3
How to Solve Big Problems

THIS CHAPTER EXAMINES HOW TO BETTER solve programming problems. The aim is to get tasks to a level where we can have some functionality accomplished at the end of every day. Then we do not have to be so concerned with the bigger-picture stuff while we solve today's problems.

This chapter, like Chapter 1, is not specific to the .NET Framework. You can use these ideas in other development environments. We will be using the lessons learned from this chapter later on in this book. Be sure that you understand how to break down tasks and make time estimates before you jump ahead.

When mentoring development teams, I often encounter developers who do not (or cannot) break down a task into really small pieces of work that are achievable in a matter of a few hours or minutes. This is a skill that you need to learn. This chapter examines why it is important to have these small tasks, and shows how to break down some complex problems into simple easy-to-solve units of work. This chapter discusses XP practices that help us accomplish this task breakdown.

The Software Development Problem

When developing a piece of software, you are often dealing with many issues at the same time, some of which might detract your focus from the particular problem you are trying to solve. Issues you are dealing with will include the following:

- Adhering to the design of the system
- Making sure your new code doesn't break any existing functionality
- Ensuring you are following the coding conventions
- Worrying how this solution will impact future tasks that need to be completed

Wouldn't it be good if you could forget about those other issues? Then you could just code a solution today to the problem you have at hand. Problems are hardly ever so small and focused that you can actually accomplish them today, or even this week.

That is what we are going to attempt to achieve. First, we will explore what makes a good solution. Second, we will work through an exercise that demonstrates how we can carry out focused small tasks that lead us towards the completed solution.

The Genius Is in the Simplicity

A genius is a person who can take a complex problem and find a very simple solution that solves it. This solution is usually fairly trivial to implement *after* you know what it is. Most of us take the easy option, however, and create complex solutions to solve our complex problems. We do this for a number of reasons; the two main ones are as follows:

- We take on too much at one time.
- We look at how a similar complex problem has been solved before and try to copy that solution.

Our egos don't help in this matter. We want to show off how smart we are at implementing some complicated solution. We want to justify that we should be paid more and promoted. We don't feel comfortable going to our boss and saying, "That problem was solved in five lines of code, but it took me the whole of the past three days to write those five lines."

Big, Complex Solutions

The trouble with designing these big, complex solutions is that we often then get stuck when trying to actually implement them. They are just too hard and have too

many points of failure. These solutions end up becoming the bane of our existence. We have to maintain them and fix bugs in them. Sure, we can be good citizens and use design patterns and well-documented architectures. Ultimately, however, we end up, time and time again, juggling big issues while trying to get a new piece of functionality into the system.

Among the XP practices are some techniques that will help us reduce the complexity of the problem.

The Genius Function

Ideally, we should find some way that we can all be geniuses. If we are all geniuses, we can always create simple solutions to all of our problems. Something we all know is that it is easy to create simple solutions to solve simple problems. So to become a genius, all we need to do is break down all of our problems into lots of smaller problems that are easy to solve.

Any problem that I estimate to take longer than four hours to solve is too big for me to feel comfortable with. I want to accomplish at least one thing in a day; if I accomplish more than that, I feel even better. This is a personal decision, but most people I have worked with aim for four hours or less. I have never worked with anyone[1] who was comfortable with tasks that lasted more than six hours. If I get a task that I estimate will take longer than my maximum four-hour period, I break it down into several smaller tasks. If any of those tasks will take longer than four hours, I break those down again. Something that I have found interesting is that the most effective developers I know break down the majority of their tasks so that they require less than an hour to complete.

Example of Problem Breakdown

Let's see how this works, beginning with a trivial example before I lead you through a more complex problem breakdown.

Suppose the customer has asked us to develop a piece of software that emulates a calculator on the desktop. So this is our problem: develop a .NET Framework calculator. Easy. So how long will the development take? Do you know immediately? I don't, but I bet it will take longer than four hours, so I need to break down the problem. Here's a list of smaller problems:

1. **Add function**—Add two numbers together
2. **Subtract function**—Subtract one number from another
3. **Divide function**—Divide one number by another
4. **Multiply function**—Multiply two numbers together

Can I give time estimates for each of these functions? I feel confident that I can, and I estimate between one and two hours for each function. Some people might ask, "Why so much?" After all, they know that I am an experienced developer and might think that I should be able to put a simple calculator together pretty quickly. I am estimating that this will take me up to a day to complete based on my experience. I am breaking the problem down into even smaller steps, and I will go through that process with you now for the add function.

Task	Time to Complete
Test for adding zeros	15 minutes
Test for adding negative numbers	15 minutes
Test for adding minimum numbers and maximum numbers (forcing overflows)	15 minutes
Test for adding positive numbers	15 minutes
User interface	15 minutes
Check code into source control and integrate with any existing solution	5 minutes
Total	1 hour 20 minutes

Notice that I emphasize writing tests. In the next chapter, you will learn why and how to write tests before the code. For now, it is important to understand that it gives me more focus on developing only what is required and nothing extra. These tests define the behavior expected of the object that will be developed.

Also notice that I am including the time it takes to write these tests, to develop the user interface, and integrate the code with a source control system. All these extra things take time that is often not accounted for by less-experienced developers and project managers. These extra things are often the reason so many software projects

run late; developers too often fail to account for the "other stuff" that we need to do to write high-quality code.

With less-trivial problems, you will often find yourself asking some tricky questions, and this is where having contact with the customer becomes important. It is hard to break tasks down without being able to get answers to questions you have. Consider the preceding example. I have made an assumption that the calculator will deal with negative numbers. What if negative numbers are not permitted or required by the system? Having the customer present while doing this task breakdown will make this very apparent. The conversation in our eXtreme .NET team might go something like this:

Chris: Why are you testing for adding negative numbers? We don't want to add negative numbers!

Pete: Panic! What should happen if we get a negative number?

Chris: You should inform the user that negative numbers are not allowed.

Eddie: Okay, change that test to "Test for handling negative inputs, 10 minutes."

Remember one of the practices discussed in Chapter 1 was the whole team. Here is an example where having the customer as part of the team can make a difference. If you cannot get a customer to be present during a task breakdown session, try to get him or her on the phone or instant messenger. Failing that, try your manger or other developers. At least they may spot some mistakes or incorrect assumptions you have made before you actually commit them to code. You should e-mail your customer the results from each task breakdown session so that they can review them and change their minds or reset the direction.

We return to this scenario later on in this book to write code for some of these tasks.

Problem Breakdown Exercise

Hopefully you have a good idea of what we're aiming for now, so let's try to tackle a slightly more complex problem. This problem may still seem trivial in comparison to many real-world problems you will encounter, but we have to start somewhere, and this should give you at least introduce you to the concept of breaking down customer stories into tasks.

Exercise 3-1: Defining the Story

The first thing to happen is that the customer will provide the development team with a story. A *story* is a simple description of a piece of functionality that the system needs to perform. In a full XP environment, the team will play the planning game. This involves the customer writing a number of stories on cards (usually index cards). Then the development team will place rough time estimates on each story. The customer then decides which stories to do first. The planning game provides a long-term view of how the project could pan out eventually, but also includes short-term goals for the next few weeks. By repeating the planning game every few weeks, the accuracy of the estimates should get better and you enable the customer to choose the direction of the project on a smaller scale. Figure 3-1 shows a sample story card.

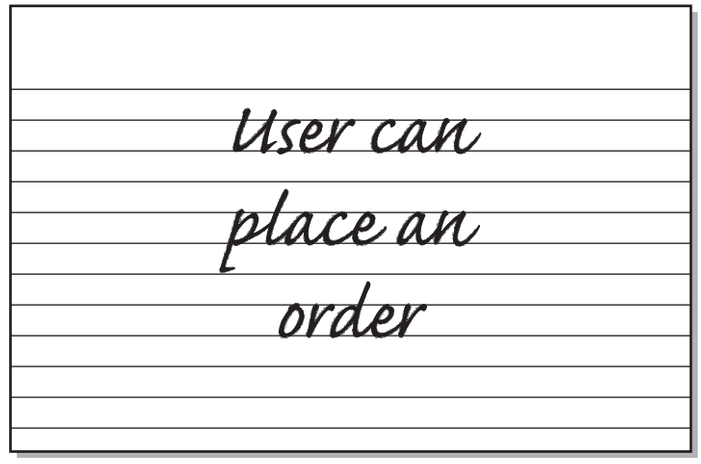

Figure 3-1 An XP story card.

A story can be considered similar (and are often compared) to a UML Use Case. The story should not explain all the details of the functionality. A story provides a rough idea of the scope. Stories are promises to hold conversations between the developers and the customer. This conversation occurs when the time comes to begin implementing the story. The conversation enables the developers to explore exactly what tasks will have to be carried out to satisfy the customer that the functionality described in the story is finished.

The best way to start this is to begin a conversation with your customer. Let's see how our team does it:

Eddie: What would you like this software to do?

Chris: I need to be able to display the time in different places in the world on my desktop.

Pete: You mean you want clocks, one for each place in the world?

Chris: Yes, but I want to be able to choose which places to display the time for, not just have a clock for every time zone that exists.

Eddie: Okay, how about we start with simply doing one clock for one time zone and then build the system from there.

Chris: Um… well, I want to choose the time zone and have multiple clocks.

Eddie: Yes, I understand that, but we want to get some core functionality for you to see as soon as possible, and then we can add to the software from there.

Chris: Okay, as long as you are aware that we will need more than one clock.

Eddie: Sure, so what shall we put down as the first story? How about "Select time zone to display?"

Chris: Sounds good. I also want to be able to put my own label next to the clock, so if I select GMT, I want to be able to label it as London.

Pete: Fine, so the second story could be "Add custom label to selected time zone?"

Chris: Yes that's right; so what next? I can select a time zone and associate a label with it, but I need to be able to see the clock. Do you need a story for displaying the clock?

Eddie: You're right; how do you want it displayed? I assume that just a digital text output is okay? Something like "Time in London is: 14:23." How does that sound?

Chris: No, no, no! You've misunderstood me! I need to see a *clock*. You know, with hands and a round face! Like this. (The customer grabs a scrap of paper and scribbles the picture shown in Figure 3-2.)

Figure 3-2 Customer drawing.

Pete: Panic! That wasn't what I was thinking.

Eddie: Sure, we can do that, but it will take a bit more work.

Chris: How much work? Doesn't sound that hard to me.

Eddie: Well let's break it down and I'll tell you how long it will take.

Chris: I don't have time for all this; I just want a clock on my screen for different places in the world!

Eddie: Okay, we'll get back to you soon with some estimates for timeframes.

Chris: Great, when?

Eddie: In the next hour.

Chris: Oh okay, that's not so bad.

Some of this conversation probably sounds familiar. It is not uncommon for customers (managers) to be short of time and want answers on the spot. As developers, we need to respect that they have other pressures and do what we can to help them. It is important not to let their stress get to us and force us to make on-the-spot decisions or provide time estimates based on nothing more than numbers plucked out of the air. Notice how Eddie, who has more experience, steers the conversation. Eddie is guiding both Pete and Chris through the process in a way that gives them the best chance of succeeding.

Exercise 3-2: Breaking Down the Stories into Small Subtasks

From the preceding conversation, Eddie and Pete can draw the following story cards:

1. Allow the user to select a time zone.
2. Allow the user to associate a custom label with the time zone selected.
3. Draw a clock for the time zone and a label for it.

From these stories, could you give reasonable time estimates? For the first two, you may be able to get reasonably close (apart from one issue). The third one is a little trickier because the clock has to show the correct time and be updated as the time changes. Let's see what Eddie and Pete come up with when they break these stories down.

Story 1: Allow User to Select a Time Zone

Task	Estimated Time
Get a list of time zones from the operating system	?
Test to check selected time zone is valid	15 minutes
Test to validate selected time zone is stored in memory	15 minutes
Test to validate correct behavior when invalid time zone selected	15 minutes
User interface to allow user to select a time zone from the list	10 minutes
Check code into source control	5 minutes

Eddie has put a question mark next to the first task because neither he nor Pete are sure what is involved in getting all the time zones from the operating system. They know the time zones must be there because when they set the Windows clock they can select from a list of available time zones. To find out where this list is stored, they need to carry out some investigation and experimentation.

In XP terminology, this is called *doing a spike* or *spiking*. This spike will be another task. Because they don't know how long the spike will take, they set an upper limit to the amount of time spent spiking before providing feedback to the customer. Eddie and Pete now have a list of tasks that looks like this:

Task	Estimated Time
Spike to discover how OS stores time zone information	4 hours
Get a list of time zones from the operating system	? (based on outcome of spike)
Test to check selected time zone is valid	10 minutes
Test to validate selected time zone is stored in memory	10 minutes
Test to test behavior when invalid time zone selected	10 minutes
User interface to allow user to select a time zone from the list	10 minutes
Check code into source control	5 minutes

They can now break down the next story.

Story 2: Allow User to Associate a Custom Label with the Time Zone Selected

Task	Estimated Time
Test to validate correct behavior when label is blank	10 minutes
Test to check label is stored and associated with time zone	10 minutes
Test to check label with non text characters is valid	10 minutes
User Interface to allow user to enter label for selected time zone	15 minutes
Check code into source control and integrate with existing code	5 minutes

Finally, they can tackle the third story: Draw a clock for the time zone and a label for it. Eddie thinks this story is too big, so he proposes they break it down into smaller stories:

3.1 Draw clock face
3.2 Draw label next to clock face
3.3 Draw clock hands
3.4 Update clock hands to reflect current time in selected time zone
3.5 As time changes, redraw clock hands to reflect current time in selected time zone

From these five stories, they can create a set of tasks for which it should be reasonably easy to provide time estimates. The tasks for each story are estimated as follows.

Story 3.1: Draw Clock Face

Task	Estimated Time
Code to draw circle	15 minutes
Code to draw 12 marks to indicate time intervals	15 minutes
Check code into source control and integrate with existing code	5 minutes

Story 3.2: Draw Label Next to Clock Face

Task	Estimated Time
Code to calculate offset from clock face to draw label	15 minutes
Code to draw label	10 minutes
Check code into source control and integrate with existing code	5 minutes

Story 3.3: Draw Clock Hands

Task	Estimated Time
Code to get time	10 minutes
Code to draw hour hand	15 minutes
Code to draw minute hand	10 minutes
Check code into source control and integrate with existing code	5 minutes

Story 3.4: Update Clock Hands to Reflect Current Time in Selected Time Zone

Task	Estimated Time
Test converting current time when time zone is in summer time savings	20 minutes
Test converting current time when time zone is in standard time	20 minutes
Code to draw clock hands based on converted time (dependent on story 3.3)	10 minutes
Check code into source control and integrate with existing code	5 minutes

Story 3.5: As Time Changes, Redraw Clock Hands to Reflect Current Time in Selected Time Zone

Task	Estimated Time
Thread function which updates clock hands based on current time (dependant on story 3.4)	20 minutes
Check code into source control and integrate with existing code	5 minutes

Based on this task breakdown, Eddie and Pete can get back to Chris, their customer, with some confidence in the estimated timeframes. The timeframes are still estimates, but the developers are happier that they have thought about the job and the tasks to complete. They can also explain that the first thing they need to do is spend half a day understanding how time zone information is stored in the operating system. When they know how the time zones are stored, they can use them in the program.

As you can see, the estimated times for each task are small; all of them except the spike are less than 30 minutes. This is a good indicator that they have got their tasks to the correct level of granularity. If you have several tasks that are four or five hours long, you should think harder about how to break them down. For me, the spike task is still cause for concern. After the customer has had time to review the task breakdowns and give the go ahead on the project, I would break down the spike task further. Let's see how Eddie and Pete go about doing this.

Exercise 3-3: Breaking Down the Subtasks Even Further

To break down the spike task, Eddie and Pete need to think about the approach to take to find the information they are after. There are some likely candidates to find out how they can access the time zone information:

1. .NET Framework support
2. Win32 API
3. Work out how the date and time Control Panel applet works
4. COM controls
5. Windows Registry

Based on the fact they have given themselves 4 hours to come up with some answers, Eddie suggests spending no longer than 45 minutes investigating each area. Because both he and Pete are working on the project, they could each take a different area to investigate. This gives Eddie and Pete a task list for the spiking tasks.

Spiking Tasks

Task	Estimated Time
Investigate .NET support for time zones	45 minutes
Investigate Win32 support for time zones	45 minutes
Work out how the date and time Control Panel applet gets it time zone data	45 minutes
Investigate whether there are any COM controls that provide access to time zone data	45 minutes
Explore the Registry for time zone data	45 minutes

If they come to a dead end before 45 minutes, they can stop and move on to the next area of investigation. Likewise, if they find a mechanism that provides access to the time zone data, they don't need to continue with any of the other spiking tasks.

Eddie and Pete are now ready to start this project with some very tightly defined tasks and outcomes at the end of each task. This tight definition of tasks enables them to move at a fast and furious pace during the working day.

Exercises to Help You Toward Genius

You now understand some of the reasoning behind breaking down stories into extremely focused tasks. This section contains some conversations between you and your customer. Read these conversations, and then try to work out the stories and break them down into tasks. At the end of the book (Appendix I), I provide some possible solutions to these exercises; however, they are not definitive answers. When carrying out the task breakdown, there is not *one right way* to do it, but there are certainly some wrong ways. Remember to keep each story adding one piece of functionality and each task doing just one thing. Each exercise should take you no more than an hour to complete.

Exercise 3-4: The Shopping Cart

Customer: I need a shopping cart for my Web site.

You: Okay, what do you need it to do?

Customer: You know, shopping cart stuff!

You: I guess you need to add items to the cart?

Customer: Of course! And take items out

You: Anything else?

Customer: Yes, I want to have a running total of the value of items in the user's cart.

You: So each user can see his or her own running total?

Customer: Yep.

You: Does the system need to remember what's in my cart the next time I log on?

Customer: No. I don't think so; they can start again.

You: Okay, give me an hour and I'll get you some timeframes and an idea of how we'd go about developing that.

Exercise 3-5: Derived Stock Market Data

Stock trader (customer): I want to be able to see a set of derived data for my portfolio.

You: Where is your portfolio stored?

Stock trader: I don't know! It's in my software somewhere; I think it saves a file to my disk.

You: Okay, and what do you want calculated?

Stock trader: The open price for the day, the high for the day, the low for the day, the close price for the previous day, and how much I've made on that stock.

You: Hold on. That's too much for me to do at one time; let's go through those one by one, first the open price for the day.

Stock trader: Yes, that's the first price that the stock gets quoted at after the market opens at 9 a.m.

You: So, the high for the day is the highest price the stock is quoted at from the time the market opens till it closes?

Stock trader: Yes, the market closes at 5 p.m.

You: And the low is the lowest price between open and close?

Stock trader: Exactly.

You: The close for the previous day? Is that the price at 5 p.m.?

Stock trader: Pretty much. It's the last quoted price before the market closes.

You: And the last one was how much you've made on the stock?

Stock trader: Yes. My software tells me how much of a stock I've got and how much I paid for it, so you can use that information to work out how much I've made, or lost, on the stock.

You: Okay, I'll have to work out how to get that information from your software.

Stock trader: Great. Well, now you know what I want, tell me how long will it take and how much it will cost me.

You: Give me an hour and I'll get back to you with some estimates.

Exercise 3-6: What's the Weather Like?

Customer: I am running a hot air balloon race around the world and I need to have some way to know what the weather is like in the areas around where the balloons will be.

You: So you want me to write some software that does weather forecasting?

Customer: No! That would be a bit expensive, and the race would probably have finished before you get it right!

You: So what do you want?

Customer: Well, there's all this weather information on the Web, and I thought if we could somehow collate it, it would be really useful for me.

You: You want me to develop a portal site for weather information Web sites around the world?

Customer: Would that be like a Web page where I could click a map of the world and get a weather report for the location I click?

You: Yes, we could do that. Do you have a list of sites that we can use to get the weather from?

Customer: No. I thought you'd know what was best

You: I'll put together a list for you to look through. Is there anything else you need it to do?

Customer: Oh yes, I need to know if there are any severe weather conditions around the world that should be avoided.

You: So you need a list of typhoons, hurricanes, hailstorms, floods?

Customer: Yes. I need to be able to see that list at all times; it must be up-to-date.

You: Okay, let me take all this information away and get back to you with some estimates for delivery times

Customer: Great, I have to dash; I have a race to run. Call me in an hour with what you have.

Exercise 3-7: The Unfinished Solution

Manager (customer): Ah, there you are; I've been looking for you.

You: Why? What do you want?

Manager: Remember Bob was working on that Mailer program?

You: Sure, he seemed to be enjoying doing a .NET project.

Manager: Yes, well he's been pulled off because he knows everything about the C++ code in the TY project.

You: And…?

Manager: I need you to pick up where Bob left off and finish the Mailer program.

You: Okay, can I have a few days with Bob to hand over the project?

Manager: It's too late for that. Bob's already on a plane to HQ to work with the other guys on TY.

You: Oh… so did Bob have a spec or any Use Cases or stories he was working from?

Manager: I don't know. I never saw him write anything down in the entire time I've worked with him. That guy is just so smart he remembers everything you tell him.

You: Great, but I need to know what the software is supposed to do!

Manager: Okay, calm down. I'll go through it with you now. It is a program that enables you to collect a group of contacts together with their e-mail addresses and then send one e-mail that goes to all of them but with different bits being only for certain people on the list.

You: Okay, so do you know how far Bob got?

Manager: Yes, he had built a way of creating a contact, with an e-mail address and a collection of tags, and adding that contact to a collection of contacts. He said it was all stored in an XML file.

You: That's a good start; sounds like each contact has a profile?

Manager: That is the idea.

You: So the next thing to do would be to provide the user with a mechanism to enter an e-mail with tags in it?

Manager: Yes, and then send the e-mail to each person in the contact collection.

You: Only sending each person the tags that included contact details in their profile?

Manager: Yes. How long will that take you to do? Because we need you back on the other project shortly.

You: Let me get back to you on that.

Conclusion

Although this chapter has not focused on anything that is special to .NET, it has introduced you to a technique you should be able to use whenever you are developing software. From the ideas presented, you have hopefully gained an understanding of why XP is often quoted as being a methodology that requires a large dose of self-discipline. You should also have gained some insight as to the value of working closely with your customers to define exactly what they want.

A valuable input that I have received when running through these task breakdown exercises in the classroom is that developers often jump straight into breaking down the problem with the idea of coding the entire solution themselves. The more experienced developers tend to do a search on the Web for any existing tools or components that do some (or even all) of the work for them. Then they can think about how to break down the rest of the work.

I hope you can see the importance of some of the key values emerging from the content of this chapter, with *communication*, *feedback*, *simplicity*, and *respect* all being evident through the dialogues presented. Now you have to find the *courage* to do it yourself at work.

[1] I have worked with more than 100 developers in the past two years while consulting with software development teams.