

How to be a Product Director

Brian Marick, marick@exampler.com, www.exampler.com

Overview

Some projects call it “the Customer.” Others call it “the product owner.” Some call it “the goal donor.” I and a few others call it the *Product Director*. Like a film director, the product director is the one person with the clearest vision of the final result. But—like a film director—the role isn’t a passive one of “expressing the vision.” It’s an active role, one of pointing the work of other people in a particular direction, evaluating the results, and adjusting the direction based on the reality of what the last bout of work produced.

The *Product Director*’s job is to tell a team what product to produce. The aim is to produce the best product possible, given the time allowed and the team available.

It’s the hardest job on an Agile project.

The job revolves around four verbs: *inform*, *observe*, *adjust*, and *represent*.

The product director *informs* the programmers of what to do next by providing them with *stories*. In order to estimate the cost of a story and implement it successfully, the programmers need:

1. a general description of what the product is supposed to do that it doesn’t currently do,
2. specific examples of desired product behavior, and
3. information about why users care about the new behavior and how they’ll use it.

A *story* is a promise from the product director to tell the programmers enough about a desired next step that they can successfully estimate the work involved, perform it, and deliver what the product director expected.

No matter how much you inform them, the programmers will not know as much as you. For that reason, it’s important to *observe* their finished work to help clear up misunderstandings. The more frequently you observe, the better help you’ll be able to give.

As Cem Kaner puts it, at the beginning of a project you know less about it than you ever will again. If your vision of the product doesn’t change during the project, you’re either inhumanly good or you’re failing to *adjust* your vision as (1) you learn more about what different interest groups want of the project, and (2) the programmers learn what kinds of features are easy to add and what kinds are hard.

Finally, you *represent* the project to the business outside it. You’re the face of the project. You need to present it and its status to the interest groups who care about it. (The more often, the better.) You need to respond to their concerns and turn them into stories for the programmers to implement. That will require making tradeoffs; not everyone can be completely satisfied.

Roles versus People

Strictly, the product director is a role. That role can be shared by several people. I recommend against starting out that way. First learn the job, then delegate away pieces of it. If you delegate too soon, it's too hard to learn the job.

It can seem too hard to free up one person to be product director. Instead, it seems more feasible to find three people, have them shed some of their tasks, and make them collectively the product director. In my opinion, that's asking for trouble. In addition to the added difficulty learning the job, the "shedding part of your work" has a way of not happening, at least not fully, and—even if the shedding works—the total time freed up probably isn't enough to do a good job as product director.

If I were told I should spend less than 50% of my time being product director for a project with 4+ programmers, I'd make sure "they" explicitly acknowledged that the end product will be disappointing, and I'd make sure they knew I suspected that no one really thinks the product is worth building.

But that kind of thing is why I'm a consultant, not an employee. You'll have to judge how hard you can afford to push back.

The iteration

The iteration is the heartbeat of the Agile project. Iterations are usually one or two weeks long. It's a team decision (not yours alone) how long the iteration should be. (I recommend experimenting with both and seeing which seems better.)

The most important thing about an iteration is that it should end with a *potentially shippable product*. Let me emphasize that:

Every iteration ends with a potentially shippable product.

There is probably nothing more important to the success of an Agile project. "Potentially shippable" means two things:

1. The product has some new behavior that's visible and valuable to some user. (Some stories are implemented.)
2. That behavior *works*. It's finished, not 90% finished. It's tested. There are no known bugs. (See below for more about this.) As far as you're concerned, there's very little chance you'll have any reason to revisit one of the finished stories.

If it's shippable, why not ship it? You should put iteration results out in front of users as often as they'll tolerate it. (There are three main reasons users don't want new versions: they're afraid the new one will be broken, they don't want to have to learn a changed user

interface, and it's a hassle reinstalling or re-customizing it. Work to eliminate or reduce those reasons.)

I prefer to organize every iteration around a *theme*, a short description of what area of the product will be worked on or what's going to be better about the next version. Here are some themes for a project I'm directing:

I'd like to finish up the core part of the articles.

The theme for this week is the mechanics of membership.

I'd like to start working on moving the Agile Development magazine over to this site. That's mostly a matter of getting group membership & permissions working, and of whacking off a bit of SC49 (adding different content types - in this case user-editable structured text and PDF).

As you can see, the themes don't have to be elaborate (and they don't have to be understandable to anyone outside the team). They're mainly a way of choosing a coherent and consistent set of stories that, added together, represents a clear jump in the featurefulness of the product.

Stories

When you look at it, there's not much to a story. Here are two I've written recently:

visitor can subscribe to RSS feed of news

member can edit personal information

That's everything there is to those two stories. A story is just a marker, something to point to when talking about some new feature. It's the talking that's the important part. That's not to say I won't sometimes give more detail. Here are two more stories I wrote:

visitor can see Agile Alliance demographics

Have there be a page (not linked from the home page for now) that breaks Agile Alliance members down into membership by country. (Magazine advertisers want that.)

Nothing fancy for this iteration.

Don't forget "Unknown" for those who didn't list a country.

administrator or editor can see Agile Development download statistics

Advertisers want the number of downloads of Agile Development. For this moment, just give us a page with total

number of hits on the PDF, for each issue. This page can be publicly readable.

That's still not all that detailed.

No matter the detail, you'll need to explain to the programmers what they have to do to satisfy you that they've completed the story. The first part of the explanation is about who uses the feature and why. For example, the explanation for a set of stories about scheduling treatments in a university veterinary clinic would tell how the day begins at 8. That's when each student "SOAPS" her cases. (SOAP stands for "subjective/objective analysis and plan.") Depending on how the cases are being treated, there are other tasks throughout the day. For example, an animal with mastitis (infection of the udder) might have one of its "quarters" (teats) "stripped" (milked) every three hours.

A big part of what you're doing here is teaching the programmers some of the language of the business: words like "SOAP", "strip", and "quarter". The better they understand the language, the faster and more accurately they'll be able to work on the product. They will probably use those words to name parts of the guts of the program.¹

Discussion is best done in person, around a whiteboard, one that will end up looking like the following.

¹ How much explaining you need to do depends on the business domain and where you are in the project. I could write "visitor can subscribe to RSS feed of news" because the programmer understands as well as I do how and why people use RSS feeds. I can also assume he remembers our earlier discussions about the difference between a "visitor" and a "member", about what "news" is, and about how "news" differs from an "event".

Strip RR QTR - 6, 9... ← 1st of day

When	← what	cow	Where
9	SOAP	Betsy	stall 1
9	milk right rear quarter	"	"
12 noon	"	"	"
3	"	"	"

in order by #

NOT AT 6 - caretakers.

9...3pm

You'll notice that this is a specific picture, an example of the treatment of a particular case with mastitis. It's important to explain with examples for three reasons. (1) People learn from examples. Physics students aren't just told $F=ma$, they solve a lot of specific problems that teach them what that means. (2) Talking about examples helps reveal special cases. For example, if a programmer said, "so the student has to strip the quarter at 6 a.m. 9, noon, 3, 6,...", the product director would stop him there and say, "Not at 6 p.m.—the caretakers will do that as part of their normal milking." (3) The programmers will use those examples in their programming:

Programmers turn examples into code.

What that specifically means is that programmers will turn the examples into *tests* written in a programming language. When they pass the tests, they will consider themselves finished with the story.

Tests record an understanding between you and the programmers.

It annoys everyone when the programmers pass the test, tell you the story's done, you try out the new feature, and it's not what you wanted. The programmers might not have understood the examples correctly, or the examples weren't complete enough. For example, if some programmer didn't happen to list *all* the times a quarter got stripped, the product director might not have been brought up short and remembered to tell them about caretakers. In a test, the programmers would have enumerated all the hours to strip, and the resulting test would be incorrect. So would the code that came after.

For that reason, especially in the beginning, you might want to review tests to see if the programmers have captured your intent. There's a problem with that:

```
public void testThatStrippingIsEveryThreeHours() {
    Clinic clinic = new Clinic(5);
    Patient animal = new Patient();
    clinic.assignTreatment(clinic, animal, new QuarterStripping(1));
    assertTrue(animal.treatmentAt(6) instanceof QuarterStripping);
    assertTrue(animal.treatmentAt(9) instanceof QuarterStripping);
    assertTrue(animal.treatmentAt(12) instanceof QuarterStripping);
    assertTrue(animal.treatmentAt(15) instanceof QuarterStripping);
    assertTrue(animal.treatmentAt(18) instanceof QuarterStripping);
    assertTrue(animal.treatmentAt(21) instanceof QuarterStripping);
    assertTrue(animal.treatmentAt(24) instanceof QuarterStripping);
    assertTrue(animal.treatmentAt(3) instanceof QuarterStripping);
    assertEquals(8, animal.treatmentCount());
}
```

The kind of tests programmers want to write are *technology facing*. They're concerned with the guts of the program. What you're interested in are *business-facing tests*. You don't want to see any words that aren't in the language of the business.

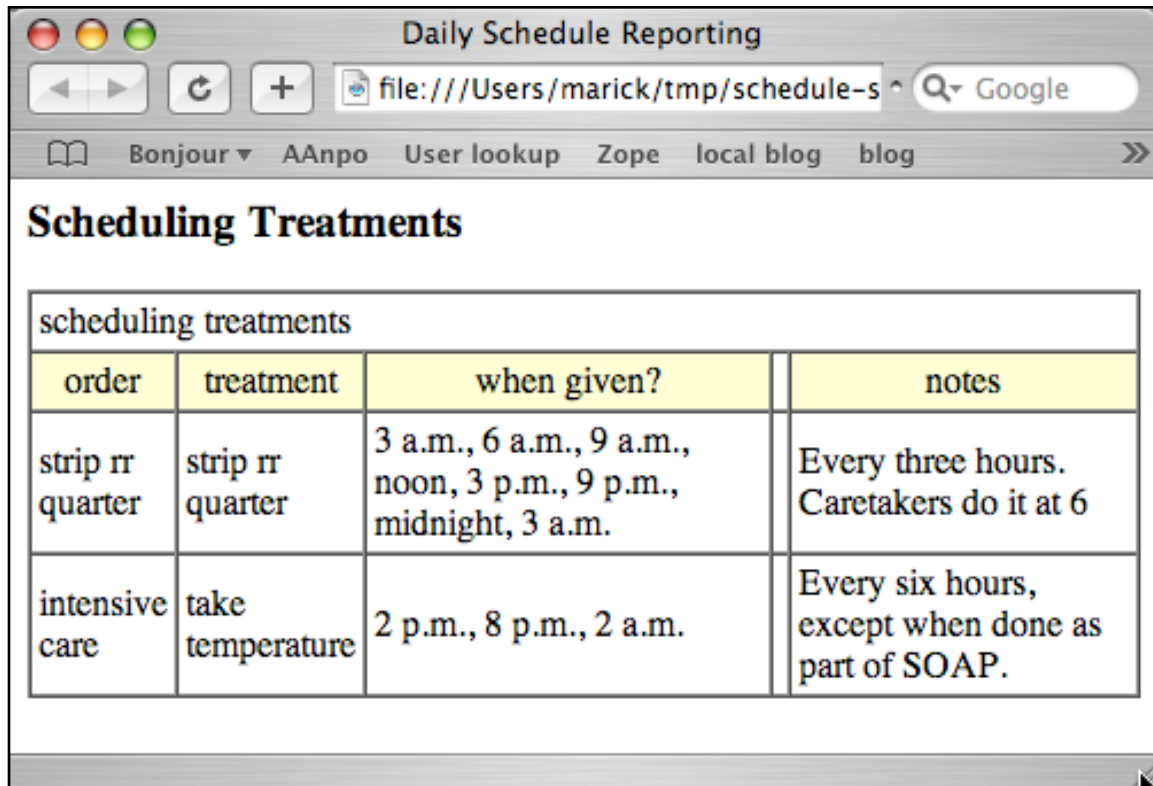
You might settle for those kinds of tests, walking through them with a programmer. I recommend you go at least one step further, which is to have the programmers write the tests in terminology you understand, but still use their favorite programming language to implement them. Here's an example:

```
public void test_that_stripping_is_every_three_hours() {
    order_strip_rr_quarter();
    quarter_is_stripped_at("6 a.m.", "9 a.m.", "noon", "3 p.m.",
                          "9 p.m.", "midnight", "3 a.m.");
    quarter_is_NOT_stripped_at("6 p.m."); // because of caretaker.
}
```

That seems to me a reasonable compromise: more readable, but still using their favorite language and tools. It does require more work of the programmer, but it has compensating advantages: (1) stripping away all the implementation detail will make overlooked special cases jump out more readily, saving work later, and (2) oftentimes the extra work

pays off because it forces the programmers to deal with issues like the representation of time. It will make for a more capable and modifiable program.²

Another option is to use a tool called Fit that lets you write tests as HTML tables, like this:



scheduling treatments			
order	treatment	when given?	notes
strip rr quarter	strip rr quarter	3 a.m., 6 a.m., 9 a.m., noon, 3 p.m., 9 p.m., midnight, 3 a.m.	Every three hours. Caretakers do it at 6
intensive care	take temperature	2 p.m., 8 p.m., 2 a.m.	Every six hours, except when done as part of SOAP.

Notice that this format is more concise; each row describes the timing of a new kind of treatment. A single HTML page could describe them all. And, because it's an HTML page, it can serve as documentation of the program that anyone can check.

The disadvantage is that this format is harder for the programmers to implement (but not all that much harder than the previous format), and it doesn't fit as well into the programmer workflow.

My preference is to start out with the second format unless the product really cries out for a tabular form. (Financial programs often do, as do many programs that are—at heart—all about getting tables out of a database.)

²You're actually likely to get the most push-back if you insist that `hardToReadNames` be converted to `easy_to_read_names`. I'm a programmer myself, but I find the fondness for the first sort inexplicable. There are good reasons for the programmers to use it in the product code (for compatibility with the rest of the world), but no particular reasons—other than habit—to use it for tests.

Pacing and Tests

The tests that come from you aren't the only tests the programmers use. They write some for their own purposes (usually called "unit tests"). Their habit—and it's a good one—is not to write all the tests in advance of coding, but rather to write a test, write code to pass it, write another test, write more code...

Because programmers don't expect or need huge batches of tests up front, don't try to finish up the tests at the beginning of the iteration. Whether you write the tests or delegate the writing to a tester or a programmer, you'll be a bottleneck. (Even if someone else writes them, she'll need you to answer questions and check her interpretations.) By not writing a test until just before a programmer needs it, the bottleneck is removed. But that means you have to be available pretty much all the time to answer questions and give examples. That's why it's best if you spend most of your time in the same room as the programmers, or (less good) if you're always reachable by phone, pager, and instant messaging. (In all cases, make sure the programmers know you're

Iteration planning

At the beginning of the iteration, you should have a list—the *backlog*—of stories you'd like to see done. It's your job to explain them well enough that the programmers can estimate how long they'll take. Estimates are usually not given in clock time, but rather in some abstract unit like "ideal time", "points", "NUTs" ("nebulous unit of time"), or "beans". The units don't matter so much as that a two-point story should take twice as long as a one-point story.

The programmers should also know their *velocity*, which is how many story points they can finish in the next iteration.

At the beginning of the iteration, you should pick some set of stories whose total points fit within the velocity. Usually, you pick the stories so that the finished set has higher total value than other alternatives. That doesn't necessarily mean that you put the backlog in priority order and just pick from the top of the list; I prefer picking related stories that make up a valuable theme (plus, usually, some stragglers, since a theme almost never fits exactly within the velocity).

It's important not to obsess over story points. It doesn't matter if every story is estimated correctly. You don't care if one story was underestimated by two points and another overestimated by two points, so long as they average out so the predicted velocity is met. Don't bog the beginning of the iteration down explaining stories in more detail than is needed for estimation.

If an estimate is bigger than you like, do *not* suggest that the programmers ought to be able to find some way to do the story faster. The team can go as fast as they can go. If you try to make them go faster, you'll almost certainly only pressure them into declaring a story done when it's incomplete or buggy. That devolves into the traditional mess, where a product has to undergo a lengthy and unpredictable "stabilization period" before it's ready to release.

Faced with a too-large story, try to find a way to pare it down. Expect to have conversations like this:

Mark: *For SC2: User can register as a Corp Member (based on company size). Should this be on-line, or handled only by the admin?*

Brian: Here's the way it works now. Someone reads our instructions for becoming a corporate member. They send me mail. I fill in a form that makes the membership, then forward the mail to the bookkeeper, who sends them an invoice. Here's a copy of the form.

It would be OK to keep doing that.

We could make them do the work of filling in the form, have me just approve it. What would that cost?

What would it then cost, upon approval, to auto-send the bookkeeper the information in the form?

What I've done is ask the programmer for the cost of the simplest implementation, then the cost of adding two features.

Mark: I think the two approaches are similar from a development POV. I still need to develop a data entry screen. It's just for different users. Admin or corporate potential user. The additional approval process is about an hour including sending the email.

An hour is less than one story point, so I told him to go ahead and do the maximum. Had it been larger, I might have balanced the dollar cost of the maximum against the dollar cost and annoyance cost of having the webmaster do more work.

As the programmers learn more about the domain, they'll get good at suggesting cheaper alternatives to your ideas. Expect that of them, but don't let implementation convenience drive your decisions. The cheapest product rarely has the best return on investment.

The programmers will also discover alternatives as they dive into the implementation. Here's an instant messaging session:

Mark: got a sec?

Brian: yes

Mark: The story Admin can see passwords ...

Mark: The passwords are currently one-way encrypted.

Mark: no way to recover them

Mark: you can reset them is that ok

Mark: it's more secure that way, but maybe you don't need that

Brian: That's fine for the admin. What would it look like for the user?

Brian: That is, what does the "forgot your password?" link lead to?

Mark: if they lose the password they get one generated for them and emailed. This is working currently.

Mark: I need to add, that they can change it at anytime

Brian: OK, so it sends to the address of record?

Mark: yes

Brian: That's OK, then.

Mark: so admin will be able to change it, but no one can see it.

Brian: good

Mark: when we talked Credit cards, I thought it was worth going that route

Brian: yes, the less user information that can be trapped , the better.

We changed the meaning of the story on the fly. The way it will now work is (I know from experience) somewhat less convenient for the administrator, but not enough so that I didn't jump at the chance for a cheaper solution that fit with the third-party package Mark was already using.

Good stories

It takes time and effort to get good at creating stories. What are the characteristics of a good story?

Good stories are small.

Small stories make estimating work better for two reasons. (1) Experience has taught us that the larger the story, the less accurate the estimate. If a programmer estimates a story as two points, it will probably really take two points of time. A story estimated at eight points might come in at eight points, but it'll be no surprise if it takes seven, or nine, or even six or ten. (2) Longer stories means there are fewer in the iteration, which means

that there's less chance for estimation errors to average out. Since you cannot direct the project without accurate estimates, small stories will be an enormous help to you.

Good stories have independent business value.

Consider an application that handles 401k plans (a kind of business-funded pension plan in the US). For tax purposes, a business has to be classified as either family-owned or not, so that will be a feature of the program. Say that it's a big feature, one that will take more than an iteration to implement. You don't want a feature to span an iteration (else the result of the end of the iteration won't be a potentially shippable product). One way to break the feature up into small stories is like this:

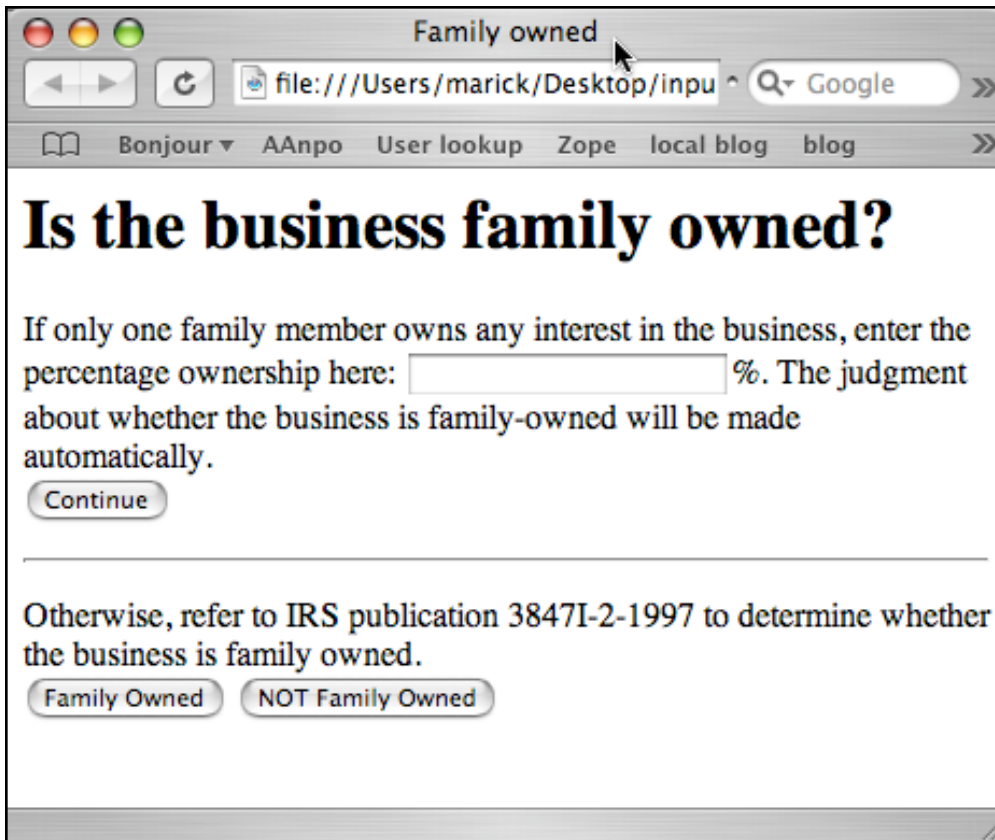
- I. data entry screen for family relationships and business ownership
- II. calculate whether business is family-owned for tax purposes
- III. result display screen for business, including justification according to IRS rules

The problem with these stories is that stories I and II are of no use to the business until story III is finished. There's no point in releasing any of them until all of them are finished; essentially, they're a single story, split up into three bite-sized pieces that make no sense from the point of view of the business.

A better way of dividing is to ask yourself what the smallest useful feature would be. Make that a story. Then ask what small thing can be added to the story to make it more useful—that's a new story. Continue until the feature is completed. Those stories might look like this:

- I. Family ownership calculation for businesses with one family member having an interest. Display yes/no judgment.
- II. Family relationship calculation for businesses with interest distributed among parents and children. Yes/no judgment.
- III. Family relationship calculation for non-descendent interest (nephews, etc.) Yes/no judgment.
- IV. All remaining family relationship cases in the IRS code. Yes/no judgment.
- V. Display justification according to IRS rules.

The first story might lead to a rudimentary input screen like this:



Each following story would elaborate on the user interface to make it collect more information. The final story would provide more elaborate results (perhaps a popup window that displays a nicely printable justification for the decision that refers back to the relevant sections of the tax code).

Good stories are not *about* the user interface.

The user interface at the top of the page is, to put it kindly, crude. That's because most of the business value from this feature comes from how it does tax code calculations for you, not in how it lets you enter data. So, following the rule of thumb that one should schedule the highest-value features first, I'd put off adding any fanciness to the GUI unless it was really cheap.

It's hard for many product directors to think that way. It's not natural to think of the user interface as a separate piece of the product. When you look at it, that's all you see, so it's natural to think of the user interface *as* the product. As a result, many product directors describe stories in terms of what the user interface should look like.

Avoid that. Try to think in terms of the business rules the product's supposed to capture. One trick to help you do that is to think of yourself calling a customer support person to describe a problem. You probably would not say things like "I went to this page and put 50 in the first field and clicked Continue and..." Instead, you'd say something like, "I'm working on a business that's 50% owned by the founder, with 10% interest for each of two children and the rest distributed to non-family members. The program says that's not a family-owned business, but my reading of the tax code is that it is."

That's the right level of description. (It's also the right level at which to write tests for this feature. Very few of the tests on a project should refer to buttons or text fields.)

Having said that, there's one compelling reason for adding a fancy user interface early. As I said on the first page, you represent the product to the business. If you're giving a demo of a product with incredibly capable innards and a crude-looking GUI, the people viewing it are prone to thinking everything is as unfinished as the GUI. If you can't persuade them out of that view, it's prudent to make the GUI more polished.

Fear "Infrastructure"

Programmers have traditionally been taught to build products bottom-up. The idea is that you think hard about the design of some cohesive, coherent chunk of code that does everything needed for a related group of features. Then you write it.

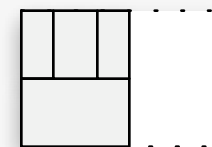
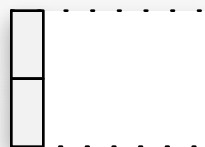
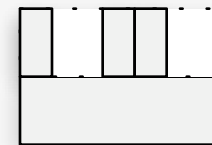
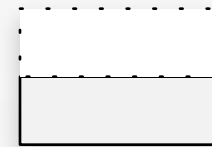
Once you've written that infrastructure, it becomes trivial to pop features on top of it, one by one. All the hard thinking has been done.

The problem with that, from an Agile point of view, is that no one sees business value—or anything much at all—until close to the end, so flaws remain hidden too long. That's the root of the old problem of a project remaining "90% done" forever and ever and ever.

The Agile style is different. Each story produces a "slice" through the product, all the way from the user interface down to the database. As the slices accumulate, the programmers exploit commonality to "grow" the infrastructure.

You make no speculative investments: at any given moment, you've only paid for infrastructure that is needed for delivered business value.

If you're working with programmers new to Agile, they'll likely find this a hard transition to make. Be patient, but don't be talked into paying for code that will supposedly make everything easy—someday.



Finishing a story

I think it's wise to try out each story after it's finished. Part of the reason is to double-check that the product does what you wanted, but another part is to see whether what you wanted was right.

Why do you test drive a car before buying it? Because imagining what something will be like is not the same as experiencing the real thing. You notice things in real life that you don't think of when envisioning the future. You should expect to get new ideas for stories. Put them in the backlog.

Don't be afraid to change your mind.

In conventional projects, programmers hate it when you change your mind after they've finished the requirements, or the design, or especially the code. In an Agile project, they shouldn't care. Because they write little or no code in anticipation of next iteration's stories, it doesn't matter to them if that story was one planned all along or one just created. They're forced to make the code malleable enough to accommodate both. (That is, a story shouldn't cost extra just because it's modifying an existing story.)

Whether to rework a feature is up to you, and you decide solely based on whether the change is worth the cost the programmers estimate.

What happens when you find a bug in a completed story? Most teams will fix the bug right away if it doesn't take much time (substantially less than a point's worth of time—an hour, say). If it takes significant time, make it a story and schedule it into an iteration when no other backlog item has a better return on the investment.

Notice that something isn't a bug if you never told the programmers about it. For example, in the family ownership stories, I mentioned nothing about input validation. (What happens if the user puts 1000 in the percent ownership box?) It would be OK for the programmers not to write input validation code, expecting it to be covered in some later iteration's story. Next time, you'll know to say what to do about error-handling, and they'll remember to ask. Over time, they'll come to know more and more what you'll want without having to be explicit about it.

In a way, it's completely irrelevant whether something you discover is a bug or not. It's either something that can be changed right away or something to become a story. The old "it's a bug!" "no, it's as designed!" dance has no place in an Agile project.

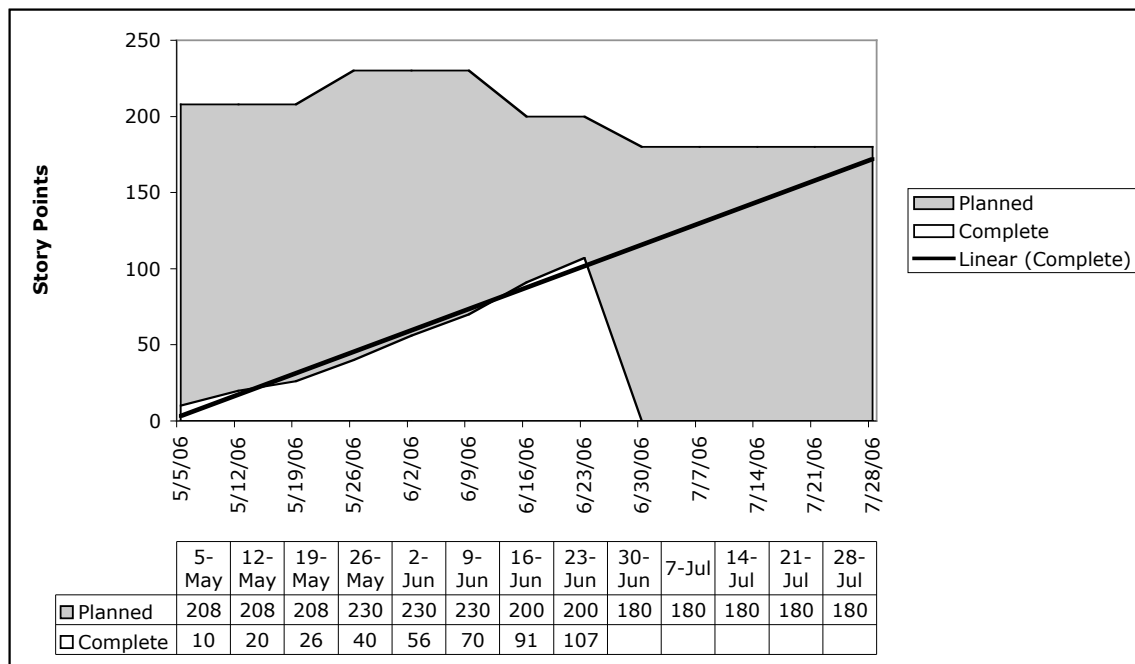
Release planning

An iteration is a machine that takes in descriptions of stories and spits out new business value. The same is true of the entire project.

Once a project's been underway for a while, an iteration should be extremely predictable. The team will estimate stories, you'll pick the stories, and you'll be confident that they'll be done at the end of the iteration. You'd like the entire project to be that predictable.

It won't be, for two reasons. (1) You can't break the entire product down into chunks that can be accurately estimated, especially if you try right at the beginning of the project. (2) By the time the project is over, the business will want something different anyway.

Nevertheless, you probably have a fixed budget of time, you (and your bosses) want to know what the business will get at the end of that time, and you don't want to know that only after the time's up. Here's a *burnup chart* that can be useful:³



The top line shows how many stories you plan to get done. The Planned line changes as stories are added, dropped, or re-estimated. As each iteration ends, the Complete line moves up by the number of stories completed. When the two lines cross, the project's done. More importantly, you can roughly extrapolate when the two lines will meet and adjust the content of the product to make it come in on time.

In conventional projects, it would be a crisis if it looked as if the product couldn't complete all its features by the deadline. That's because missing the deadline means many

³ I wish I knew how to get rid of that alarming dropoff of Completed down to zero, but I've never really learned Excel. This chart is a variation of one used by Wayne Allen <http://weblogs.asp.net/wallen/>

features will be partially finished but utterly unusable or too bug-ridden to deploy. Part of your job will be to avert any such crisis in three ways:

1. Insist that every iteration end in a working, shippable product with all completed stories *really* complete. That way, every euro spent on the project will have delivered real business value.
2. Ship or deploy the product as often as you can. Don't place those who funded the project in the position of judging it based on nothing but promises, demos, and a spreadsheet. Make sure they've also heard of the real value real users have already been getting from it.
3. Order development roughly in order of descending business value. That way, any stories that "slip off the end" will be the ones that were of the most marginal ROI anyway.

To learn more

Agile Estimating and Planning and *User Stories Applied*, both by Mike Cohn, cover those topics thoroughly. (Note: *User Stories Applied* has a nice longish example in the back—I suggest you read it first.)

Planning Extreme Programming, by Kent Beck and Martin Fowler, is a shorter book than Mike's. If you're short on time, you might read this instead. I don't think there's a reason to read it in addition.

Bill Wake has an article, "Twenty Ways to Split Stories", which is about how to split big stories into smaller ones. <<http://xp123.com/xplor/xp0512/index.shtml>>

Fit for Developing Software, by Rick Mugridge and Ward Cunningham, is a thorough introduction to the Fit testing tool I mentioned earlier.

There are two different styles for using Fit. One is to act something like a user, so the tests are written "do this, do that, check this". The other is to lay out business rules in a tabular form, as I did in the example for scheduling treatments. The first is the one people tend to use first, but we find them moving toward the second over time. Bill Wake's "Procedural and Declarative Tests" <<http://xp123.com/xplor/xp0503/index.shtml>> talks more about the difference. His "Acceptance Tests for a Query System" <<http://xp123.com/xplor/xp0105b/index.shtml>> works through a declarative test in some detail.

James Shore's "A Vision for Fit" is a good story about how he uses Fit. Our styles are very compatible. <<http://www.jamesshore.com/Blog/A-Vision-For-Fit.html>>

I've talked little about testers (as opposed to tests). The role of tester is less defined than your role or the programmers' role. Start off by thinking of testers as part of your support team. They can help you devise examples, especially examples of what could go wrong and how the product should handle it. Lisa Crispin's book, *Testing Extreme Programming*, takes this perspective, as does a shorter article "Extreme Rules of the Road" <<http://www.testing.com/agile/crispin-xp-article.pdf>>. For further notes on the testers

role, see my “A Roadmap for Testing on an Agile Project” <http://www.testing.com/writings/2004-roadmap.html> and a series of essays on “Agile Testing Directions” <http://www.testing.com/cgi-bin/blog/2004/05/26#directions-toc>.

I haven’t covered user interface design. You might do it, or you might have a specialist do it. My impression is that it’s more important to design the user’s navigation through the program than to design particular screens. *Software for Use*, by Constantine and Lockwood, talks about that.

“Nickieben Bourbaki is a Customer on an Agile Project. Boy Does He Have Problems...” is a list of problems and solutions some of us have seen.

<http://www.testing.com/cgi-bin/blog/2004/10/25#customer-help>

The *agile-customer-today* mailing list is a place for product directors to help each other. There are hardly any messages. <http://groups.yahoo.com/group/agile-customer-today>

The *agile-testing* mailing list is more active. It can help with questions about examples and tests. <http://groups.yahoo.com/group/agile-testing/>

Thanks to...

... Bill Wake for suggested reading, and to the product directors and other team members I’ve watched and spoken with. All the good ideas are theirs.