In this chapter, you'll see:

- Incremental development
- Use cases, page flow, and data
- Priorities

CHAPTER 5

The Depot Application

We could mess around all day hacking together simple test applications, but that won't help us pay the bills. So let's sink our teeth into something meatier. Let's create a web-based shopping cart application called Depot.

Does the world need another shopping cart application? Nope, but that hasn't stopped hundreds of developers from writing one. Why should we be different?

More seriously, it turns out that our shopping cart will illustrate many of the features of Rails development. You'll see how to create maintenance pages, link database tables, handle sessions, create forms, and wrangle modern JavaScript. Over the next twelve chapters, we'll also touch on peripheral topics such as unit and system testing, security, and page layout.

Incremental Development

We'll be developing this application incrementally. We won't attempt to specify everything before we start coding. Instead, we'll work out enough of a specification to let us start and then immediately create some functionality. We'll try ideas, gather feedback, and continue with another cycle of mini design and development.

This style of coding isn't always applicable. It requires close cooperation with the application's users because we want to gather feedback as we go along. We might make mistakes, or the client might ask for one thing at first and later want something different. It doesn't matter what the reason is. The earlier we discover we've made a mistake, the less expensive it'll be to fix that mistake. All in all, with this style of development, there's a lot of change as we go along.

Because of this, we need to use a toolset that doesn't penalize us for changing our minds. If we decide we need to add a new column to a database table or change the navigation among pages, we need to be able to get in there and do it without a bunch of coding or configuration hassle. As you'll see, Ruby on Rails shines when it comes to dealing with change. It's an ideal agile programming environment.

Along the way, we'll be building and maintaining a corpus of tests. These tests will ensure that the application is always doing what we intend to do. Not only does Rails enable the creation of such tests but it even provides you with an initial set of tests each time you define a new controller.

On with the application.

What Depot Does

Let's start by jotting down an outline specification for the Depot application. We'll look at the high-level use cases and sketch out the flow through the web pages. We'll also try working out what data the application needs (acknowledging that our initial guesses will likely be wrong).

Use Cases

A *use case* is simply a statement about how some entity uses a system. Consultants invent these kinds of phrases to label things we've known all along. (It's a perversion of business life that fancy words always cost more than plain ones, even though the plain ones are more valuable.)

Depot's use cases are simple (some would say tragically so). We start off by identifying two different roles or actors: the *buyer* and the *seller*.

The buyer uses Depot to browse the products we have to sell, select some to purchase, and supply the information needed to create an order.

The seller uses Depot to maintain a list of products to sell, to determine the orders that are awaiting shipment, and to mark orders as shipped. (The seller also uses Depot to make scads of money and retire to a tropical island, but that's the subject of another book.)

For now, that's all the detail we need. We *could* go into excruciating detail about what it means to maintain products and what constitutes an order ready to ship, but why bother? If some details aren't obvious, we'll discover them soon enough as we reveal successive iterations of our work to the customer.

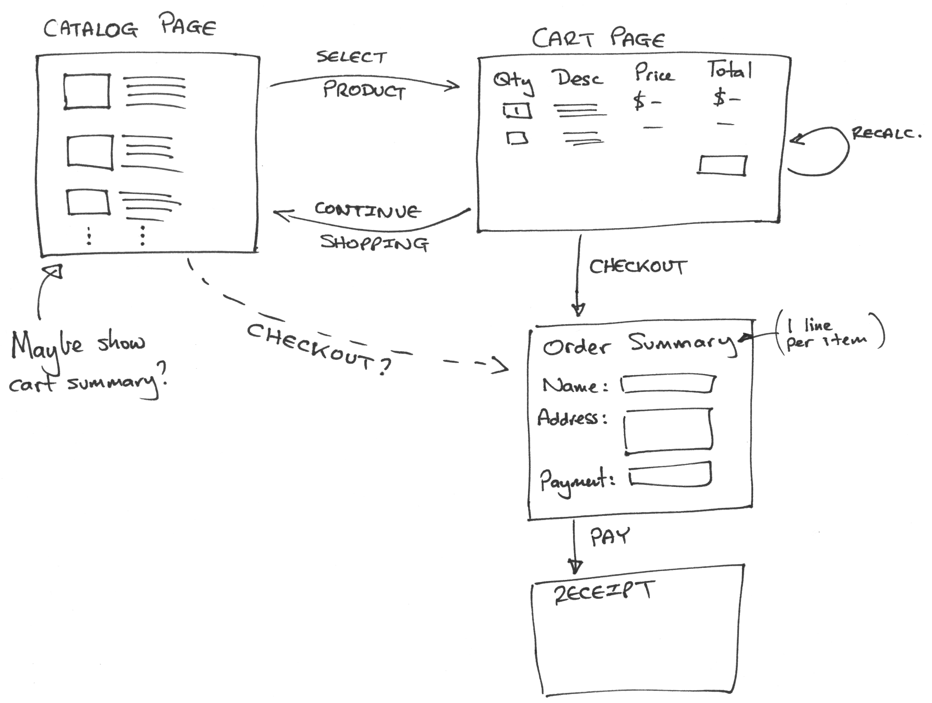
Speaking of getting feedback, let's get some right now. Let's make sure our initial (admittedly sketchy) use cases are on the mark by asking our users. Assuming the use cases pass muster, let's work out how the application will work from the perspectives of its various users.

Page Flow

We always like to have an idea of the main pages in our applications and to understand roughly how users navigate among them. This early in the development, these page flows are likely to be incomplete, but they still help us focus on what needs doing and know how actions are sequenced.

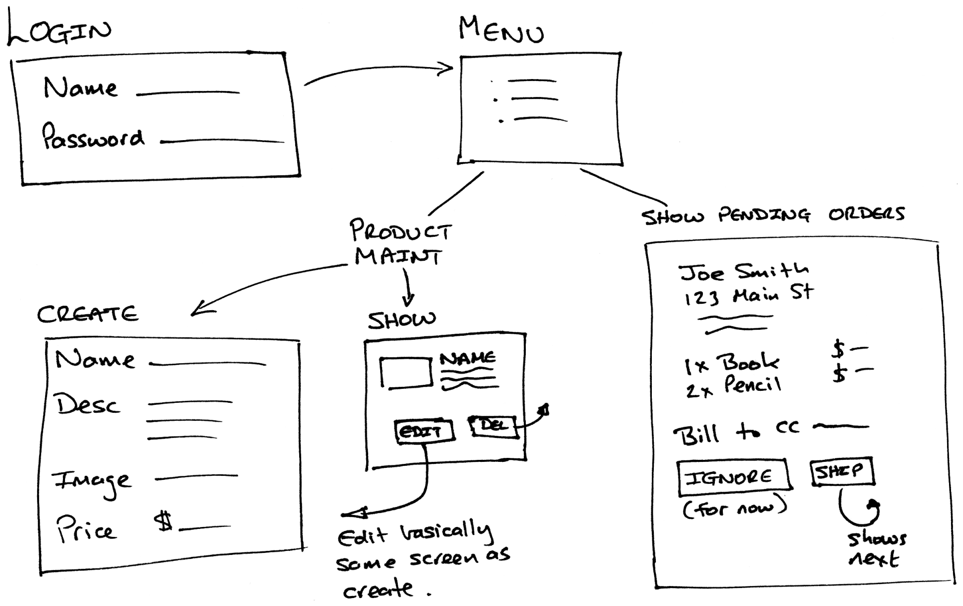
Some folks like to use Photoshop, Word, or (shudder) HTML to mock up web application page flows. We like using a pencil and paper. It's quicker, and the customer gets to play too, grabbing the pencil and scribbling alterations right on the paper.

The first sketch of the buyer flow is shown in the following figure.



It's pretty traditional. The buyer sees a catalog page, from which he selects one product at a time. Each product selected gets added to the cart, and the cart is displayed after each selection. The buyer can continue shopping using the catalog pages or check out and buy the contents of the cart. During checkout, we capture contact and payment details and then display a receipt page. We don't yet know how we're going to handle payment, so those details are fairly vague in the flow.

The seller flow, shown in the next figure, is also fairly basic. After logging in, the seller sees a menu letting her create or view a product or ship existing orders. When viewing a product, the seller can optionally edit the product information or delete the product entirely.



The shipping option is simplistic. It displays each order that hasn't yet been shipped, one order per page. The seller can choose to skip to the next or can ship the order, using the information from the page as appropriate.

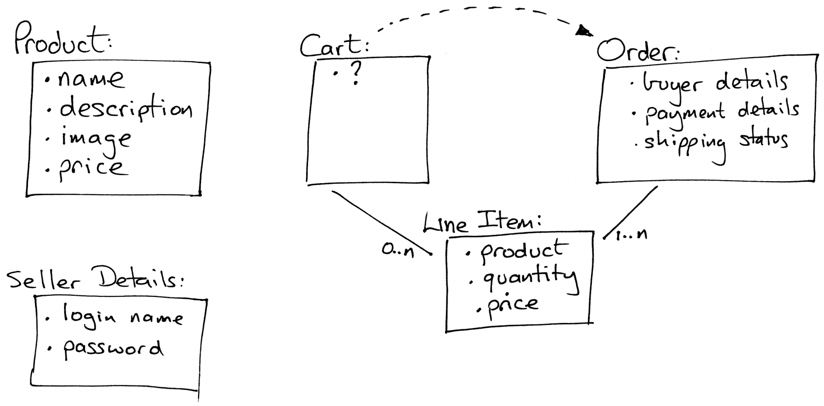
The shipping function is clearly not going to survive long in the real world, but shipping is also one of those areas where reality is often stranger than you might think. Overspecify it up front, and we're likely to get it wrong. For now, let's leave it as it is, confident that we can change it as the user gains experience using our application.

Data

Finally, we need to think about the data we're going to be working with.

Notice that we're not using words such as *schema* or *classes* here. We're also not talking about databases, tables, keys, and the like. We're talking about data. At this stage in the development, we don't know if we'll even be using a database.

Based on the use cases and the flows, it seems likely that we'll be working with the data shown in the figure on page 67. Again, using pencil and paper seems a whole lot easier than some fancy tool, but use whatever works for you.



Working on the data diagram raised a couple of questions. As the user buys items, we'll need somewhere to keep the list of products they bought, so we added a cart. But apart from its use as a transient place to keep this product list, the cart seems to be something of a ghost—we couldn't find anything meaningful to store in it. To reflect this uncertainty, we put a question mark inside the cart's box in the diagram. We're assuming this uncertainty will get resolved as we implement Depot.

Coming up with the high-level data also raised the question of what information should go into an order. Again, we chose to leave this fairly open for now. We'll refine this further as we start showing our early iterations to the customer.

General Recovery Advice

Everything in this book has been tested. If you follow along with this scenario precisely, using the recommended version of Rails and SQLite 3 on Linux, MacOS, or Windows, everything should work as described. However, deviations from this path can occur. Typos happen to the best of us, and not only are side explorations possible, but they're positively encouraged. Be aware that this might lead you to strange places. Don't be afraid: specific recovery actions for common problems appear in the specific sections where such problems often occur. A few additional general suggestions are included here.

You should only ever need to restart the server in the few places where doing so is noted in the book. But if you ever get truly stumped, restarting the server might be worth trying.

A "magic" command worth knowing, explained in detail in Part III, is `bin/rails db:migrate:redo`. It'll undo and reapply the last migration.

If your server won't accept some input on a form, refresh the form on your browser and resubmit it.

Finally, you might have noticed that we've duplicated the product's price in the line item data. Here we're breaking the "initially, keep it simple" rule slightly, but it's a transgression based on experience. If the price of a product changes, that price change shouldn't be reflected in the line item price of currently open orders, so each line item needs to reflect the price of the product at the time the order was made.

Again, at this point we'll double-check with the customer that we're still on the right track. (The customer was most likely sitting in the room with us while we drew these three diagrams.)

Let's Code

So after sitting down with the customer and doing some preliminary analysis, we're ready to start using a computer for development! We'll be working from our original three diagrams, but the chances are pretty good that we'll be throwing them away fairly quickly—they'll become outdated as we gather feedback. Interestingly, that's why we didn't spend too long on them; it's easier to throw something away if you didn't spend a long time creating it.

In the chapters that follow, we'll start developing the application based on our current understanding. However, before we turn that page, we have to answer one more question: what should we do first?

We like to work with the customer so we can jointly agree on priorities. In this case, we'd point out to her that it's hard to develop anything else until we have some basic products defined in the system, so we suggest spending a couple of hours getting the initial version of the product maintenance functionality up and running. And, of course, the client would agree.