use cases DO:

- *Hold Functional Requirements in an easy to read, easy to track text format.* We can do this in code. Ruby is widely known for being easy to read and the community values beautiful code. As far as holding functional requirements, that should be very simple for executable code to do.
- *Represents the goal of an interaction between an actor and the system.* The goal represents a meaningful and measurable objective for the actor. This is both an important aspect of your business, and an important aspect of DCI. When we're attempting to achieve our business goals, we should write software that is uniquely designed to do that. The context is an object that encapsulates this concept.
- *Records a set of paths (scenarios) that traverse an actor from a trigger event (start of the use case) to the goal (success scenarios).* In simple code this is easy enough to do with if/else blocks or case statements, for example.
- *Records a set of scenarios that traverse an actor from a trigger event toward a goal but fall short of the goal (failure scenarios).* An example here might be the above if/else blocks or perhaps a rescue from an exception. A use case describes a complete interaction between the user and your system and it is the responsibility of your context to implement this.
- *Are multi-level: one use case can use/extend the functionality of another.* This is reflected in DCI in the fact that we want to achieve the vision of Alan Kay, to create a network of interacting objects much like biological cells or computer networks. A context can trigger other contexts within.

What you're attempting to do with DCI is not battle junk drawer objects with other junk drawer objects, but to implement business logic in an organized set. Take a specific need and describe it, including variations, using a single context to coordinate the data and interactions. The context has real meaning and real value to your business, it's not just a place to call `extend` on your objects.

In Rails, your controllers should handle the user actions that trigger these use cases. You might have multiple ways to trigger a use case. For example, in a typical view your

---

[26] http://alistair.cockburn.us/Use+case+fundamentals

user can interact with objects in your system, but in admin view another user can do the same with perhaps an alternate scenario allowing him to override certain aspects of the scenario. It makes a lot of business sense to look at this use case and scenarios together, so why not put that code into one place? Why not create context that explains all of this in executable code.

Put your use cases in executable code and trigger them from wherever your interface requires it. Before you attempt this, begin first by writing your use cases. Use cases reveal the needs of the program and show the value of the DCI context in centralizing your business needs.

# Crossing the Use Chasm

My former business partner always used to remind me that "software is never finished, you just stop working on it." There's always a new perspective, a new way to simplify code, a new feature to add, and there's always change. Good application architecture will allow you to respond to change.

We can write use cases to clarify the requirements of our program. Creating a context helps us to implement those requirements in executable code. With this approach we can leave the decisions about incidental support, such as data persistence, to a later time, focusing on the semantics of what our system will *do* during its operation.

We've seen code that reflects ideas for user activation and money transfers between accounts but it is useful to work with a more complex scenario to experience how this approach can clarify your program. Let's begin by defining a use case and then look to Ruby to help us make it reality.

## Framing The Problem

When working with my own finances, my worries are put at ease by asking an expert for advice. I review my account statements and just pick up the phone or fire off an email to get the answer I need.

We'll be building an application so let's create a scenario for this use case: *a user submits a question and receives an answer from an expert*.

Get yourself prepared in your command line and `cd` into wherever you manage your source code.

```
rails new ask_the_experts
```

With that command we've got a standard Rails application framework ready to go.

What we're not going to do is jump into defining our models. We're not going to begin defining a user or an expert. We're working on a program to allow people to help each other so let's spend our time modeling that interaction.

```
touch app/models/expert_questioning.rb
```

Now we've got our first file with a name signaling it's purpose. This is the context for describing our use case. Here is where we'll begin to define our system.

## A Note On Naming And Code Use

You might be tempted to give yourself some other signal about what this is. Perhaps `expert_questioning_context.rb` seems attractive as a filename. It's a context, after all, and Rails requires that we name our controllers with a similar pattern. So why not take a note from the book of Rails?

File appendages, in the case of Rails controllers such as `users_controller.rb`, are a great way to simplify the setup of an application. The `_controller` convention allows our code to lean on the framework for support in what would be mundane and repetitive work.

The convention for the `_controller` appendage in Rails simplifies the setup of routing requests to the appropriate class to handle them. A `UsersController` will be automatically initialized when a request matches the `:users` route. Almost no thinking needs to be done by us to decide how to handle the web requests. Follow the naming convention and the magic takes care of it. Without a naming convention, the Rails router would likely require more manual configuration to get the appropriate class hooked up to handle the request.

Additionally, as third-party framework, Rails takes control of names that we likely wouldn't choose for our own code. By favoring the `Controller` appendage for controller classes, Rails will take the name `AccountController` for its needs, allowing us to make

use of the `Account` name for our domain objects. To paraphrase a Bible verse[27]: *render unto Rails things which are Rails', and render unto your domain the things that are your domain's*.

Choosing to add `Context` to our context classes and `_context` to their filenames does nothing to simplify our code nor make magic happen. It's unlikely that you'll find a need for this and we certainly don't have any need for it now.

Perhaps you've already jumped ahead and chosen to use a namespace within a module. A namespace allows you to organize your ideas, and you might feel it makes sense to put a file in `app/models/contexts`. That sets you up to refer to your class as `Context::ExpertQuestioning`.

A namespace does exactly what the word suggests: provides space for the name to exist. But it does this, primarily, to avoid clashes in names. These two classes, for example, are entirely separate:

```
module Context
  class ExpertQuestioning; end
end

class ExpertQuestioning; end
```

We're at the beginning of our development and we have neither of these classes yet defined. We have no reason to choose the more verbose `Context::ExpertQuestioning` name. By choosing to nest our class inside a namespace we would be signaling to others that we don't want this class to clash with the other class named `ExpertQuestioning`. The other class, however, would never exist.

Lastly, regarding our class name, a *gerund* is often an intention revealing choice for a context. A gerund, a word that functions like a noun but is derived from a verb, helps us to describe what the system is doing in simple terms. Our domain models are **objects** represented by nouns, and our object interactions are **methods** represented by verbs. Our context is something **happening** represented by a gerund.

We're likely to find alternates to these ideas and only the needs of a program and the terms we use to describe it will best determine good names, but this will guide us toward code that helps both us and others understand.

---

[27] http://en.wikipedia.org/wiki/Render_unto_Caesar

Keep your entire team involved in the language you use to describe your business process. Too often developers choose names that make sense to a developer but which need to be translated in discussion with a non-programmer business analyst. Think about how you will describe your process in spoken language before you put it into terms of a programming language.

## Expanding Our Use Case

Now that we've got our context created, let's open it up and begin.

First, we'll have our use case triggered by a regular user of the system. This is our **Primary Actor**.

```
# Primary Actor: a regular user
```

Next, we want to specify that our **Goal** is for the user to gain information related to her question.

```
# Goal: user gains information about a specific finance question
```

We should list any **Supporting Actors** so that we have a better understanding of who is involved. These may be additional users, other parts of your system, or other systems entirely.

```
# Supporting Actors: a financial expert
```

And we'll specify that we expect our user to be allowed to trigger the use case in our **Preconditions**. We don't currently have any need to be concerned with any aspect of either of authentication and authorization so we'll begin by just expecting that they are already solved.

```
# Preconditions: user is authenticated and authorized
```

We begin writing our code like this because it quickly answers questions that we'll have. Who is doing this? What is the goal? Who else is involved? And what else do I need for this to work?

Without creating any working code, we've begun with a clear understanding of what this program is going to do. With these initial comments we use BLUF (Bottom Line Up Front[28]) to get to the important information faster. This allows us to focus on the concerns of our current problem by limiting the things which we need to understand. We'll also ignore other parts of our larger system we don't need to understand.

---

[28] http://en.wikipedia.org/wiki/BLUF_(communication)

## Preparing The Context

Because our context is an object in our program, let's begin be defining what we need to bring it to life. To get started, we know we'll need someone to ask a question, and we'll need her question. We'll begin with the specification so let's make the files.

```
mkdir -p spec/models
touch spec/models/expert_questioning_spec.rb
```

Next we specify the requirements of the initialization method where we define what arguments we'll need.

```
describe ExpertQuestioning do
  it 'initializes with a questioner and a question' do
    questioner = Object.new
    question = "what?"
    lambda{
      described_class.new(questioner, question)
    }.should_not raise_error
  end
  it 'errors without a questioner and question' do
    lambda{ described_class.new }.should
       raise_error(ArgumentError, %r{0 for 2})
  end
end
```

The `described_class` method in RSpec will return the class used in the current `describe` block. This helps to keep our focus on the behavior of the object under test. Additionally, if we decide to change the name of our class, we only need to change the class and the describe block.

We could run this file with `rspec spec` but we know we'll run into the `uninitialized constant ExpertQuestioning (NameError)` because we haven't required our model file yet. Let's do that by taking a tip from Avdi Grimm's book Objects On Rails[29] by making a simpler helper file giving us only what we need.

```
touch spec/spec_helper_lite.rb
```

Typically we'd load the `spec/spec_helper.rb` file generated by RSpec but it will boot up our Rails application and we don't need that here. Later, and in other tests, we may want to load the entire application so rather than alter the default helper file, we'll create our own.

---

[29] http://objectsonrails.com/

To that we'll add references to RSpec's libraries (or another testing library if you prefer).

```
require 'rspec/autorun'
```

And lastly we'll add references to our necessary files in our `spec/models/expert_questioning_spec.rb`. We've done some back and forth, so here's what our entire test looks like now:

```
require 'spec_helper_lite'
require_relative '../../app/models/expert_questioning'

describe ExpertQuestioning do
 it 'initializes with a questioner and a question' do
    questioner = Object.new
    question = "what?"
    lambda{
      described_class.new(questioner, question)
    }.should_not raise_error
  end
  it 'errors without a questioner and question' do
    lambda{ described_class.new }.should
       raise_error(ArgumentError)
  end
end
```

We can run this spec file with `rspec spec` and we'll see that it fails because the class is not yet defined. Then, if we define the class and run it again we'll see that we need to define the initialize method because we'd have an `ArgumentError`. Let's cut through all that and just make the class definition. Here's what our model looks like now:

```
# Primary Actor: a regular user
# Goal: user gains information about a specific finance question
# Supporting Actors: a financial expert
# Preconditions: user is authenticated and authorized

class ExpertQuestioning
   def initialize(questioner, question_text)
   end
end
```

Now we can see our first success.

```
rspec spec/
..

Finished in 0.0005 seconds
2 examples, 0 failures
```

## Decoupling the File System

The `require_relative` method is useful for pulling in relevant files for our tests, but the path argument stinks of coupling to the file system. We've specified that our model file will be two levels up and then two levels down in another part of the directory structure with `'../../app/models/expert_questioning'`. This may seem unimportant, but being able to move things around in an application without having to rewrite irrelevant details like the path to a file allows us a bit more freedom if we decide we need some reorganization.

Instead, we can specify what our test needs and use simple `require` statements. Here's a helper method to use in `spec_helper_lite.rb`.

```
def needs(path_fragment)
  full_path = File.expand_path(Dir.pwd + '/app/' + path_fragment)
  unless $LOAD_PATH.include?(full_path)
    $LOAD_PATH.unshift full_path
  end
end
```

In our test we can replace our `require_relative` statement with:

```
needs 'models'
require 'expert_questioning'
```

While one could argue that we still have a reference to our file system here, but we're also free to change that. Although we might have our classes in `app/models` we could alternatively use `lib/ask_the_experts/models` or somewhere else that makes sense. All we'd need to do is change this one method in our helper file to accommodate that.

Why do this at all? Well, we could automatically add `app/models` to our load path whenever we run these tests without ever creating the `needs` method. With all paths loaded we could simply rely on `require` to do its job, but we may want more control over the load path later. The more we add to our load path for every file that we need, the more overhead we'll have in just booting up to run our tests.

It's better to keep things simple at this point by keeping our load path small.

## Defining Success

Now that we have our tests running, we can implement the behavior for the success path of our program. So let's begin discussing success.

Our goal is for a user to gain information about a finance question. What's the scenario for making that happen? There are a few things we'll need to do.

Our experts probably won't be sitting around waiting for questions to come in so we'll need to notify them when a question needs their attention. Likewise, we'll need to notify the user when the question has been answered. Let's take a look at this scenario.

The **Trigger** for this use case is the submitting of a question by the **Primary Actor**.

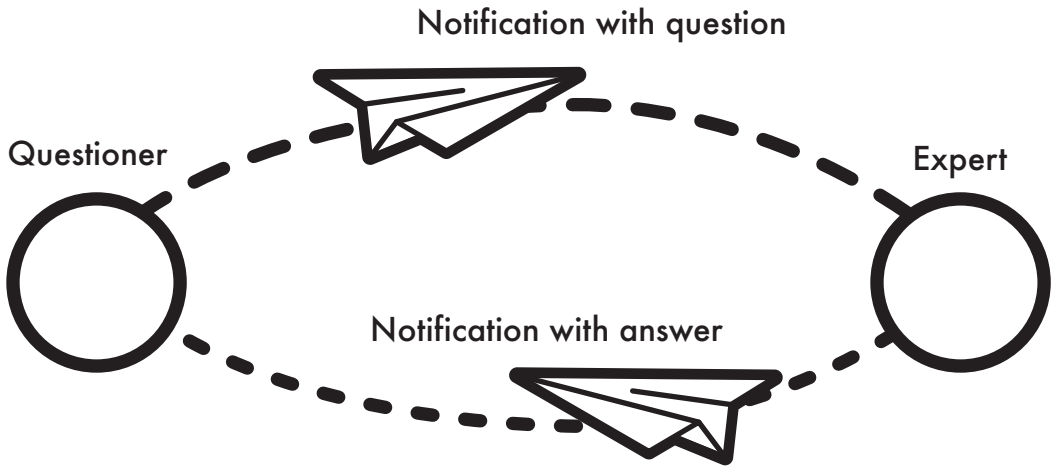
1. A notification is sent to the user that her question will be answered soon.
2. An available expert is assigned to the question.
3. The expert is notified of the unanswered question.
4. The expert submits an answer to the question.
5. The user is notified of the answer to her question.

There are two main parts to this procedure. In one part we see the actions triggered by the Primary Actor (steps 1, 2, and 3) and in the second, the actions triggered by a Supporting Actor (steps 4 and 5).

This describes a value to the business. We're not discussing what makes an expert nor are we defining a notification system. The sole purpose with this procedure is to define exactly what needs to happen for us to be successful. Our success comes when these steps are completed and our Primary Actor gets her answer.

An important point to remember in defining our use case is that it describes what the actions are and not what the actors are. While it does mention the actors, there's no opinion about what makes an actor. There's no mention of implementation: no parameters, no database, no extra details at all.

This procedure gets to the heart of the matter. When we want to know what to do with our money, we ask the system and we're notified of an answer from an expert. Our implementation can be as simple as throwing paper airplanes around for a notification delivery system.




It's fun to fold up some paper and toss a plane across the room but the larger point to remember here is about the human aspect of our software. Our goal is not to push bits around in a computer system, our goal is to help people get what they need. By driving our design with a use case, we remove the distractions of technical thinking.

Because the success of this procedure is our goal, we make the procedure a first-class citizen in our code. That's not to say that every procedure will become it's own class, and we don't want to live in the Kingdom of Nouns[30], but when we have interaction between

[30] http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html

and among objects, a use case driven approach with DCI can narrow our focus to what really matters.

There's more, however, to a well-defined use case than a single procedure. The five steps we've created represent just one scenario. There are also **Extensions** to this use case that we'll see where we have alternate outcomes and failures.

## BRAINSTORMING

When we form a mental model, for any task, we typically do just enough to understand and operate effectively. That is to say that we form functional, but incomplete models.

When *programming* a system model, however, we need to spend more time and effort exploring the possibilities. As our program is used, we're more likely to run into edge cases and alternate outcomes that our simple mental model didn't cover.

What do we do when we fail at any point in the system? What alternatives are there to our main success scenario? Are there other actors we haven't considered?

Here are some Extensions for our procedure:

2a. No expert is available

2a.1. The user is notified that no expert is available for her question

4a. The expert determines that the question is answered already.

4a.1. The expert assigns the question to an existing answer

There's so much that we could do, too much to explore fully in this book, but this is an exercise for every team member. We can learn from the Lean[31] approach that software development should include this upfront thinking about the domain with as many stakeholders of the system as possible.

The approach to determining the aspects of this use case should be exhaustive. For now, we'll stop here because we have enough of an understanding to move on.

Even in this short bit of brainstorming, we've found a problem. Our success scenario first notifies a user that her question will be answered soon. Then if no expert is available we notify the user that no expert is available to provide the answer. That would be quite confusing to receive two contradictory emails.

---

[31] http://en.wikipedia.org/wiki/Lean_software_development

This thinking up front, *before* writing our code, is at the heart of BDD and helps us find problems like this. We won't always save ourselves from mistakes this early in the process, but hopefully we can establish an approach that will make it more likely.

Now we can see that we should swap our first and second steps from our main success scenario.

1. An available expert is assigned to the question.
2. A notification is sent to the user that her question will be answered soon.

## Setting Boundaries

With our code and tests hooked up and our needs described we can begin digging into the dirt to grow this use case into something functional. We'll start by thinking about the language we'll use to describe this use case.

Our `ExpertQuestioning` will begin when a user submits a question. This is more, however, than just a single trigger. We're building a Rails application that requires two users to act, so we'll need a break in the procedure where we wait for the expert to answer a question.

The terms we use should be simple to understand. In previous chapters we've built code to execute a money transfer. Later, we'll find that as we build our applications we need to think more about the terms we use to describe how we interact with them.

We're venturing into mental models that are more complex, so it's important to spend time considering the names we use and how they affect our communication. What triggers the context? Will an `execute` method always reflect our understanding? Perhaps, perhaps not.

There's a definite benefit to following naming conventions. Doing so limits the required information for understanding a system. But there is a downside too. Our use case is a description of system behavior and we should focus on communicating the business process in our code not fitting our process to our code. The conventions we follow should be derived from implementing our business processes, not the other way around.

Discussing, with another developer, the way we will use our code will help us choose terms that will reveal our intentions. A good way to begin this discussion is to ask: *how would you expect to use this?*

Would you "execute" this use case?

*No. I don't think about it that way. We have a person asking a question, I don't "execute" my questioning.*

We have multiple triggers for this too, since we're splitting the use case when we wait for the expert. Why not just choose "ask" and "answer"?

*Well, we could, but our primary actor is asking and our expert is answering. This isn't what the context is doing. And our actors might do other things too.*

Let's go with "start" and "finish" since we could say that we'd start with asking and finish by answering.

```
class ExpertQuestioning
  def start
    # 1. An available expert is assigned to the question.
    #   1a.   No expert is available.
    #   1a.1  The user is notified that no expert is available.
    # 2. A notification is sent to the user that her question
will be answered soon.
    # 3. The expert is notified of the unanswered question.
  end
  def finish
    # 4. The expert submits an answer to the question.
    #   4a.  The expert determines that the question has already
been answered.
    #   4a.1 The expert assigns the question to an existing
answer.
    # 5. The user is notified of the answer to her question.
  end
end
```

We chose our method names in an imaginary discussion because this is a book, after all. But even a solo programmer should go through an exercise imagining how he or she will understand the code later. The next developer to read the code might not be someone else, it might be you or me with six months of time since the last time we read it.

The discussion of the code, its use, and how it's related to our business process is an important aspect of designing our system. When you are pair programming, doing Test Driven Development, or both, you drive the design of your application like this. These

approaches help us to consider what we should do, how we should do it, and what it will communicate to the next person to read it.

Now we've got a basic shell for our context after a short discussion, but we'd better move to our tests to ensure that we don't build what we don't need and give ourselves a safety net if we decide to start making changes.

```
describe ExpertQuestoning do
  context ‘when starting the questioning’ do
    it 'assigns an available expert to the question' do
      pending
    end
    it 'notifies the questioner that the answer is queued' do
      pending
    end
    it 'notifies the assigned expert of the question' do
      pending
    end
  end
  context ‘when finishing the questioning’ do
    it 'answers the question' do
      pending
    end
    it 'notifies the questioner of the answer' do
      pending
    end
  end
end
```

We're looking at a variety different actions for this use case. Every one of them can be implemented in a different way and our initial mental model of creating an ask-the-experts application begins, here, to show just how complex it is. This example still only includes the main success scenario.

All of these tests are pending for now, and each one can be a point of discussion for the implementation. Indeed, each item could be another use case altogether. We'll go with what's simple in the beginning, but how you or any other programmer might approach this will vary.

We've discussed structuring our code, but more importantly we've looked at ways to pull details from our mental model. Thinking about what we need our program to do often seems less important than building the parts. It's much easier to jump into creating a database and all the models required for the application before writing the methods they'll use.

Next, we'll dive into ways this context can be implemented and how even our implementation might contain sub-contexts to apply more fine-grained requirements.

# Screenplay In Action

Building your context takes work. Evaluating the possible extensions for your use case and determining the actors helps you plan the path. But eventually the time comes to dive into implementation.

We've got our `ExpertQuestioning` context prepared. Now we need to begin deciding how we'll implement the parts.

## Finding the Expert

The first aspect of the scenario is the assignment of an expert. The simplest approach is to just take the first expert that we find

```
expert = Expert.first
```

The obvious problem is that we'll just get back an expert having nothing to do with availability. Determining availability may become a complicated matter. Is an expert on leave for some time or no longer working? Is any given expert capable of answering the question? We can avoid making decisions about that merely by kicking the problem down the line. For now, we could just use the code we want to have:

```
expert = Expert.first_available
```

But even beyond this desired code could be yet another use case.

How is someone an expert? Does an expert volunteer themselves? Is expert status determined by calculating votes of other users? Perhaps there are legal regulations governing the qualifications of our experts.

We could sit and think of dozens of ways to determine how someone is an expert. But right now we don't care about exactly how we'll determine that. What we can do here is decide that another object will play the role of getting us our expert.

The expert selection isn't a part of what we're doing yet, so we set our code to depend on an object we deem responsible for this task. It's often a mistake to solve too many problems at once. This Context that we build keeps our focus on the interaction between a user, an expert, and the question.

For this use case, where we are modeling the interaction among the user, expert, and question, our selection of the expert is a distraction. This is a point where further thought and decisions must be made that are not immediately relevant to the task at hand. It is appropriate, therefore, to write our code in a way that will give us flexibility to easily make changes later.

Recognizing decision points in our code can help us to realize a need for abstraction. If we were to represent the knowledge for selecting an expert in this `ExpertQuestioning` class we would not only be violating SRP but the name of our class would no longer reflect its responsibilities. Our main concern is the asking and answering of a question, so focusing on expert selection will be abstracted away to deal with the decision later.

Let's leave a note for ourselves and others that this is a point for further consideration.

```
def expert
  # TODO: Expert selection requires consideration
  Expert.first_available
end
```

This approach is often a first step in leaving junk comments in a project. Developers leave notes for themselves or others never to return to them. The best way to avoid this junk is to never leave it in the first place. We will be coming back to this shortly but Rails and many text editors and IDEs provide a simple way to list comments like this.

```
rake notes
app/models/expert_questioning.rb:
  * [ 41] [TODO] Expert selection requires consideration
```

Leave comments if you must, but stay on top of them with something that will report these trouble spots.

[ more coming soon ]

# Responding to Rails

The power of a well-designed framework can simplify so many aspects of your program. Ruby on Rails' popularity grew in a large part because it highlighted the value of *convention over configuration*. It's far easier to just do what you're supposed to do rather than configure everything before you do what you're supposed to do. So how do we fit ideas like DCI into a framework like Rails?

Answering that depends a bit upon your perspective. Let's first take a look at where we started in our application. In a previous chapter, we had a simple controller hooking into our business logic. It looked like this:

```
class TransfersController < ApplicationController
  respond_to :html
  def create
    transfer = MoneyTransfer.new(params[:account_id],
                                 params[:destination_id])
    account = transfer.execute(params[:amount])
    respond_with account do |format|
      format.html { redirect_to account_transfers_path(account) }
    end
  end
end
```

While this block of code holds less complexity than what we had originally, it's still not as simple as we can get with the options that Rails provides.

In Crafting Rails Applications[32], José Valim walks us through creating Responders to DRY our controllers up. By using Responders, we can offload the behavior of redirection, rendering and other controller responsibilities.

We want to make our controller code even simpler if we can. If we decide later to add responses for XML or JSON or some other type then we'll quickly turn this small controller action into a mental speed bump. We'll need to slow down to understand what's happening or slow down just to find the part we care about. In both code execution and code comprehension we aim for speed.

When the flow of our code is typical, it's OK for the details to be hidden somewhere else. Burying conventional code is a feature. But when we need to understand special decisions or other elements of our program we want them as close to the surface as possible. We would do well to consider what a new level of abstraction gives and what it takes away.

With conventional expectations, we'd rather have our controller be as simple as this:

[32] http://pragprog.com/book/jvrails/crafting-rails-applications

```
class TransfersController < ApplicationController
  respond_to :html
  def create
    transfer = MoneyTransfer.new(params[:account_id],
                                 params[:destination_id])
    respond_with transfer.execute(params[:amount])
  end
end
```

The way Rails conventionally works is that after a create action a user is sent to the representative view of the thing just created. Here we would expect that a user would end up at a view of the transfer. But what happens when we run this code?

It all depends on the return value of the `execute` method.

If we alter that method to return our `account` object we'll get a default responder for the `account`. If it returns a `transfer` object, we'll get a `transfer` responder. This raises the question of where this responsibility belongs. Should the responder be a part of the Context we've created?

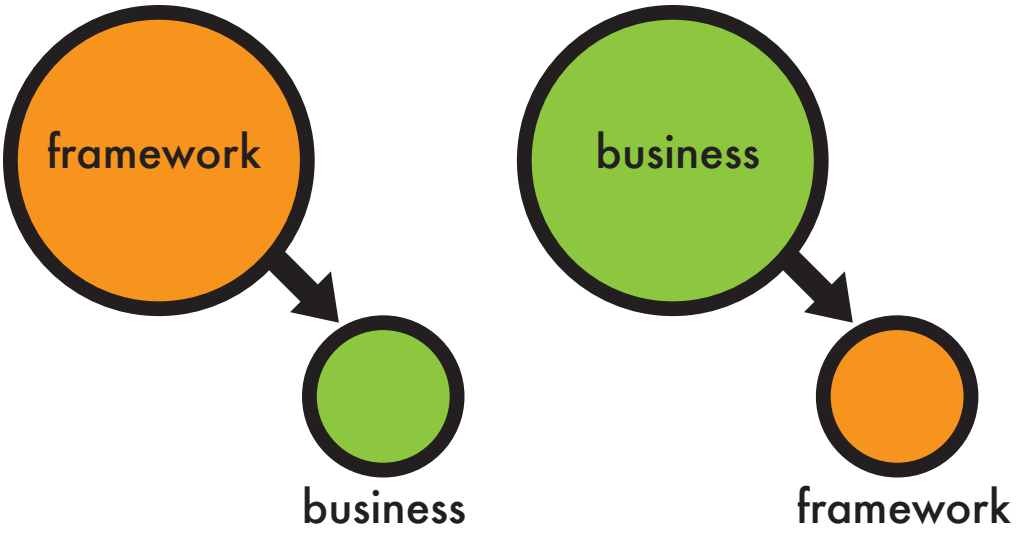
## Considering Dependencies

Let's stop and think about dependencies.

While we are discussing responding to Rails, we're evaluating the place of Rails in our application. What does a framework do for us and where does it belong?

Do we stand up a framework to make use of our domain? Or do we build software for our domain around a framework?

The answer comes down to coupling. As long as we have loosely coupled system, where parts of our system share little to no knowledge of each other, we can better arrange the parts as we see fit. If our domain logic requires our framework, then a change in our framework may affect the way our domain objects behave. If our framework requires our domain objects, then the objects of concern retain the behavior we desire.




Diagrams can be both helpful or confusing, so let's look at what this diagram is illustrating along with some code.

We're using arrows to show the direction of dependency between objects. The arrow points to an object which must exist and must be known for the object at the beginning of the arrow to function properly. In code representing the first portion of our diagram, the framework depending on the business logic might look like this:

```
class SomeController < ActionController::Base
  def action
    if BusinessClass.logic_method
      redirect_to :some_path
    else
      render :some_other_action
    end
  end
end
```

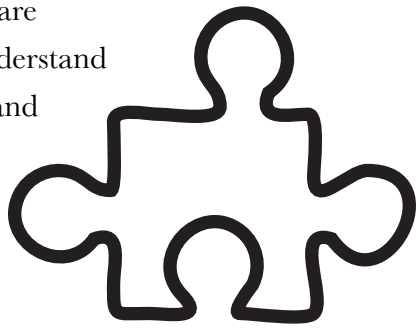
Rails is a framework that takes care of the parts of our business that we don't want to address. Handling web requests, creating HTTP headers, and other aspects of our platform are a solved problem. So we choose something like Rails and place our trust in it's mature and well-tested code to remove that concern for us. We put the decisions about redirection or rendering in the Rails-based part of our application.

When our needs are simple, this approach is quick and easy to get an application going. But when we want to understand what process `BusinessClass` and `logic_method` represent, we have knowledge of the behavior in **both** `BusinessClass`

and `SomeController`. In doing so, we force ourselves to mentally combine these parts to get a complete understanding of the procedure.

Our controller acts as the connection to the framework and now has knowledge not only of `BusinessClass` and `logic_method`. It knows that the method can be evaluated to be `true` or `false` and that there are *two separate behaviors* depending on the return value. That's five aspects of our business process that are tied to a procedure in this controller.

If this procedure in it's entirety is important to our business logic, why would we split knowledge among these objects?



Our objects begin to look a little different when we share knowledge among them like this. When we attempt to understand our program, we need to keep more things in our minds and piece them together by matching up methods and behaviors. Rather than having objects that manage complete responsibilities, we have objects that look more like puzzle pieces which only reveal their process when examined together.

The fact that Rails handles the web for us isn't a good argument for keeping decisions in the controller. The *decision* made by `if` and `else` is relevant to our business process, but the *execution* of it is relevant to our framework. So what can we do?

We can simplify our program and ease our understanding by writing a controller like the following:

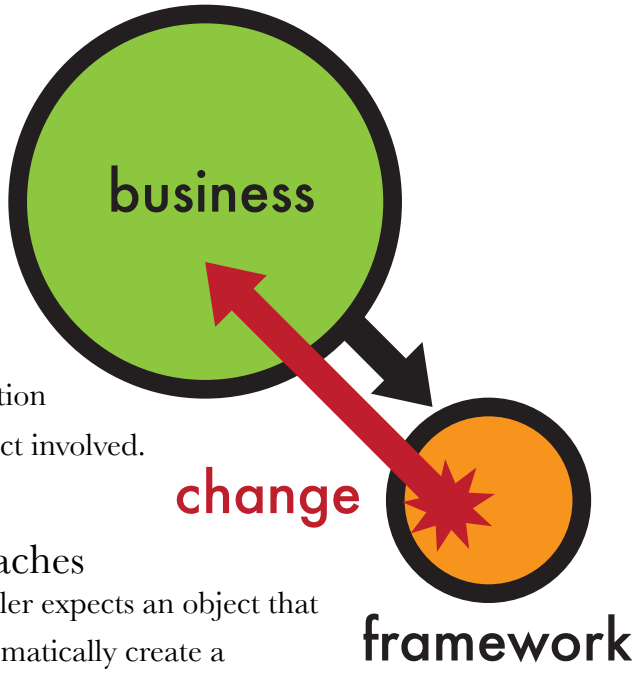
```
class SomeController < ActionController::Base
  def action
    respond_with BusinessClass.logic_method
  end
end
```

When looking at this action you might be left wondering "what happens with success or failure?" That question forces you to look into the `BusinessClass` for the answer. This is intentional.

Now the controller only needs to know two things. It knows about `BusinessClass` and it knows about `logic_method`. The decision of what to do based upon the outcome of that method is left up to the business logic in our application. The execution is left to our controller in `respond_with`.

Often dependency diagrams leave me scratching my head because they don't quite paint the clearest picture. There's something we've left out of our illustration of the relationship between framework and business models.

With a simple change to show the backward relationship we can see the effect of changes made to parts of the system.




Regardless of which part of our system is changing, we piece it together with relationships. When we manage dependencies and try to limit the disparate spread of responsibility, we are managing relationships. It helps us not to think about individual responsibilities alone, but how the execution of those responsibilities affects each object involved.

## Alternate Dependency Approaches

By using `respond_with`, our controller expects an object that behaves like a Responder. Rails will automatically create a Responder for us and follow the conventions of going to an appropriate location or perhaps re-render a view with errors.

If the decision made by a Responder is important to our business process, then we may decide that we want to see that decision in plain view. Relying on Rails to create a default responder for us might be hiding important information. Once again we'd need to dig through documentation or framework code to put together several pieces of our puzzle before we had a complete understanding.

One option to handle the behavior is to evaluate the expectations of our framework and agree to the contract that it expects. Rails Responders have several useful features for managing our application flow and take care of everything we'd typically expect. All we need is an object that responds to `call`. This is a simple contract to follow and can be handled easily with a lambda.

In this case our framework, Rails, requires that we either set the responder in arguments:

```
class SomeController < ActionController::Base
  def action
    respond_with BusinessClass.logic_method,
                 :responder => lambda{ ... }
  end
end
```

Or we may set the responder in the controller:

```
class SomeController < ActionController::Base
  self.responder = lambda{ ... }

  def action
    respond_with BusinessClass.logic_method
  end
end
```

Using this feature of Rails makes sense when we're changing conventions. We could have our `BusinessClass` implement a `responder` method and return an object that behaves the way Rails expects. That would allow us to keep the details of a Responder in our `BusinessClass.`

```
class BusinessClass
  class BusinessResponder
    # ...
  end
end
```

This might help keep the concerns of our use case in one place. By doing this, however, we either give up any useful behavior from our framework, or we set our `BusinessResponder` to inherit from `ActionController::Responder` creating an explicit dependency on classes provided by our framework.

Changes to the framework might render our code useless (at least until we spend the time to clean up all our methods). Additionally any decision to make changes to the implementation of our business logic would carry the extra weight of how those decisions could affect our use of the framework.

Instead of implementing the interface that our framework expects, we can tell the framework related parts what to do. It's easy to invert the control over what should

happen with Dependency Injection[33] where we pass an object to our business model that will carry out a task for handling part of our procedure.

```
class TransfersController < ApplicationController
  def create
    transfer = MoneyTransfer.new(params[:account_id],
                                 params[:destination_id])
    transfer.command(self)
    transfer.execute(params[:amount])
  end
end
```

Here we've introduced a `command` method that we haven't discussed before. Our context can use this method to set the object that takes commands. When we set the responsible object with this method, we can alter our transfer procedure to take next steps.

By using Dependency Injection we can push the decision about success and failure into the business logic. Because the object we send is external to our context, we keep the implementation of the resultant behavior external as well.

This method follows no particular naming convention. We are telling the `transfer` object to `command` the current controller what to do. This method could just as well be named `set_listener`, `guide`, or any other term that best reveals the intentions for you and your development team. The result is that the object provided will be told what to do when the business rules execute.

[33] http://en.wikipedia.org/wiki/Dependency_injection

```
class MoneyTransfer
   def execute(amount)
      @source.transfer_to(@destination, amount.to_i,
         failure: ->{ controller.display_error @source },
         success: ->{ controller.go_to @source })
      @source
   end

   def command(controller)
      @controller = controller
   end

   private
   attr_reader :controller

   module Transferrer
      def transfer_to(destination, amount, callbacks={})
         transaction do
           self.balance -= amount
           destination.balance += amount
           callbacks[:success].call
         rescue
           callbacks[:failure].call
         end
      end
   end
end
```

Now at the start of this transaction we clearly see what the behavior will be for failure and success. We've put together all the decisions regarding what should happen and we've kept out the implementation of how it should happen.

We haven't covered the tests for these changes yet. In all likelihood there are some things we've forgotten. We've even introduced a `transaction` block and a non-specific `rescue` to handle the failure. We'll look more closely at these things again, but what we can see is how these approaches can keep the concepts of our program together.

Sometimes we want to have Rails do the conventional thing. As our programs grow, however, we often find that we care more about this user flow than we first realized. Starting with the design of the user flow helps us keep a better eye on what is and isn't happening in the application. Keeping these decisions in code tied to objects that represent our business logic makes understanding the use case much easier; ease of communication with stakeholders follows.

## Framework Glue

To tie our controller to the behavior we've created here, we'll need to adjust some things. In the `execute` method we tell the `controller` to `go_to` a location. Rails doesn't use this method so we'll need to implement it.

But first, why would we do this? If Rails already has methods for our controller that allow us to `redirect_to` another location or `render` a view, why bother creating some go-between?

Well, there are two reasons. The first is that we should use language that best represents our domain. The various members of our team will better understand what the application is doing when we use terms that aren't specific to our framework. The second reason is that using a different method name helps illustrate the first point. So maybe that's just one reason.

If we blindly rely on our framework, however, we'll allow the implementation details to creep into our domain modeling. If we have a `controller` object from Rails, we might rely too much on it's behavior and find that we have tightly coupled the behavior of our system to the implementation of our framework.

Let's look at a simple example:

```
class BusinessClass
  def logic_method(controller)
    if true
      controller.redirect_to controller.some_special_path
    else
      controller.render :some_controller_action
    end
  end
end
```

This pseudo-code represents what we might find ourselves doing if we heavily rely on the `controller` object that we send. The `controller` is responsible for redirecting, providing a path for a location, and rendering another view marked by some symbol.

It's not so bad that we expect the controller to do all these things, but we expect it to do these things in a very specific way. Our `BusinessClass` maintains knowledge of not only it's own behavior, but that of the external object as well.

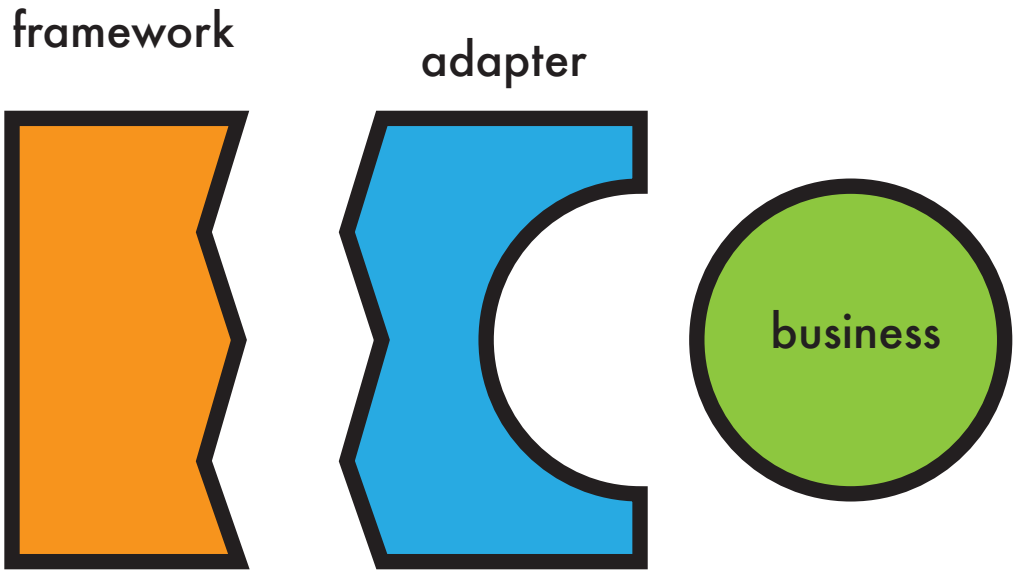
Certainly we can clean this code up with something like `Forwardable`.

```
require 'forwardable'
class BusinessClass
  extend Forwardable
  def_delegator :@controller, :redirect_to, :some_special_path,
                              :render
  def controller=(object)
    @controller = object
  end

  def logic_method
    if true
      redirect_to some_special_path
    else
      render :some_controller_action
    end
  end
end
```

This, however, only moves things around; it doesn't change the dependency. The class still maintains all the details of the `controller` object.

We've been looking at dependencies but the illustrations haven't been entirely accurate. Our framework is the classes that Rails provides. The controllers in a Rails application act as the glue between our framework and our business logic. In both the `SomeController` and `TransfersController` examples we've discussed the objects function as a proxy between the parts we care about and the parts we don't.




The framework provides behavior for us to handle the concerns of web requests and we may compose or inherit from those objects to adjust to our own needs. But these new objects that we create from the libraries we use in our framework work well as adapters and configuration. It's unlikely that our business logic specifically requires that we address HTTP requests. What's more likely is that the web and Rails provide an excellent delivery mechanism for our business logic.

With all that in mind, it makes communication clearer to focus on the domain first. Since we can view our Rails-based controllers as adapters between our framework and domain, we should adjust to our domain there as well.

```
class TransfersController < ApplicationController
  def create
    # ...
  end

  private
  def go_to(resource_or_action)
    case resource_or_action
    when Symbol then
      render resource_or_action
    else
      redirect_to resource_or_action
    end
  end
end
```

Adding this bit of behavior to our framework glue allows our business logic to use terms that are relevant to our domain. We define those terms as methods and adapt them to our framework. With this approach we are more free to make changes to our framework up to and including changing it completely.

You may determine that this isn't necessary. Rails uses terminology and conventions that are very clear, but blindly accepting a dependency on the behaviors that a framework provides may lead to complicated responsibilities later.

# East-oriented Code

[ coming soon ]

# Understanding Context

[ coming soon ]

# Inheritance, Forwarding and Delegation

Important, in the design of OO systems, is the composition of your objects. The way the system is structured to perform actions and share behavior will affect the way you use it and understand it. Subsequently the structure will also affect the way bugs are introduced, so knowing the ins and outs of different techniques as well as what Ruby can and can't do for you is important.

## Simplicity of Inheritance

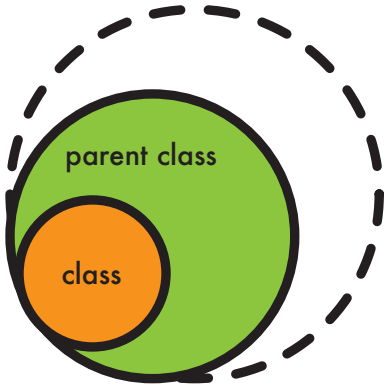
Inheritance is simple to implement and understand. It's easy to think about inheritance as nested categories. A category has sub-categories and can have a parent category. Every object has the same attributes and behaviors of its parent.

Inheriting features from a parent class is often how we think about our world:

- A Person can walk, talk, eat and many other things.
- A Doctor is a Person, but also understands anatomy and physiology of the human body.

• A Cardiologist is a Doctor who has special abilities and understanding of heart anatomy and physiology.

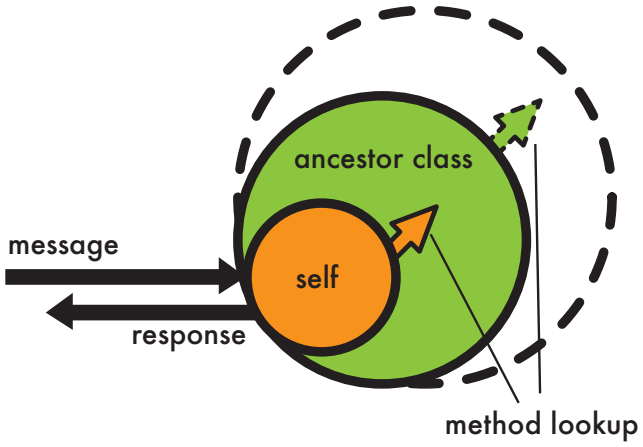
In these three simple examples, we see exactly how easy it is to begin thinking about hierarchy. But if we were to make a program assisting cardiologists, would we first spend our time describing a person? Probably not.




We often think about our world with inheritance in mind. So, when we turn to write our software, we expect the same. When defining what things in our system *are*, inheritance helps us to form a mental model of our categorization of them.

With class inheritance, however, our objects get locked into the implementation defined in their class, or the class' superclass, or even further up the inheritance tree. That doesn't mean that we can't alter the behavior where we need but splitting implementation details among multiple classes may split our focus for what is decidedly related behavior.

In a typical Class-oriented program, an object must find its behavior from its parent or any of its ancestor classes. This approach guides programmers on the path of thinking about broad and static categorization of objects. If an object's behavior is determined by the methods found in its class ancestry, then the behavior must be put in the ancestry... or so we have come to think.



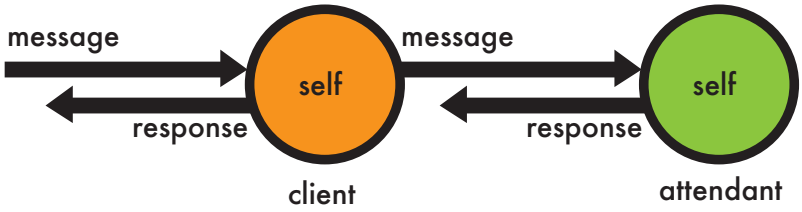

Our systems, however, are a complex network of interacting objects where behavior and responsibilities can be implemented by interaction between objects. Our vision of the system is objects sending messages, so more objects handling more specific aspects fits well with our mental model.

## Composition and Forwarding

Rather than lock down the behavior sharing to a single path, we can share code among objects with composition. We can make more objects which handle behavior individually.

Typically this means that we build our objects to call methods or forward certain messages to additional objects which contain more or better information about handling a message. We can refer to the message receiver as the **client** because it will ask another object to handle the behavior much like a client application would send requests to a server.

The other object is often referred to as the **delegate**, but for now let's use the term **attendant**. The attendant will tend to the needs of the client.




The client and attendant are entirely separate objects each with their own identity and state which affects the way the message is handled. The client responds to a message not by performing an algorithm itself, but by passing along the response from an attendant object which handled the message. The attendant might perform an algorithm or it may also forward the message on to its own attendant.

This object collaboration is implemented with one object maintaining an internal reference to another object to forward complete responsibility for responding to a message. Let's look at a simple example of message forwarding with the Forwardable library from Ruby's standard library.

```
require 'forwardable'
class Calculator
  extend Forwardable
  def initialize(account)
    @account = account
  end
  def_delegators :@account, :balance, :balance_on_date
  # or
  delegate [:balance, :balance_on_date] => :@account

  def change_from_date(date)
    balance - balance_on_date(date)
  end
end
account = Account.find(1)
calculator = Calculator.new(account)
```

Here, a `Calculator` will be initialized with an `account` object. When a `calculator` receives the messages `:balance` or `:balance_on_date`, they will be forwarded to its `@account` reference.

The `calculator` acts as a **proxy** for additional behavior and forwards messages to our attendant. The implementation of handling the behavior is managed internally to the object. An attendant object may ultimately handle the response, but to any programs relying on an account object, the interface remains on the account object.

Also in the standard library, we have Delegate which allows the same message forwarding semantics as Forwardable. Let's look at a short example where

```
require 'delegate'
class Calculator < SimpleDelegator
  def change_from_date(date)
    balance - balance_on_date(date)
  end
end
account = Account.find(1)
calculator = Calculator.new(account)
```

Rails has its own way of handling what it calls delegation as well.

```
class Calculator
  delegate :balance, :balance_on_date, :to => :@account
  def initialize(account)
    @account = account
  end
  def change_from_date(date)
    balance - balance_on_date(date)
  end
end
account = Account.find(1)
calculator = Calculator.new(account)
```

The Rails provided option also gives us a way to prefix our messages before they are forwarded. The Rails `delegate` method takes a `:prefix` argument which allows us to alter our method names with a default or specified prefix, turning `balance` to `account_balance`. This helps us to choose names that will give us clues about their behavior as well as give some way to differentiate method calls such as `checking_account_balance` and `savings_account_balance` when forwarding messages to objects with similar interfaces.

There are many ways we can choose to use these features to organize our objects' behavior. The Delegate library also offers a `DelegateClass()` constructor method to control forwarding of messages for class methods. We can use `SingleForwardable` from the Forwardable library for an alternate approach. Beyond these classes and modules from the standard library we can also rely on `method_missing` to manage our own implementation of forwarding messages. Often the one we select, however, has much to do with personal preference.

These different implementations have a few things in common:

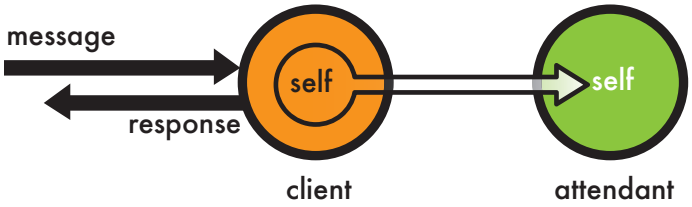
1. They define an association between objects.
2. They forward unaffected messages leaving the method names intact (with an exception from Rails).
3. They **do not** implement delegation.

## Consultation is Not Delegation

What is often labelled delegation is anything but. Message recipients which forward messages to an attendant object, often called the delegate, merely act as a proxy for the attendant's implementation of the method call.

Henry Lieberman coined the term **delegation** in *Using prototypical objects to implement shared behavior in object-oriented systems* released with the OOPLSA '86 Conference proceedings on Object-oriented programming systems, languages and applications.[34] Specific to his definition of delegation was the use of prototypes and the dynamic binding of `self` to the original message recipient.

Forwarding and delegation are related but not the same, despite the confusing names used in the Ruby standard library. When the attendant object maintains its own notion of `self`, this is only a forwarded message: a method call. When a message is forwarded from one object to another, it is *delegation* when `self` refers to the original message recipient.




While Ruby does provide us with a Delegate and a Forwardable library, they allow implementations of the same pattern: proxy. Günter Kniesel, in his dissertation *Dynamic Object-Based Inheritance with Subtyping* [35], coined the term **consultation** to refer to this type of message forwarding without dynamic binding of `self`.

In discussion, Lieberman pointed out to me that the addition of a new term is, in his opinion, unnecessary: forwarding messages to another object is merely a method call. I agree, yet there's definitely something here given the widely held confusion. The Ruby standard library calls this delegation, but delegation it is not.

Consultation is a term we can use to properly describe what we are implementing with the Delegate and Forwardable libraries. There are two characteristics that are present in all of these examples that we've discussed.

---

[34] http://dl.acm.org/citation.cfm?id=960112.28718

[35] More information at http://javalab.cs.uni-bonn.de/research/darwin/index.html

The first is the connascence of method names between collaborating objects. Connascence refers to the point and type of coupling between objects where a change in one object requires a change in the other[36]. Consultation occurs when the message recipient forwards a message to another object which handles the response to the same message. A `calculator` would receive a `balance` message and forward that message to an `account` which responds to `balance`. Rails' implementation of `delegate` does include a caveat that the message might be prefixed before it is forwarded, but the implementation is still connascent by method name.

The second aspect of consultation is that the message recipient makes no modifications to the algorithm for handling the response. A message of `balance` would be forwarded, unaffected, to the attendant object.

```
delegate :balance :to => :@account
```

Object collaboration breaks from consultation when the message recipient makes alterations to the arguments or the response.

```
def balance
  @account.balance.to_s
end
```

In the end, however, consultation is just a method call.

## Diving Into Methods

As we have seen, the core features Ruby gives us merely allow us to follow patterns of object collaboration and consultation. Ruby does have a way to delegate methods but provides no formal assistance in the standard library. Extending an object with a module allows us to preserve the `self` reference inside the method block, but extending an object is **object extension**, not delegation. So what can we do for delegation?

First let's take a look at methods by initializing an object to see what we can do:

---

[36] See more information about examples of connascence in Ruby at http://blog.rubybestpractices.com/posts/gregory/056-issue-24-connascence.html

```
class Account
  def balance
    100
  end
end
account = Account.new
```

With our account initialized we can use a method called `method` that takes an argument of a method name and returns an instance of a `Method`.

```
balance_method = account.method(:balance)
=> #<Method: Account#balance>
balance_method.methods.sort
=> ["==", "===", "=~", "[]", "__id__", "__send__", "arity", ...]
```

We are able to see information about the method and even execute it without explicitly involving the `account` instance:

```
balance_method.call
=> 100
```

The method can also tell us about its receiver, the object to which it is currently bound:

```
balance_method.receiver
=> #<Account:0x10e5823e8>
balance_method.receiver == account
=> true
balance_method.owner
=> Account
```

## Sharing Behavior With Methods

There is seemingly little use of Ruby providing this feature. As an individual instance of a `Method` what more can we do with it than view some information and call it? We can bind it to other objects.

```
unbound_balance_method = balance_method.unbind
=> #<UnboundMethod: Account#balance>
```

`UnboundMethod`s are `Method`s that are not bound to an object. In other words, there is no receiver for an `UnboundMethod`. And these methods may not be called before first being tied to an object for the execution.

```
unbound_balance_method.receiver

NoMethodError: undefined method `receiver' for #<UnboundMethod:
Account#balance

unbound_balance_method.call

NoMethodError: undefined method `call' for #<UnboundMethod:
Account#balance
```

We can bind a method to another object of the same type as the original receiver and then call it.

```
other_account = Account.new

=> #<Account:0x10e4355f8>

unbound_balance_method.bind(other_account).call

=> 100
```

We are restricted, however, from using unbound methods on objects that are not of the same type as the object where the method was defined.

```
class Other

end

other = Other.new

=> #<Other:0x10e36a6c8>

unbound_balance_method.bind(other).call

TypeError: bind argument must be an instance of Account
```

## Delegating Methods

With a clearer understanding of delegation and how Ruby handles methods, we can open our code to new ways of sharing behavior. We can finally *delegate* responsibility rather than merely *consult*.

We can create objects with the behaviors we need to act as the delegates for behavior. Let's look at a simple example of what this means.

```
source = Account.find(1)
action_account = Account.new
def action_account.transfer_to(amount, other)
  self.decrement(amount)
  other.increment(amount)
end

destination = Account.find(2)
transfer_method = action_account.method(:transfer_to).unbind
transfer_method.bind(source).call(200, destination)
```

This small sample shows us that we can define methods on one object and delegate to it the responsibility of managing the behavior. This dynamic method binding preserves the `self` reference without extending the target object's behavior. We can use this technique of delegating responsibility to overcome problems that may arise when extending object behavior or when using multiple objects to represent data and behavior.

This example doesn't use prototypical inheritance, but we're a bit limited in how Ruby can handle that. We'll dive deeper later and we'll explore using these approaches in the next chapters.

[ more coming soon ]