Acceptance TDD Explained

Lasse Koskela
This article is based on a chapter from the book "Practical TDD and Acceptance TDD for Java Developers"
http://www.manning.com/koskela/

In the spacecraft business no design can survive the review process, without first answering the question—how are we going to test this thing?

—Glen B. Alleman, Director Strategic Consulting for Lewis & Fowler

In the previous chapters, we have explored the developer-oriented practice of test-driven development, covering the fundamental process itself as well as a host of techniques for employing TDD on a variety of Java technologies. In this chapter, we'll take the core idea of TDD and apply it to the overall product development process.

TDD helps software developers produce working, high-quality code that's maintainable and, most of all, reliable. Our customers are rarely, however, interested in buying *code*. Our customers want software that helps them to be more productive, make more money, maintain or improve operational capability, take over a market, and so forth. This is what we need to deliver with our software—functionality to support business function or market needs. Acceptance test-driven development (acceptance TDD) is what helps developers build high-quality software that fulfills the business's needs as reliably as TDD helps ensure the software's technical quality.

Acceptance TDD helps coordinate software projects in a way that helps us deliver exactly what the customer wants when they want it, and that doesn't let us implement the required functionality only half way. In this chapter, we will learn what acceptance test-driven development is, why we should consider doing it, what these mysterious acceptance tests are, and how acceptance TDD integrates with TDD.

Acceptance test-driven development as a name sounds similar to test-driven development, doesn't it? So, what's the difference between the two? Surely it has something to do with the word *acceptance* prefixed onto TDD, but is there something else beneath the surface that differentiates the two techniques? What exactly are these acceptance tests? And what are we accepting with our tests in the first place? These are the questions we're going to find answers to as we start our journey into the world of acceptance tests and acceptance test-driven development.

We'll begin by describing a lightweight and extremely flexible requirements format called *user stories*. After all, we need to know how our system should behave, and that's exactly what user stories tell us. From there, we'll continue by exploring what acceptance tests are and what kinds of properties they should exhibit. By then, we will know what our requirements—the user stories—might look like, and we will know how our acceptance tests should look, which means we'll be ready to figure out how we work with these artifacts or what the process of acceptance TDD looks like.

An essential property of acceptance TDD is that it's a team activity and a team process, which is why we'll also discuss team-related topics such as the roles involved and who might occupy

these roles. We'll also ask ourselves why we should do this and respond by identifying a number of benefits of acceptance TDD. Finally, we'll talk about what exactly we are testing with our acceptance tests and what kind of tools we have at our disposal.

But now, let us introduce ourselves with a handy requirements format we call *user stories*.

1 Introduction to user stories

User stories are an extremely simple way to express requirements. In its classic form, a user story is a short sentence stating *who* does *what* and *why*. In practice, most stories just tell us who and what, with the underlying motivation considered apparent from the context. The reason a story is typically only one sentence long (or, in some cases, just one or two words that convey meaning to the customer and developers) is that the story is not intended to document the requirement. The story is intended to *represent* the requirement, acting as *a promise of a future conversation* between the customer and the developer.

1.1 Format of a story

A number of people have suggested writing user stories that follow an agreed format such as "As a (role) I want (functionality) so that (benefit)." However, I and a number of other proponents of user stories for requirements management recommend not fixing the format as such but focusing on the user story staying on a level of detail that makes sense, using terms that make sense to the customer. This is not to say that such a template would be a bad idea—which it isn't; it's just that one size doesn't fit all, and the people in your organization might feel differently about the format than I do [1].

On the physical format of user stories

In part because user stories are concise, many co-located teams keep their user stories written on small index cards or sticky notes, managing them on a whiteboard or other task board. This method is beneficial because communication is clear and progress is immediately obvious. Therefore, more and more multisite projects are also adopting such physical *story cards* for managing their work locally. After all, the benefits often far outweigh the hassle of relaying the status from the task board to electronic format for distribution. The use of index cards on an early XP project gave birth to the mnemonic of 3 Cs: card, conversation, confirmation.

1.2 Power of storytelling

Hannah Arendt, a German political scientist, has said, "storytelling reveals meaning without committing the error of defining it. [2] This particular quote eloquently communicates how user stories focus on meaning without stumbling on nitty-gritty details.

Inside every story is another one trying to come out

Just like there are multiple solutions to most computing problems, there is always another way to write a given user story. Indeed, it might make sense to take a closer look at a user story before rubberstamping it as a technical story. There is usually a way to express the story in a way that conveys the underlying value - the rationale - of the story. If you can't figure out that value, try again.

User stories are in many ways a form of storytelling, which is an effective medium for transferring knowledge. For one, people like listening to stories. Storytellers are good at keeping our attention - a lot more so than, say, structured documents of equal volume - and it's not just

audible stories that have this advantage; prose with a storyline and context is far more interesting reading than a seemingly unconnected sequence of statements.

Let's see how much this property shows through in practice by looking at a couple of examples of user stories.

1.3 Examples of user stories

To get a better idea of what user stories look like, here are some examples of the kinds of user stories I personally tend to write:

- "Support technician sees customer's history onscreen at the start of a call"
- "The system prevents user from running multiple instances of the application simultaneously"
- "Application authenticates with the HTTP proxy server"

These user stories express just enough for the customer to be able to prioritize the feature in relation to other features and for the developer to be able to come up with a rough effort estimate for the story. Yet these stories don't burden the developer by prescribing the implementation, and they don't drown the team with excessive detail.

The first story about a technical-support application doesn't tell us what the screen will look like; and the second story doesn't talk about desktop shortcuts, scanning process listings, and so on. They convey *what* provides value to the customer—not *how* the system should provide that value. The third story is a bit different. It's clearly a technical user story, not having much to do with business functionality. It does, however, have enabling value, and it expresses that need in a clear manner. Furthermore, although harder to quantify, some technical stories might create value for the customer through lower total cost of ownership.

That's about all we're going to say about user stories for now. For a more in-depth description of user stories as a requirements management and planning tool, a great pick would be Mike Cohn's book *User Stories Applied* (Addison-Wesley, 2004).

As we already mentioned, the format of a user story doesn't matter all that much as long as it communicates the necessary information—who, what, why—to all involved parties, either explicitly or implicitly. In fact, just like the format of a story isn't one-size-fits-all, using stories as a requirements-management or planning tool isn't in any way a requirement for doing acceptance test-driven development—it's a natural fit.

Now that we know what the mysterious stories are (or what they can be), let's figure out what we mean by *acceptance tests*.

2. Acceptance tests

Acceptance tests are specifications for the desired behavior and functionality of a system. They tell us, for a given user story, how the system handles certain conditions and inputs and with what kinds of outcomes. There are a number of properties that an acceptance test should exhibit; but before taking a closer look, let's see an example.

2.1 Example tests for a story

Let's consider the following example of a user story and see what our acceptance tests for that particular story might look like. I present you figure with 1.

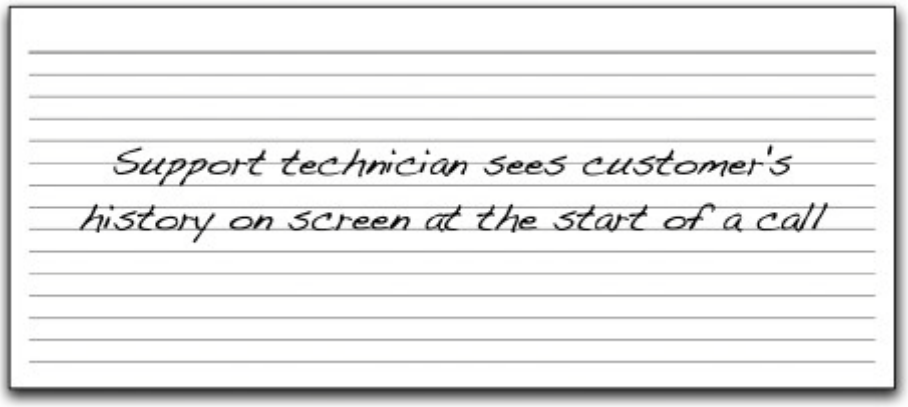

Figure 1. Example of a user story, written on a story card

The functionality that we're interested in is for the system to obtain and display the customer's history of records when a call comes through the customer support system. I might, for example, think of the tests for this story that are scribbled down as figure 2.

These three tests would essentially tell us whether the system behaves correctly from the perspective of a user—conditions of satisfaction. They tell us nothing about how the system implements that behavior.

Now, with these example tests in mind, let's look at some essential properties of acceptance tests, starting with who owns them and who writes them.

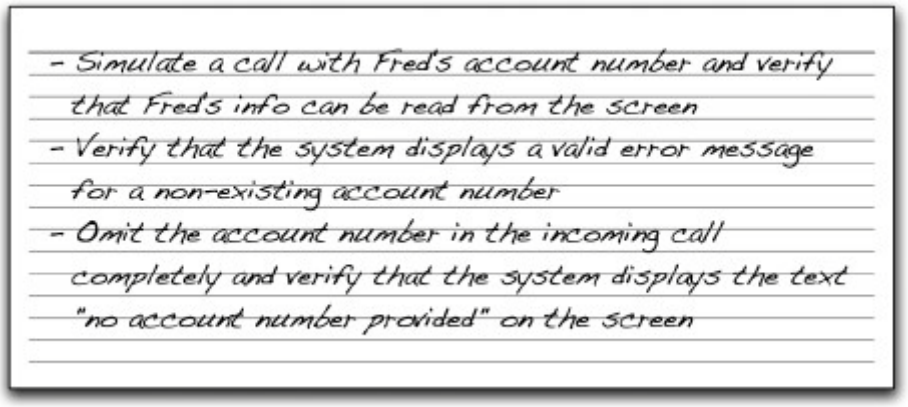

Figure 2. Example tests for the story, written on the back of the story card from figure 1

2.2 Properties of acceptance tests

So far, you've probably deduced that acceptance tests are typically short and somewhat informal. There's more to the nature of acceptance tests, however, and next we're going to look at some general properties.

To make a long story short, acceptance tests are

- Owned by the customer
- Written together with the customer, developer, and tester
- About the *what* and not the *how*

- Expressed in the language of the problem domain
- Concise, precise, and unambiguous

Let's expand these sound bites one by one and see what they mean.

Owned by the customer

Acceptance tests should be owned by the customer because their main purpose is to specify acceptance criteria for the user story, and it's the customer—the business expert—who is best positioned to spell out those criteria. This also leads to the customer being the one who should ideally be writing the acceptance tests.

Having the customer write the acceptance tests helps us avoid a common problem with acceptance tests written by developers: Developers often fall into the pit of specifying technical aspects of the implementation rather than specifying the feature itself. And, after all, acceptance tests are largely a specification of functionality rather than tests for technical details (although sometimes they're that, too).

Written together

Even though the customer should be the one who owns the acceptance tests, they don't need to be the only one to write them. Especially when we're new to user stories and acceptance tests, it is important to provide help and support so that nobody ends up isolating themselves from the process due to lack of understanding and, thus, being uncomfortable with the tools and techniques. By writing tests together, we can encourage the communication that inevitably happens when the customer and developer work together to specify the acceptance criteria for a story.

With the customer in their role as domain expert, the developer in the role of a technical expert, and the tester in a role that combines a bit of both, we've got everything covered. Of course, there are times when the customer will write stories and acceptance tests by themselves—perhaps because they were having a meeting offsite or because they didn't have time for a discussion about the stories and their accompanying tests.

The same goes for a developer or tester who *occasionally* has to write acceptance tests without access to the customer. On these occasions, we'll have to make sure that the necessary conversation happens at some point. It's not the end of the world if a story goes into the backlog with a test that's not ideal. We'll notice the problem and deal with it eventually. That's the beauty of a simple requirement format like user stories!

Another essential property for good acceptance tests ties in closely with the customer being the one who's writing the tests: the focus and perspective from which the tests are written.

Focus on the what, not the how

One of the key characteristics that make user stories so fitting for delivering value early and often is that they focus on describing the *source* of value to the customer instead of the mechanics of *how* that value is delivered. User stories strive to convey the needs and wants—the *what* and *why*—and give the implementation—the *how*—little attention. In the vast majority of cases, the customer doesn't care how the business value is derived. Well, they shouldn't. Part of the reason many customers like to dictate the *how* is our lousy track record as an industry. It's time to change that flawed perception by showing that we can deliver what the customer wants as long as we get constant feedback on how we're doing.

Let's look at an example to better illustrate this difference between what and why and how. Figure 3 shows an example of acceptance tests that go into too much detail about the solution rather than focusing on the problem - the customer's need and requirement. All three tests shown on the card address the user interface - effectively suggesting an implementation, which isn't what we want. While doing that, they're also hiding the real information - what are we actually testing here?" - behind the technology.

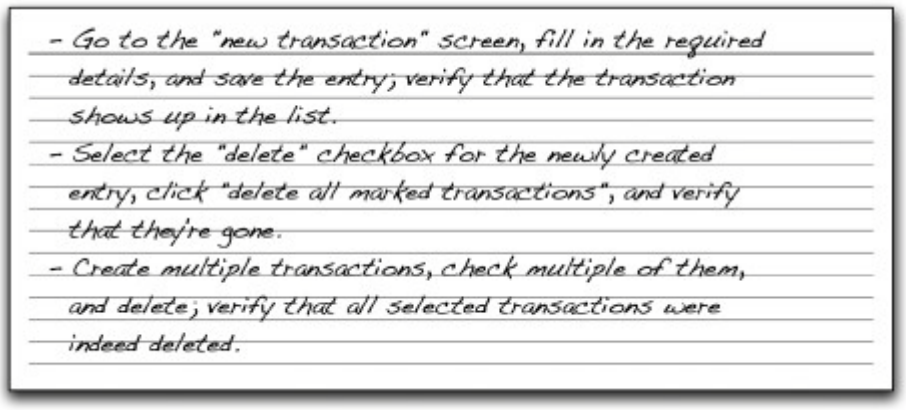

Figure 3. Acceptance test focusing too much on the implementation

Instead, we should try to formulate our tests in terms of the problem and leave the solution up to the developers and the customer to decide and discuss at a later time when we're implementing the tests and the story itself. Figure 4 illustrates a possible rewrite of the tests in figure 3 in a way that preserves the valuable information and omits the unnecessary details, which only clutter our intent.

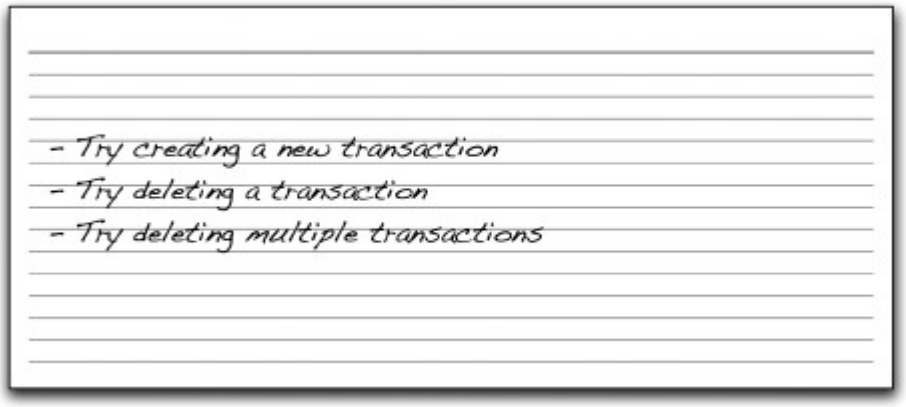

Figure 4. Trimmed-down version of the tests from figure 3

Notice how, by reading these three lines, a developer is just as capable of figuring out what to test as they would be by reading the more solution-oriented tests from figure 3. Given these two alternatives, which would you consider easier to understand and parse? The volume and focus of the words we choose to write our tests with have a big effect on the effectiveness of our tests as a tool. We shouldn't neglect that fact.

There's more to words, though, than just volume and focus. We also have to watch for our language.

Use the language of the domain

An important property of acceptance tests is that they use the language of the domain and the customer instead of geek-speak only the programmer understands. This is the fundamental requirement for having the customer involved in the creation of acceptance tests and helps enormously with the job of validating that the tests are correct and sufficient. Scattering too much technical lingo into our tests makes us more vulnerable to having a requirement bug sneak into a production release—because the customer's eyes glaze over when reading geek-speak and the developers are drawn to the technology rather than the real issue of specifying the right thing.

By using a domain language in specifying our tests, we are also not unnecessarily tied to the implementation, which is useful since we need to be able to refactor our system effectively. By using domain language, the changes we need to make to our existing tests when refactoring are typically non-existent or at most trivial.

Concise, precise, and unambiguous

Largely for the same reasons we write our acceptance tests using the domain's own language, we want to keep our tests simple and concise. We write each of our acceptance tests to verify a single aspect or scenario relevant to the user story at hand. We keep our tests uncluttered, easy to understand, and easy to translate to executable tests. The less ambiguity involved, the better we are at avoiding mistakes and the working with our tests.

We might write our stories as simple reminders in the form of a bulleted list, or we might opt to spell them out as complete sentences describing the expected behavior. In either case, the goal is to provide just enough information for us to remember the important things we need to discuss and test for, rather than documenting those details beforehand. Card, conversation, confirmation—these are the three Cs that make up a user story. Those same three Cs could be applied to acceptance tests as well.

Remember the acceptance tests we saw earlier, for the story about a customer support system? Take another look at them, back in figure 2.

Would you say these tests are simple and concise? Perhaps you would. Perhaps you wouldn't. Personally, I'd say there are some things in these tests that could be safely omitted while still preserving enough information to carry the original intent, and some things that shouldn't be there. Figure 5 shows a revamp of the same tests shown in figure 2.

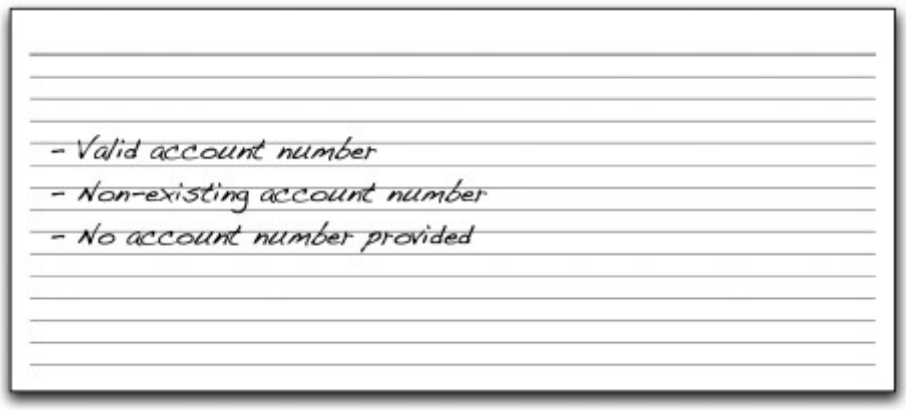

Figure 5. Revamped acceptance tests from figure 2

Notice the difference in conciseness? Notice how the developer would still know to test for the right things, provided they can ask the customer for the details, such as what kind of a message should be displayed for a non-existent account number or when the number is omitted altogether? The tests in figure 5 can be considered *more accurate* than the tests in figure 2 because they omit details that could change by the time we get around to implementing the story.

Obviously, some prefer to have more details written down for the acceptance tests than do others. Whether you're into more text or less, or whether you prefer sketching little diagrams and UI designs as part of your user stories and the accompanying acceptance tests, it's up to you and your team to decide. It's all good as long as you remember to keep your acceptance tests simple and concise, and as long as you avoid writing down ambiguous things that can be interpreted wrongly at a later stage. Specifically, avoid writing down details that are easy to find out later and that don't add crucial information for estimating the size of the story.

The last property of acceptance tests that we'll list here has more to do with automating the tests than the way or form in which they're written.

2.3 Implementing acceptance tests

Yet another common property of acceptance tests is that they might not be implemented (translation: automated) using the same programming language as the system they are testing. Whether this is the case depends on the technologies involved and on the overall architecture of the system under test. For example, some programming languages are easier to interoperate with than others. Similarly, it is easy to write acceptance tests for a web application through the HTTP protocol with practically any language we want, but it's often impossible to run acceptance tests for embedded software written in any language other than that of the system itself.

The main reason for choosing a different programming language for implementing acceptance tests than the one we're using for our production code (and, often, unit tests) is that the needs of acceptance tests are often radically different from the properties of the programming language we use for implementing our system. To give you an example, a particular real-time system might be feasible to implement only with native C code, whereas it would be rather verbose, slow, and error-prone to express tests for the same real-time system in C compared to, for example, a scripting language.

The ideal syntax for expressing our acceptance tests could be a declarative, tabular structure such as a spreadsheet, or it could be something closer to a sequence of higher-level actions written in plain English. If we want to have our customer collaborate with developers on our acceptance tests, a full-blown programming language such as Java, C/C++, or C# is likely not an option. "Best tool for the job" means more than technically best, because the programmer's job description also includes collaborating with the customer.

Now that we know something about acceptance tests and we have an idea of who's writing the tests in the first place, let's see how we use them to drive our development. What does acceptance test-driven development look like on paper?

3 Understanding the process

Test-driven development gives a programmer the tools for evolving their software in small steps, always certain of the software working as expected. This certainty comes from the programmer expressing their expectations in the form of automated unit tests. In acceptance

test-driven development, this certainty is gained not on the level of technical correctness but rather on the feature level of, "does the software do what I want it to do?"

In other words, although in TDD we're first defining the specific behavior we want our code base to exhibit and only then implementing the said behavior, in acceptance TDD we first define the specific user- or customer-valued functionality we want our system as a whole to exhibit and only then implement the said behavior, most likely using TDD as our vehicle of choice.

Because we know what acceptance tests look like, how about if we take a quick tour through the overall process of acceptance test-driven development and then broaden our view and look at what happens on the scale of a whole iteration? After that, we can go back and zoom in on the details of the more interesting bits.

About the customer

You may have noticed that user stories as a requirements-management technique tend to stress having close and frequent interaction with the customer. If you're worried about not having an on-site customer, or having a customer who's not keen on having much to do with developing the software, you can stop worrying. There are ways around this obvious limitation, and we'll talk about those later on. For now, just consider *the customer* as referring to a role rather than a specific person—a role that can be played by, say, one of the test engineers or developers who knows enough about the product's domain to be able to make the kinds of decisions a real customer would make.

But now, I present you with the process of acceptance test-driven development, distilled into four small steps.

3.1 The acceptance TDD cycle

In its simplest form, the process of acceptance test-driven development can be expressed as the simple cycle illustrated by figure 6.

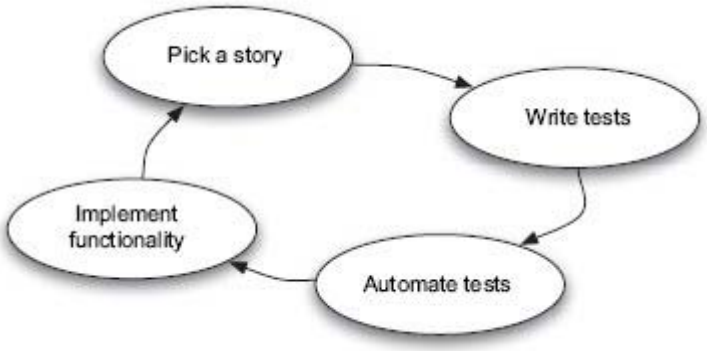

Figure 6. The acceptance TDD cycle

This cycle continues throughout the iteration as long as we have more stories to implement, starting over again from picking a user story; then writing tests for the chosen story, then turning those tests into automated, executable tests; and finally implementing the functionality to make our acceptance tests pass.

In practice, of course, things aren't always that simple. We might not yet have user stories, the stories might be ambiguous or even contradictory, the stories might not have been prioritized,

the stories might have dependencies that affect their scheduling, and so on. We'll get to these complications later. For now, let's keep our rose-tinted glasses on and continue thinking about the simple steps outlined previously. Speaking of steps, let's take a closer look at what those steps consist of.

Step 1: Pick a user story

The first step is to decide which story to work on next. Not always an easy job; but, fortunately, most of the time we'll already have some relative priorities in place for all the stories in our iteration's work backlog. Assuming that we have such priorities, the simplest way to go is to always pick the story that's on top of the stack—that is, the story that's considered the most important of those remaining. Again, sometimes, it's not that simple.

Generally speaking, the stories are coming from the various planning meetings held throughout the project where the customer informally describes new features, providing examples to illustrate how the system should work in each situation. In those meetings, the developers and testers typically ask questions about the features, making them a medium for intense learning and discussion. Some of that information gets documented on a story card (whether virtual or physical), and some of it remains as tacit knowledge. In those same planning meetings, the customer prioritizes the stack of user stories by their business value (including business risk) and technical risk (as estimated by the team).

What kinds of issues might we have when picking stories from this stack of user stories? There are times when the highest-priority story requires skills that we don't possess, or we consider not having enough of. In those situations, we might want to skip to the next task to see whether it makes more sense for us to work on it. Teams that have adopted pair programming don't suffer from this problem as often. When working in pairs, even the most cross-functional team can usually accommodate by adjusting their current pairs in a way that frees the necessary skills for picking the highest priority task from the pile.

The least qualified person

The traditional way of dividing work on a team is for everyone to do what they do best. It's intuitive. It's safe. But it might not be the best way of completing the task. Arlo Belshee presented an experience report at the Agile 2005 conference, where he described how his company had started consciously tweaking the way they work and measuring what works and what doesn't. Among their findings about stuff that worked was a practice of giving tasks to the *least qualified person*. For a full closure on their experience and an explanation of why this approach works, listen to Arlo's interview at the Agile Toolkit Podcast website (http://agiletoolkit.libsyn.com/).

There can be more issues to deal with regarding picking user stories, but most of the time the solution comes easily through judicious application of common sense. For now, let's move on to the second step in our process: writing tests for the story we've just picked.

Step 2: Write tests for a story

With a story card in hand (or onscreen if you've opted for managing your stories online), our next step is to write tests for the story. If you paid attention earlier in this chapter, we just learned that it's the customer who should be writing the tests. So how does this play out?

The first thing to do is, of course, get together with the customer. In practice, this means having a team member sit down with the customer (they're the one who should own the tests, remember?) and start sketching out a list of tests for the story in question.

As usual, there are personal preferences for how to go about doing this, but my current preference (yes, it changes from time to time) is to quickly scram out a list of rough scenarios or aspects of the story we want to test in order to say that the feature has been implemented correctly. There's time to elaborate on those rough scenarios later on when we're implementing the story or implementing the acceptance tests. At this time, however, we're only talking about coming up with a bulleted list of things we need to test—things that have to work in order for us to claim the story is done.

We already saw a couple of examples of the kind of tests we're referring to when we discussed the properties of acceptance tests. For example, you might want to peek back at figure 4, showing three tests on the back of a story card. That is the kind of rough list we're after in this step.

On timing

Especially in projects that have been going on for a while already, the customer and the development team probably have some kind of an idea of what's going to get scheduled into the next iteration in the upcoming planning meeting. In such projects, the customer and the team have probably spent some time during the previous iteration sketching acceptance tests for the features most likely to get picked in the next iteration's planning session. This means that we might be writing acceptance tests for stories that we're not going to implement until maybe a couple of weeks from now. We also might think of missing tests during implementation, for example, so this test-writing might happen pretty much at any point in time between writing the user story and the moment when the customer accepts the story as completed.

Once we have such a rough list, we start elaborating the tests, adding more detail and discussing about how this and that should work, whether there are any specifics about the user interface the customer would like to dictate, and so forth. Depending on the type of feature, the tests might be a set of interaction sequences or flows, or they might be a set of inputs and expected outputs. Often, especially with flow-style tests, the tests specify some kind of a starting state, a context the test assumes is part of the system.

Other than the level of detail and the sequence in which we work to add that detail, there's a question of when—or whether—to start writing the tests into an executable format. Witness step 3 in our process: automating the tests.

Step 3: Automate the tests

The next step once we've got acceptance tests written down on the back of a story card, on a whiteboard, in some electronic format, or on pink napkins, is to turn those tests into something we can execute automatically and get back a simple pass-or-fail result. Whereas we've called the previous step *writing tests*, we might call this step *implementing* or *automating* those tests.

In an attempt to avoid potential confusion about how the executable acceptance tests differ from the acceptance tests we wrote in the previous step, let's pull up an example. Remember the acceptance tests in figure 5? We might turn those tests into an executable format by using a variety of approaches and tools. The most popular category of tools (which we'll survey later) these days seems to be what we call *table-based tools*. Their premise is that the tabular format of tables, rows, and columns makes it easy for us to specify our tests in a way that's both human

and machine readable. Figure 7 presents an example of how we might draft an executable test for the first test in figure 5, "Valid account number".

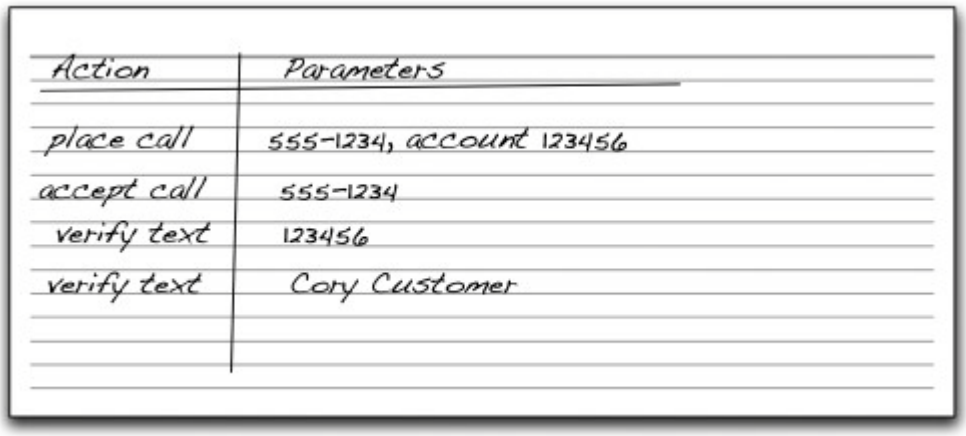

Figure 7. Example of an executable test, sketched on a piece of paper

In figure 7, we've outlined the steps we're going to execute as part of our executable test in order to verify that the case of an incoming support call with a valid account number is handled as expected, displaying the customer's information onscreen. Our test is already expressed in a format that's easy to turn into a tabular table format using our tool of choice—for example, something that eats HTML tables and translates their content into a sequence of method invocations to Java code according to some documented rules.

Java code? Where did that come from? Weren't we just talking about tabular formats? The inevitable fact is that most of the time, there is not such a tool available that would understand our domain language tests in our table format and be able to wire those tests into calls to the system under test. In practice, we'll have to do that wiring ourselves anyway—most likely the developers or testers will do so using a programming language. To summarize this duality of turning acceptance tests into executable tests, we're dealing with expressing the tests in a format that's both human and machine readable and with writing the plumbing code to connect those tests to the system under test.

On style

The example in figure 7 is a *flow-style test*, based on a sequence of actions and parameters for those actions. This is not the only style at our disposal, however. A declarative approach to expressing the desired functionality or business rule can often yield more compact and more expressive tests than what's possible with flow-style tests. The volume of detail in our tests in the wild is obviously bigger than in this puny example. Yet our goal should—once again—be to keep our tests as simple and to the point as possible, ideally speaking in terms of *what* we're doing instead of *how* we're doing it.

With regard to writing things down (and this is probably not coming as a surprise), there are variations on how different teams do this. Some start writing the tests right away into electronic format using a word processor; some even go so far as to write them directly in an executable syntax. Some teams run their tests as early as during the initial authoring session. Some people, myself included, prefer to work on the tests alongside the customer using a physical medium, leaving the running of the executable tests for a later time. For example, I like to sketch the executable tests on a whiteboard or a piece of paper first, and pick up the computerized tools only when I've got something I'm relatively sure won't need to be changed right away.

The benefit is that we're less likely to fall prey to the technology—I've noticed that tools often steal too much focus from the topic, which we don't want. Using software also has this strange effect of the artifacts being worked on somehow seeming more formal, more final, and thus needing more polishing up. All that costs time and money, keeping us from the important work.

In projects where the customer's availability is the bottleneck, especially in the beginning of an iteration (and this is the case more often than not), it makes a lot of sense to have a team member do the possibly laborious or uninteresting translation step on their own rather than keep the customer from working on elaborating tests for other stories. The downside to having the team member formulate the executable syntax alone is that the customer might feel less ownership in the acceptance tests in general—after all, it's not the *exact* same piece they were working on. Furthermore, depending on the chosen test-automation tool and its syntax, the customer might even have difficulty reading the acceptance tests once they've been shoved into the executable format dictated by the tool.

Just for laughs, let's consider a case where our test-automation tool is a framework for which we express our tests in a simple but powerful scripting language such as Ruby. Figure 8 highlights the issue with the customer likely not being as capable of feeling ownership of the implemented acceptance test compared to the sketch, which they have participated in writing. Although the executable snippet of Ruby code certainly reads nicely to a programmer, it's not so trivial for a non-technical person to relate to.

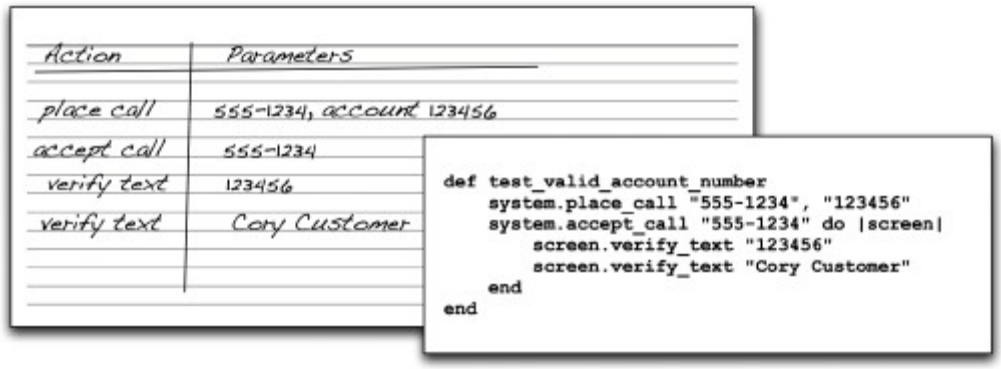

Figure 8. Contrast between a sketch an actual, implemented executable acceptance test

Another aspect to take into consideration is whether we should make all tests executable to start with or whether we should automate one test at a time as we progress with the implementation. Some teams—and this is largely dependent on the level of certainty regarding the requirements—do fine by automating all known tests for a given story up front before moving on to implementing the story.

Some teams prefer moving in baby steps like they do in regular test-driven development, implementing one test, implementing the respective slice of the story, implementing another test, and so forth. The downside to automating all tests up front is, of course, that we're risking more unfinished work—inventory, if you will—than we would be if we'd implemented one slice at a time. My personal preference is strongly on the side of implementing acceptance tests one at a time rather than try getting them all done in one big burst. It should be mentioned, though, that elaborating acceptance tests toward their executable form during planning sessions could help a team understand the complexity of the story better and, thus, aid in making better estimates.

Many of the decisions regarding physical versus electronic medium, translating to executable syntax together or not, and so forth also depend to a large degree on the people. Some customers

have no trouble working on the tests directly in the executable format (especially if the tool supports developing a domain-specific language). Some customers don't have trouble identifying with tests that have been translated from their writing. As in so many aspects of software development, it depends.

Regardless of our choice of how many tests to automate at a time, after finishing this step of the cycle we have at least one acceptance test turned into an executable format; and before we proceed to implementing the functionality in question, we will have also written the necessary plumbing code for letting the test-automation tool know what those funny words mean in terms of technology. That is, we will have identified what the system should do when we say "select a transaction" or "place a call"—in terms of the programming API or other interface exposed by the system under test.

To put it another way, once we've gotten this far, we have an acceptance test that we can execute and that tells us that the specified functionality is missing. The next step is naturally to make that test pass—that is, implement the functionality to satisfy the failing test.

Step 4: Implement the functionality

Next on our plate is to come up with the functionality that makes our newly minted acceptance test(s) pass. Acceptance test-driven development doesn't say how we should implement the functionality; but, needless to say, it is generally considered best practice among practitioners of acceptance TDD to do the implementation using test-driven development—the same techniques we've been discussing in the previous parts of this book.

In general, a given story represents a piece of customer-valued functionality that is split—by the developers—into a set of *tasks* required for creating that functionality. It is these tasks that the developer then proceeds to tackle using whatever tools necessary, including TDD. When a given task is completed, the developer moves on to the next task, and so forth, until the story is completed—which is indicated by the acceptance tests executing successfully.

In practice, this process means plenty of small iterations within iterations. Figure 9 visualizes this transition to and from test-driven development inside the acceptance TDD process.

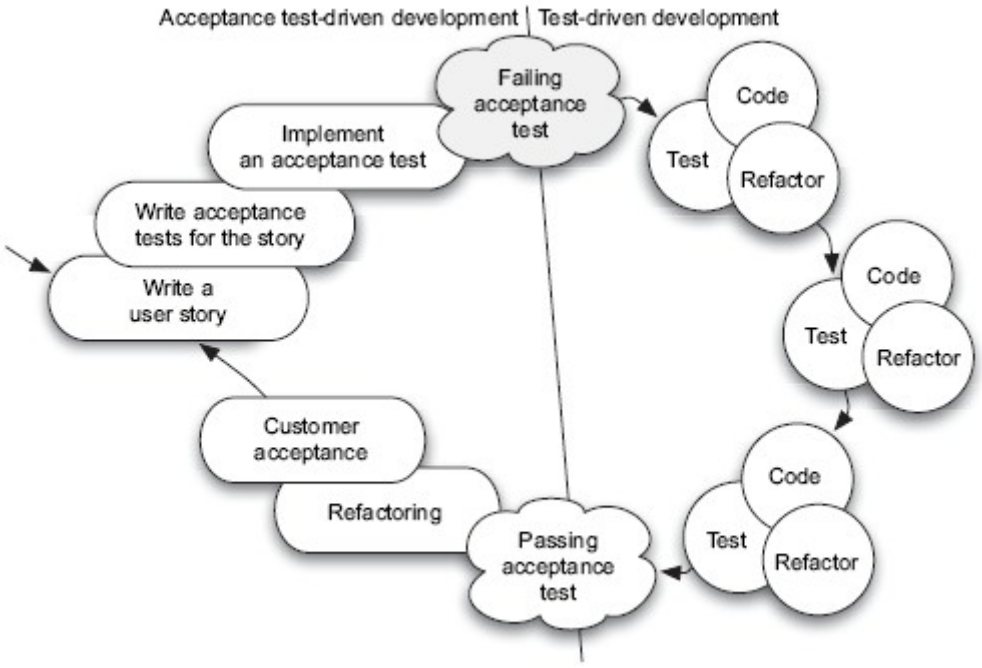

Figure 9. The relationship between test-driven development and acceptance test-driven development

As we can see, the fourth step of the acceptance test-driven development cycle, implementing the necessary functionality to fix a failing acceptance test, can be expanded into a sequence of smaller TDD cycles of test-code-refactor, building up the missing functionality in a piecemeal fashion until the acceptance test passes. The proportions in the figure should not be considered to reflect reality, however. Whereas the TDD cycle might range from one minute to a dozen, we might be chopping out code for a couple of hours or even the whole day before the acceptance test is passing.

While the developer is working on a story, frequently consulting with the customer on how this and that ought to work, there will undoubtedly be occasions when the developer comes up with a scenario—a test—that the system should probably handle in addition to the customer/developer writing those things down. Being rational creatures, we add those acceptance tests to our list, perhaps after asking the customer what they think of the test. After all, they might not assign as much value to the given aspect or functionality of the story as we the developers might.

At some point, we've iterated through all the tasks and all the tests we've identified for the story, and the acceptance tests are happily passing. At this point, depending on whether we opted for automating all tests up front (which I personally *don't* recommend) or automating them just in time, we either go back to Step 3 to automate another test or to Step 1 to pick a brand-new story to work on.

It would probably not hurt to walk around a bit and maybe have a cup of coffee, possibly check out your email. Getting acceptance tests passing is intensive work. As soon as you're back from the coffee machine, we'll continue with a broader view of how this simple four-step cycle with its small steps fits into the bigger picture of a complete iteration within a project.

3.2 Acceptance TDD inside an iteration

A healthy iteration consists mostly of hard work. Spend too much time in meetings or planning ahead, and you're soon behind the iteration schedule and need to de-scope (which might translate to another planning meeting…ugh!). Given a clear goal for the iteration, good user stories, and access to someone to answer our questions, most of the iteration should be spent in small cycles of a few hours to a couple of days writing acceptance tests, collaborating with the customer where necessary, making the tests executable, and implementing the missing functionality with our trusted workhorse, test-driven development.

As such, the four-step acceptance test-driven development cycle of picking a story, writing tests for the story, implementing the tests, and implementing the story is only a fraction of the larger continuum of a whole iteration made of multiple—even up to dozens—of user stories, depending on the size of your team and the size of your stories. In order to gain understanding of how the small four-step cycle for a single user story fits into the iteration, we're going to touch the zoom dial and see what an iteration might look like on a time line with the acceptance TDD–related activities scattered over the duration of a single iteration.

Figure 10 is an attempt to describe what such a time line might look like for a single iteration with nine user stories to implement. Each of the bars represents a single user story moving through the steps of writing acceptance tests, implementing acceptance tests, and implementing the story itself. In practice, there could (and probably would) be more iterations within each story, because we generally don't write and implement *all* acceptance tests in one go but rather proceed through tests one by one.

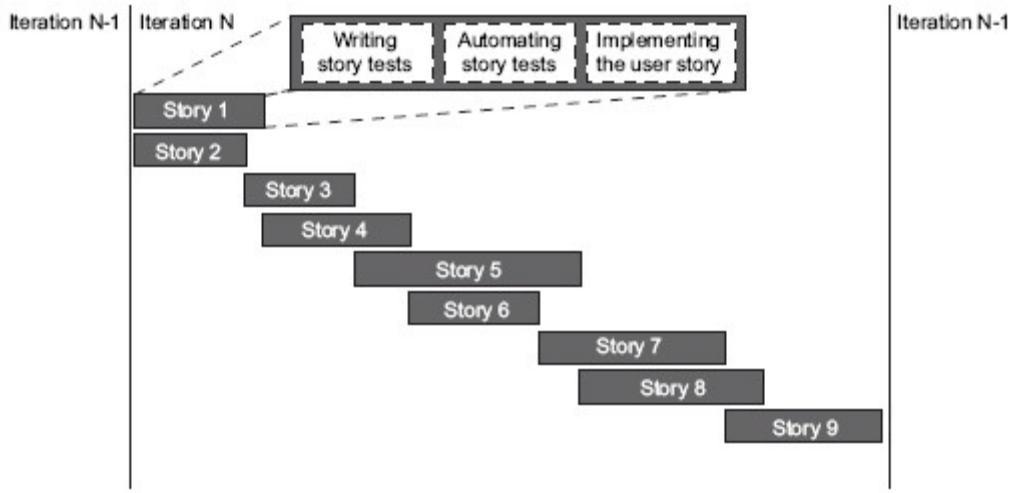

Figure 10. Putting acceptance test-driven development on time line

Notice how the stories get completed almost from the beginning of the iteration? That's the secret ingredient that acceptance TDD packs to provide indication of real progress. Our two imaginary developers (or pairs of developers and/or testers, if we're pair programming) start working on the next-highest priority story as soon as they're done with their current story. The developers don't begin working on a new story before the current story is done. Thus, there are always two user stories getting worked on, and functionality gets completed throughout the iteration.

So, if the iteration doesn't include writing the user stories, where are they coming from? As you may know if you're familiar with agile methods, there is usually some kind of a planning meeting in the beginning of the iteration where the customer decides which stories get implemented in that iteration and which stories are left in the stack for the future. Because we're scheduling the stories in that meeting, clearly we'll have to have those stories written before the meeting, no?

That's where continuous planning comes into the picture.

Continuous planning

Although an iteration should ideally be an autonomous, closed system that includes everything necessary to meet the iteration's goal, it is often necessary—and useful—to prepare for the next iteration during the previous one by allocating some amount of time for pre-iteration planning activities. Otherwise, we'd have long-lasting planning meetings, and you're probably not any more a friend of long-lasting meetings than I am. Suggestions regarding the time we should allocate for this continuous planning range from 10–15% of the team's total time available during the iteration. As usual, it's good to start with something that has worked for others and, once we've got some experience doing things that way, begin zeroing in on a number that seems to work best in our particular context.

In practice, these pre-iteration planning activities might involve going through the backlog of user stories, identifying stories that are most likely to get scheduled for the next iteration, identifying stories that have been rendered obsolete, and so forth. This ongoing pre-iteration planning is also the context in which we carry out the writing of user stories and, to some extent, the writing of the first acceptance tests. The rationale here is to be prepared for the next iteration's beginning when the backlog of stories is put on the table. At that point, the better we

know our backlog, the more smoothly the planning session goes, and the faster we get back to work, crunching out valuable functionality for our customer.

By writing, estimating, splitting if necessary, and prioritizing user stories before the planning meeting, we ensure quick and productive planning meetings and are able to get back to delivering valuable features sooner.

When do we write acceptance tests?

It would be nice if we had all acceptance tests implemented (and failing) before we start implementing the production code. That is often not a realistic scenario, however, because tests require effort as well—they don't just appear from thin air—and investing our time in implementing the complete set of acceptance tests up front doesn't make any more sense than big up-front design does in the larger scale. It is much more efficient to implement acceptance tests as we go, user story by user story.

Teams that have dedicated testing personnel can have the testing engineers work together with the customer to make acceptance tests executable while developers start implementing the functionality for the stories. I'd hazard a guess that most teams, however, are much more homogeneous in this regard and participate in writing and implementing acceptance tests together, with nobody designated as "the acceptance test guy."

The process is largely dependent on the availability of the customer and the test and software engineers. If your customer is only onsite for a few days in the beginning of each iteration, you probably need to do some trade-offs in order to make the most out of those few days and defer work that can be deferred until after the customer is no longer available. Similarly, somebody has to write code, and it's likely not the customer who'll do that; software and test engineers need to be involved at some point.

We start from those stories we'll be working on first, of course, and implement the user story in parallel with automating the acceptance tests that we'll use to verify our work. And, if at all possible, we avoid having the same person implement the tests and the production code in order to minimize our risk of human nature playing its tricks on us.

Again, we want to keep an eye on putting too much up-front effort in automating our acceptance tests—we might end up with a huge bunch of tests but no working software. It's much better to proceed in small steps, delivering one story at a time. No matter how valuable our acceptance tests are to us, their value to the customer is negligible without the associated functionality.

The mid-iteration sanity check

I like to have an informal sanity check in the middle of an iteration. At that point, we should have approximately half of the stories scheduled for the iteration running and passing. This might not be the case for the first iteration, due to having to build up more infrastructure than in later iterations; but, especially as we get better at estimating our stories, it should always be in the remote vicinity of having 50% of the stories passing their tests.

Of course, we'll be tracking story completion throughout the iteration. Sometimes we realize early on that our estimated burn rate was clearly off, and we must adjust the backlog immediately and accordingly. By the middle of an iteration, however, we should generally be pretty close to having half the stories for the iteration completed. If not, the chances are that there's more work to do than the team's capacity can sustain, or the stories are too big compared to the iteration length.

Learning from our mistakes, we've come to realize that a story's burn-down rate is constantly more accurate a source of prediction than an inherently optimistic software developer. If it looks like we're not going to live up to our planned iteration content, we decrease our load.

Decreasing the load

When it looks like we're running out of time, we decrease the load. We don't work harder (or smarter). We're way past that illusion. We don't want to sacrifice quality, because producing good quality guarantees the sustainability of our productivity, whereas bad quality only creates more rework and grinds our progress to a halt. We also don't want to have our developers burn out from working overtime, especially when we know that working overtime doesn't make any difference in the long run. Instead, we adjust the one thing we can: the iteration's scope—to reality. In general, there are three ways to do that: swap, drop, and split. Tom DeMarco and Timothy Lister have done a great favor to our industry with their best-selling books *Slack* (DeMarco; Broadway, 2001) and *Peopleware* (DeMarco, Lister; Dorset House, 1999), which explain how overtime reduces productivity.

Swapping stories is simple. We trade one story for another, smaller one, thereby decreasing our workload. Again, we must consult the customer in order to assure that we still have the best possible content for the current iteration, given our best knowledge of how much work we can complete.

Dropping user stories is almost as straightforward as swapping them. "This low-priority story right here, we won't do in this iteration. We'll put it back into the product backlog." But dropping the lowest-priority story might not always be the best option, considering the overall value delivered by the iteration—that particular story might be of low priority in itself, but it might also be part of a bigger whole that our customer cares about. We don't want to optimize locally. Instead, we want to make sure that what we deliver in the end of the iteration is a cohesive whole that makes sense and can stand on its own.

The third way to decrease our load, splitting, is a bit trickier compared to dropping and swapping—so tricky that we'd better give the topic its own little section.

Splitting stories

How do we split a story we already tried hard to keep as small as possible during the initial planning game? In general, we can split stories by function or by detail (or both). Consider a story such as *"As a regular user of the online banking application, I want to optionally select the recipient information for a bank transfer from a list of most frequently and recently used accounts based on my history so that I don't have to type in the details for the recipients every time."*

Splitting this story by function could mean dividing the story into "…from a list of recently used accounts" and "…from a list of most frequently used accounts." Plus, depending on what the customer means by "most frequently and recently used," we might end up adding another story along the lines of "…from a weighted list of most frequently and recently used accounts" where the weighted list uses an algorithm specified by the customer. Having these multiple smaller stories, we could then start by implementing a subset of the original, large story's functionality and then add to it by implementing the other slices, building on what we have implemented for the earlier stories.

Splitting it by detail could result in separate stories for remembering only the account numbers, then also the recipient names, then the VAT numbers, and so forth. The usefulness of this approach is greatly dependent on the distribution of the overall effort between the details—if

most of the work is in building the common infrastructure rather than in adding support for one more detail, then splitting by function might be a better option. On the other hand, if a significant part of the effort is in, for example, manually adding stuff to various places in the code base to support one new persistent field, splitting by detail might make sense.

Regardless of the chosen strategy, the most important thing to keep in mind is that, after the splitting, the resulting user stories should still represent something that makes sense—something valuable—to the customer.

References

[1] Ron Jeffries, "Essential XP: Card, Conversation, Confirmation," *XP Magazine*, August 30, 2001

[2] Hannah Arendt, *Men in Dark Times* (Harvest Books, 1970).