Chapter 5

Acceptance Tests

Surely you aren't going to assume you're getting what you need. Prove that it works! Acceptance tests allow the customer to know when the system works and tell the programmers what needs to be done.

Customers, remember that you have the right to see progress in the form of a working system, passing repeatable tests that you specify. Well, here's the responsibility part of that: specifying the acceptance tests.

Every system undergoes testing to find out whether it works as needed. Some don't get this testing until they go into use and actual users discover flaws for themselves. This leads to pain; pain leads to hatred; hatred leads to suffering. Don't do that—test before you ship.

Some systems put off overall testing until right before release. They often eat into the testing time with schedule overruns, but even if they allocate the full testing period, this slows things down. The reason is simple: programmers forget what they did. If I write a program today and test it a week from now, my recollection of how I wrote it will have faded, in favor of whatever I'm working on then. So, when I get the defect indication, I won't be able to guess quickly where the problem is. This leads to long debugging sessions, and slows the project down. Don't do that—test right away.

XP values feedback, and there is no more important feedback than early information on how well the program works. If it were possible to have the computer beep at the programmer one second after she made a mistake, there'd be fewer mistakes in the world today. We can't usually test the system's function every second (but see *Unit Tests*, page 93, for a discussion of how hard we try). With acceptance tests, the sooner we catch the mistake, the sooner we can make the program work. The customer responsibility is to provide those acceptance tests as part of each iteration. If you can get them to the programmers by the middle of each iteration, the project will go faster. You will get more business value by the deadline. That's a promise.

There are many different ways to implement the acceptance testing on your project, and the programmers will pick one. We'll talk about that below. In any case, you need to specify the tests.

What must I test? you're probably asking. The official XP answer is, you only have to test the things that you want to have work. Let's be clear about that: if it's worth having the programmers build it, it's worth knowing that they got it right.

Some projects have a legacy system they are replacing, and they can get their test data from the legacy. In this case, your job will be to select the legacy inputs and outputs you want tested. Some projects use spreadsheets from the customer that provide the inputs and expected

outputs. Smart XP programmers will extract the information from the spreadsheet automatically and read it into the system. Some projects use manually calculated values that are typed in by someone on the team.

Some customers give input numbers to the programmers and check the output by just reading it. There is an important issue with this one that has to be mentioned. This is an anti-pattern—a bad idea—called Guru Checks Output. Checking the output of a computer is very error-prone. It's easy to look at the numbers and decide that they look correct. It's also easy to be wrong when you do that. Far better to have the expected answers up front, even if they have to be computed by hand.

One more thing. The rights refer to repeatable tests. All tests in an XP project must be automated. We give you the ability to move very rapidly, and to change your requirements any time you need to. This means that the code will be changing rapidly. The only way to move rapidly with confidence is to have a strong network of tests, both unit and acceptance, that ensure that changes to the system don't break things that already work. The acceptance tests must be automated: insist on it as your right.

We aren't much into dire warnings and predictions, but here's one that's a sure thing: the defects in your system will occur where you don't test. Push your acceptance tests to the limit, in breadth and in depth. You'll be glad you did.

Automating the Tests

The tests must be automated, or you won't get your XP merit badges. However, there are lots of ways this can be done. The specific choice is up to your programmers. Here are some starting ideas:

- If the program is a batch program, reading inputs and producing outputs, make a standard series of input files, run the program, check the output manually (and carefully) once, and then write simple scripts that compare the test output to the known good output.
- Use the above trick for reports and lists from any program, batch or not.
- Build on the *xUnit* (page 105) testing framework. Write functional tests as programs. Better yet, make a little scripting language that the programmers can use. Then grow it and make it easier until the

customers can use it. Maybe provide a little GUI that displays more detail than the red bar/green bar.

- ✧ Allow the customers to set up tests in their favorite spreadsheet, then read in the test and execute it. Some teams read the spreadsheet data from exported files. Some actually use the "automation" feature of the spreadsheet to read the information. A few actually export test results back to the spreadsheet! This isn't as hard as it sounds—take a look at it and see if it's within your team's ability.
- ✧ Build an input recorder into the product, let the customers exercise it once to define a test. Spill output to files and check them automatically.
- ✧ Use simple file-based tools like grep and diff and Perl to check results. You can get a lot of testing automated very quickly with these tools.

Always build your acceptance tests to be automatic, but build the automation simply and incrementally as you actually need it. It's easy to get sucked into investing in test automation instead of business value. Get the tests automated, but don't go overboard.

Timeliness

Acceptance tests really need to be available in the same iteration as the story is scheduled. Think about it—you want to score the development based on getting stories done, and the only way to know if they are really done is to run the tests.

Programmers, you have the right to know what is needed. Insist on this right in the form of automated functional tests. You'll be glad you did.

Customers, you have the right to see progress in a running system, proven to work by automated tests that you specify. Insist on this right, and do your part by providing the necessary information.

Acceptance Test Samples

At first it can be difficult figuring out how to do acceptance tests. With a little practice, it becomes easy.

Here are some of the sample stories from the story chapter, with suggestions for how they might be tested.

Union dues vary by union and are taken only in the first pay period of the month. The system computes the deduction automatically. The amount is shown in the attached table.

This is an easy one, of course: the test pays some employees from various unions and checks whether they are charged the right dues, in the first pay period. Another test checks the subsequent pay periods to make sure dues are not taken.

When a transaction causes a customer's account to go into overdraft, transfer money from the overdraft protection account, if any.

Also easy. Some sample customers, with and without overdraft protection. Test what happens if there isn't enough money in the overdraft account—note that the story is probably incomplete. If the customer writes the test correctly, the programmers will see and deal with the problem quickly enough.

When a transaction causes a customer's account to go into overdraft, send an e-mail showing the transaction and balance to the customer. If overdraft protection is in effect, show the overdraft transaction and the resulting account balances in the e-mail.

Same as previous test, except that e-mails should be sent. Send them all to a fixed account; have the programmers write code that reads them and checks their formats, so the test can be automatic.

Produce a statement for each account, showing transaction date, number, payee, and amount. A sample statement is attached—make the report look approximately like the sample.

It's tempting to look at the report manually. That way lies the dark side. Check the report once, very carefully, then compare it mechanically against the good one thereafter.

When the GPS has contact with two or fewer satellites for more than 60 seconds, it should display the message "Poor satellite contact," and wait for confirmation from the user. If contact improves before confirmation, clear the message automatically.

Functionally testing small hardware devices can be tricky. Do you have a version of the hardware hooked up to testing equipment that can read the display, provide fake satellite input, and so on? Depending on your cost of testing and need for reliability, this might be valuable enough to do.

Allow the user to add new service types to the system's initial list. For example, he may wish to add a special entry for getting the car washed at the high school's "free" wash. Include the standard fields amount and date, plus allow the user to add any additional text or numeric fields. Reports should automatically sum any numeric fields. (Programmer note: story needs splitting. Please separate text and numeric fields into two stories, plus one for the summing.)

This program, probably conceived as just an interactive GUI-based system, clearly needs a programmatic interface. It's not hard, given even a simple interface, to have a little scripting language set up that simulates GUI commands and checks report output. Remember to compare the report file to a correct one rather than check it by hand.