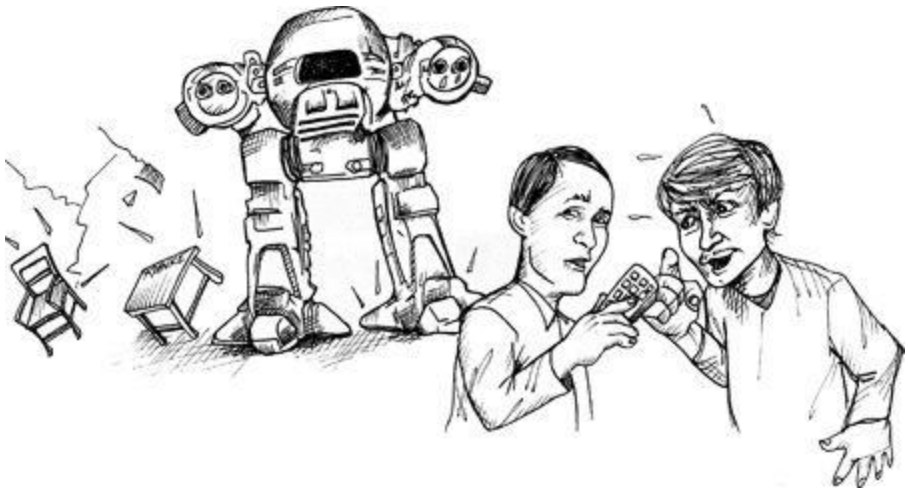
7. Acceptance Testing



The role of the professional developer is a communications role as well as a development role. Remember that garbage-in/garbage-out applies to programmers too, so professional programmers are careful to make sure that their communication with other members of the team, and the business, are accurate and healthy.

Communicating Requirements

One of the most common communication issues between programmers and business is the requirements. The business people state what they believe they need, and then the programmers build what they believe the business described. At least that's how it's supposed to work. In reality, the communication of requirements is extremely difficult, and the process is fraught with error.

In 1979, while working at Teradyne, I had a visit from Tom, the manager of installation and field service. He asked me to show him how to use the ED-402 text editor to create a simple trouble-ticket system.

ED-402 was a proprietary editor written for the M365 computer, which was Teradyne's PDP-8 clone. As a text editor it was very powerful. It had a built-in scripting language that we used for all kinds of simple text applications.

Tom was not a programmer. But the application he had in mind was simple, so he thought I could teach him quickly and then he could write the application himself. In my naivete I thought the same thing. After all, the scripting language was little more than a macro language for the editing commands, with very rudimentary decision and looping constructs.

So we sat down together and I asked him what he wanted his application to do. He started with the initial entry screen. I showed him how to create a text file that would hold the script statements and how to type the symbolic representation of the edit commands into that script. But when I looked into his eyes, there was nothing looking back. My explanation simply made no sense to him at all.

This was the first time I had encountered this. For me it was a simple thing to represent editor commands symbolically. For example, to represent a control-B command (the command that puts the cursor at the beginning of the current line) you simply typed ^B into the script file. But this made no sense to Tom. He couldn't make the leap from editing a file to editing a file that edited a file.

Tom wasn't dumb. I think he simply realized that this was going to be a lot more involved than he initially thought, and he didn't want to invest the time and mental energy necessary to learn something so hideously convoluted as using an editor to command an editor.

So bit by bit I found myself implementing this application while he sat there and watched. Within the first twenty minutes it was clear that his emphasis had changed from learning how to do it himself to making sure that what *I* did was what *he* wanted.

It took us an entire day. He would describe a feature and I would implement it as he watched. The cycle time was five minutes or less, so there was no reason for him to get up and do anything else. He'd ask me to do X, and within five minutes I had X working.

Often he would draw what he wanted on a scrap of paper. Some of the things he wanted were hard to do in ED-402, so I'd propose something else. We'd eventually agree on something that would work, and then I'd make it work.

But then we'd try it and he'd change his mind. He'd say something like, "Yeah, that just doesn't have the flow I'm looking for. Let's try it a

different way.”

Hour after hour we fiddled and poked and prodded that application into shape. We tried one thing, then another, and then another. It became very clear to me that *he* was the sculptor, and I was the tool he was wielding.

In the end, he got the application he was looking for but had no idea how to go about building the next one for himself. I, on the other hand, learned a powerful lesson about how customers actually discover what they need. I learned that their vision of the features does not often survive actual contact with the computer.

Premature Precision

Both business and programmers are tempted to fall into the trap of premature precision. Business people want to know exactly what they are going to get before they authorize a project. Developers want to know exactly what they are supposed to deliver before they estimate the project. Both sides want a precision that simply cannot be achieved, and are often willing to waste a fortune trying to attain it.

The Uncertainty Principle

The problem is that things appear different on paper than they do in a working system. When the business actually sees what they specified running in a system, they realize that it wasn’t what they wanted at all. Once they see the requirement actually running, they have a better idea of what they really want—and it’s usually not what they are seeing.

There’s a kind of observer effect, or uncertainty principle, in play. When you demonstrate a feature to the business, it gives them more information than they had before, and that new information impacts how they see the whole system.

In the end, the more precise you make your requirements, the less relevant they become as the system is implemented.

Estimation Anxiety

Developers, too, can get caught in the precision trap. They know they must estimate the system and often think that this requires precision. It doesn’t.

First, even with perfect information your estimates will have a huge variance. Second, the uncertainty principle makes hash out of early precision. The requirements *will* change making that precision moot.

Professional developers understand that estimates can, and should, be made based on low precision requirements, and recognize that those estimates *are estimates*. To reinforce this, professional developers always include error bars with their estimates so that the business understands the uncertainty. (See Chapter 10, "Estimation.")

Late Ambiguity

The solution to premature precision is to defer precision as long as possible. Professional developers don't flesh out a requirement until they are just about to develop it. However, that can lead to another malady: late ambiguity.

Often stakeholders disagree. When they do, they may find it easier to *wordsmith* their way around the disagreement rather than solve it. They will find some way of phrasing the requirement that they can all agree with, without actually resolving the dispute. I once heard Tom DeMarco say, "An ambiguity in a requirements document represents an argument amongst the stakeholders."[1]

Of course, it doesn't take an argument or a disagreement to create ambiguity. Sometimes the stakeholders simply assume that their readers know what they mean.

It may be perfectly clear to them in their context, but mean something completely different to the programmer who reads it. This kind of contextual ambiguity can also occur when customers and programmers are speaking face to face.

> Sam (stakeholder): "OK, now these log files need to be backed up."
>
> Paula: "OK, how often?"
>
> Sam: "Daily."
>
> Paula: "Right. And where do you want it saved?"
>
> Sam: "What do you mean?"
>
> Paula: "Do you want me to save it a particular sub-directory?"

Sam: “Yes, that’d be good.”

Paula: “What shall we call it?”

Sam: “How about ‘backup’?”

Paula: “Sure, that’d be fine. So we’ll write the log file into the backup directory every day. What time?”

Sam: “Every day.”

Paula: “No, I mean what time of day do you want it written?”

Sam: “Any time.”

Paula: “Noon?”

Sam: “No, not during trading hours. Midnight would be better.”

Paula: “OK, midnight then.”

Sam: “Great, thanks!”

Paula: “Always a pleasure.”

Later, Paula is telling her teammate Peter about the task.

Paula: “OK, we need to copy the log file into a sub-directory named backup every night at midnight.”

Peter: “OK, what file name should we use?”

Paula: “log.backup ought to do it.”

Peter: “You got it.”

In a different office, Sam is on the phone with his customer.

Sam: “Yes, yes, the log files will be saved.”

Carl: “OK, it’s vital that we never lose any logs. We need to go back through all those log files, even months or years later, whenever there’s an outage, event, or dispute.”

Sam: “Don’t worry, I just spoke to Paula. She’ll be saving the logs into a directory named backup every night at midnight.”

Carl: “OK, that sounds good.”

I presume you’ve detected the ambiguity. The customer expects all log files to be saved, and Paula simply thought they wanted to save last night’s log file. When the customer goes looking for months’ worth of log file backups, they’ll just find last night’s.

In this case both Paula and Sam dropped the ball. It is the responsibility of professional developers (and stakeholders) to make sure that all ambiguity is removed from the requirements.

This is *hard*, and there's only one way I know how to do it.

Acceptance Tests

The term *acceptance test* is overloaded and overused. Some folks assume that these are the tests that users execute before they accept a release. Other folks think these are QA tests. In this chapter we will define acceptance tests as tests written by a collaboration of the stakeholders and the programmers *in order to define when a requirement is done*.

The Definition of "Done"

One of the most common ambiguities we face as software professionals is the ambiguity of "done." When a developer says he's done with a task, what does that mean? Is the developer done in the sense that he's ready to deploy the feature with full confidence? Or does he mean that he's ready for QA? Or perhaps he's done writing it and has gotten it to run once but hasn't really tested it yet.

I have worked with teams who had a different definition for the words "done" and "complete." One particular team used the terms "done" and "done-done."

Professional developers have a single definition of done: Done means *done*. Done means all code written, all tests pass, QA and the stakeholders have accepted. Done.

But how can you get this level of done-ness and still make quick progress from iteration to iteration? You create a set of automated tests that, when they pass, meet all of the above criteria! When the acceptance tests for your feature pass, you are *done*.

Professional developers drive the definition of their requirements all the way to automated acceptance tests. They work with stakeholder's and QA to ensure that these automated tests are a complete specification of done.

> Sam: "OK, now these log files need to be backed up."
>
> Paula: "OK, how often?"
>
> Sam: "Daily."

Paula: “Right. And where do you want it saved?”

Sam: “What do you mean?”

Paula: “Do you want me to save it a particular sub-directory?”

Sam: “Yes, that’d be good.”

Paula: “What shall we call it?”

Sam: “How about ‘backup’”?

Tom (tester): “Wait, backup is too common a name. What are you really storing in this directory?”

Sam: “The backups.”

Tom: “Backups of what?”

Sam: “The log files.”

Paula: “But there’s only one log file.”

Sam: “No, there are many. One for each day.”

Tom: “You mean that there is one *active* log file, and many log file backups?”

Sam: “Of course.”

Paula: “Oh! I thought you just wanted a temporary backup.”

Sam: “No, the customer wants to keep them all forever.”

Paula: “That’s a new one on me. OK, glad we cleared that up.”

Tom: “So the name of the sub-directory should tell us exactly what’s in it.”

Sam: “It’s got all the old inactive logs.”

Tom: “So let’s call it `old_inactive_logs`.”

Sam: “Great.”

Tom: “So when does this directory get created?”

Sam: “Huh?”

Paula: “We should create the directory when the system starts, but only if the directory doesn’t already exist.”

Tom: “OK, there’s our first test. I’ll need to start up the system and see if the `old_inactive_logs` directory is created. Then I’ll add a file to that directory. Then I’ll shut down, and

start again, and make sure both the directory and the file are still there."

Paula: "That test is going to take you a long time to run. System start-up is already 20 seconds, and growing. Besides, I really don't want to have to build the whole system every time I run the acceptance tests."

Tom: "What do you suggest?"

Paula: "We'll create a `SystemStarter` class. The main program will load this starter with a group of `StartupCommand` objects, which will follow the COMMAND pattern. Then during system start-up the `SystemStarter` will simply tell all the `StartupCommand` objects to run. One of those `StartupCommand` derivatives will create the `old_inactive_logs` directory, but only if it doesn't already exist."

Tom: "Oh, OK, then all I need to test is that `StartupCommand` derivative.

I can write a simple FITNESSE test for that."

Tom goes to the board.

"The first part will look something like this":

```
  given                the                command
LogFileDirectoryStartupCommand
given that the old_inactive_logs directory does
not exist


when the command is executed
then the old_inactive_logs directory should exist
and it should be empty
```

"The second part will look like this":

```
  given                the                command
LogFileDirectoryStartupCommand
given that the old_inactive_logs directory exists
and that it contains a file named x
When the command is executed
Then the old_inactive_logs directory should still
exist
and it should still contain a file named x
```

Paula: "Yeah, that should cover it."

Sam: “Wow, is all that really necessary?”

Paula: “Sam, which of these two statements isn’t important enough to specify?”

Sam: “I just mean that it looks like a lot of work to think up and write all these tests.”

Tom: “It is, but it’s no more work than writing a manual test plan. And it’s *much* more work to repeatedly execute a manual test.”

Communication

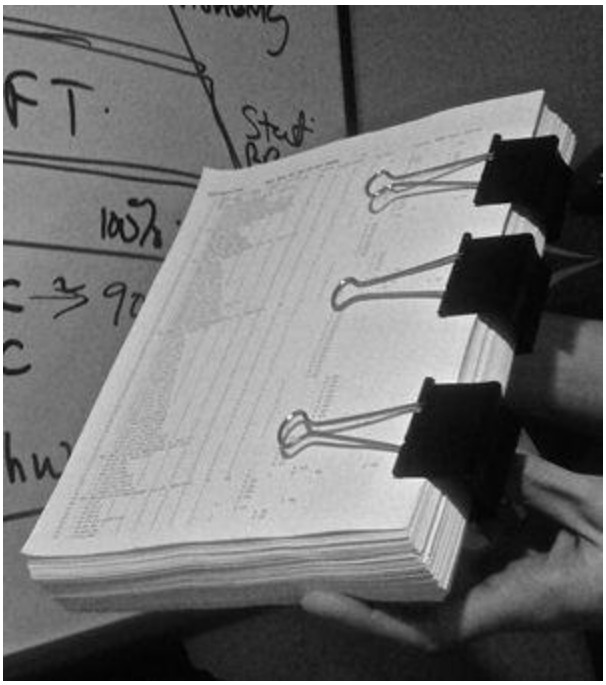
The purpose of acceptance tests is communication, clarity, and precision. By agreeing to them, the developers, stakeholders, and testers all understand what the plan for the system behavior is. Achieving this kind of clarity is the responsibility of all parties. Professional developers make it their responsibility to work with stakeholders and testers to ensure that all parties know what is about to be built.

Automation

Acceptance tests should *always* be automated. There is a place for manual testing elsewhere in the software lifecycle, but *these* kinds of tests should never be manual. The reason is simple: cost.

Consider the image in Figure 7-1. The hands you see there belong to the QA manager of a large Internet company. The document he is holding is the *table of contents* for his *manual* test plan. He has an army of manual testers in off-shore locations that execute this plan once every six weeks. It costs him over a million dollars every time. He’s holding it out for me because he’s just come back from a meeting in which his manager has told him that they need to cut his budget by 50%. His question to me is, “Which half of these tests should I not run?”

Figure 7-1. Manual test plan



To call this a disaster would be a gross understatement. The cost of running the manual test plan is so enormous that they have decided to sacrifice it and simply live with the fact that *they won't know if half of their product works!*

Professional developers do not let this kind of situation happen. The cost of automating acceptance tests is so small in comparison to the cost of executing manual test plans that it makes no economic sense to write scripts for humans to execute. Professional developers take responsibility for their part in ensuring that acceptance tests are automated.

There are many open-source and commercial tools that facilitate the automation of acceptance tests. FITNESSE, Cucumber, cuke4duke, robot framework, and Selenium, just to mention a few. All these tools allow you to specify automated tests in a form that nonprogrammers can read, understand, and even author.

Extra Work

Sam's point about work is understandable. It *does* look like a lot of extra work to write acceptance tests like this. But given Figure 7-1 we can see that it's not really extra work at all. Writing these tests is simply the work of specifying the system. Specifying at this level of detail is the only way we, as programmers, can know what "done" means. Specifying at this

level of detail is the only way that the stakeholders can ensure that the system they are paying for really does what they need. And specifying at this level of detail is the only way to successfully automate the tests. So don't look at these tests as extra work. Look at them as massive time and money savers. These tests will prevent you from implementing the wrong system and will allow you to *know* when you are done.

Who Writes Acceptance Tests, and When?

In an ideal world, the stakeholders and QA would collaborate to write these tests, and developers would review them for consistency. In the real world, stakeholders seldom have the time or inclination to dive into the required level of detail. So they often delegate the responsibility to business analysts, QA, or even developers. If it turns out that developers must write these tests, then take care that the developer who writes the test is not the same as the developer who implements the tested feature.

Typically business analysts write the "happy path" versions of the tests, because those tests describe the features that have business value. QA typically writes the "unhappy path" tests, the boundary conditions, exceptions, and corner cases. This is because QA's job is to help think about what can go wrong.

Following the principle of "late precision," acceptance tests should be written as late as possible, typically a few days before the feature is implemented. In Agile projects, the tests are written *after* the features have been selected for the next Iteration or Sprint.

The first few acceptance tests should be ready by the first day of the iteration. More should be completed each day until the midpoint of the iteration when all of them should be ready. If all the acceptance tests aren't ready by the midpoint of the iteration, then some developers will have to pitch in to finish them off. If this happens frequently, then more BAs and/or QAs should be added to the team.

The Developer's Role

Implementation work on a feature begins when the acceptance tests for that feature are ready. The developers execute the acceptance tests for the new feature and see how they fail. Then they work to connect the

acceptance test to the system, and then start making the test pass by implementing the desired feature.

Paula: “Peter, would you give me a hand with this story?”

Peter: “Sure, Paula, what’s up?”

Paula: “Here’s the acceptance test. As you can see, it’s failing.”

```
  given                the                command
LogFileDirectoryStartupCommand
given that the old_inactive_logs directory does
not exist
when the command is executed
then the old_inactive_logs directory should exist
and it should be empty
```

Peter: “Yeah, all red. None of the scenarios are written. Let me write the first one.”

```
  |scenario|given the command _|cmd|
|create command|@cmd|
```

Paula: “Do we already have a `createCommand` operation?”

Peter: “Yeah, it’s in the `CommandUtilitiesFixture` that I wrote last week.”

Paula: “OK, so let’s run the test now.”

Peter: (runs test). “Yeah, the first line is green, let’s move on to the next.”

Don’t worry too much about Scenarios and Fixtures. Those are just some of the plumbing you have to write to connect the tests to the system being tested.

Suffice it to say that the tools all provide some way to use pattern matching to recognize and parse the statements of the test, and then to call functions that feed the data in the test into the system being tested. The amount of effort is small, and the Scenarios and Fixtures are reusable across many different tests.

The point of all this is that it is the developer’s job to connect the acceptance tests to the system, and then to make those tests pass.

Test Negotiation and Passive Aggression

Test authors are human and make mistakes. Sometimes the tests as written don't make a lot of sense once you start implementing them. They might be too complicated. They might be awkward. They might contains silly assumptions. Or they might just be wrong. This can be very frustrating if you are the developer who has to make the test pass.

As a professional developer, it is your job to negotiate with the test author for a better test. What you should *never* do is take the passive-aggressive option and say to yourself, "Well, that's what the test says, so that's what I'm going to do."

Remember, as a professional it is your job to help your team create the best software they can. That means that everybody needs to watch out for errors and slip-ups, and work together to correct them.

Paula: "Tom, this test isn't quite right."

```
  ensure that the post operation finishes in 2
seconds.
```

Tom: "It looks OK to me. Our requirement is that users should not have to wait more than two seconds. What's the problem?"

Paula: "The problem is we can only make that guarantee in a statistical sense."

Tom: "Huh? That sounds like weasel words. The requirement is two seconds."

Paula: "Right, and we can achieve that 99.5% of the time."

Tom: "Paula, that's not the requirement."

Paula: "But it's reality. There's no way I can make the guarantee any other way."

Tom: "Sam's going to throw a fit."

Paula: "No, actually, I've already spoken to him about it. He's fine as long as the *normal* user experience is two seconds or less."

Tom: "OK, so how do I write this test? I can't just say that the post operation *usually* finishes in two seconds."

Paula: "You say it statistically."

Tom: “You mean you want me to run a thousand post operation and make sure no more than five are more than two seconds? That’s absurd.”

Paula: “No, that would take the better part of an hour to run. How about this?”

```
  execute 15 post transactions and accumulate times.
ensure that the Z score for 2 seconds is at least 2.57
```

Tom: “Whoa, what’s a Z score?”

Paula: “Just a bit of statistics. Here, how about this?”

```
  execute 15 post transactions and accumulate times.
ensure odds are 99.5% that time will be less than 2 seconds.
```

Tom: “Yeah, that’s readable, sort of, but can I trust the math behind the scenes?”

Paula: “I’ll make sure to show all the intermediate calculations in the test report so that you can check the math if you have any doubts.”

Tom: “OK, that works for me.”

Acceptance Tests and Unit Tests

Acceptance tests are not *unit* tests. Unit tests are written *by* programmers *for* programmers. They are formal design documents that describe the lowest level structure and behavior of the code. The audience is programmers, not business.

Acceptance tests are written *by* the business *for* the business (even when you, the developer, end up writing them). They are formal requirements documents that specify how the system should behave from the business’ point of view. The audience is the business *and* the programmers.

It can be tempting to try to eliminate “extra work” by assuming that the two kinds of tests are redundant. Although it is true that unit and acceptance tests often test the same things, they are not redundant at all.

First, although they may test the same things, they do so through different mechanisms and pathways. Unit tests dig into the guts of the system making calls to methods in particular classes. Acceptance tests invoke the system much farther out, at the API or sometimes even UI level. So the execution pathways that these tests take are very different.

But the real reason these tests aren't redundant is that their primary function *is not testing*. The fact that they are tests is incidental. Unit tests and acceptance tests are documents first, and tests second. Their primary purpose is to formally document the design, structure, and behavior of the system. The fact that they automatically verify the design, structure, and behavior that they specify is wildly useful, but the specification is their true purpose.

GUIs and Other Complications

It is hard to specify GUIs up front. It can be done, but it is seldom done well. The reason is that the aesthetics are subjective and therefore volatile. People want to *fiddle* with GUIs. They want to massage and manipulate them. They want to try different fonts, colors, page-layouts, and workflows. GUIs are constantly in flux.

This makes it challenging to write acceptance tests for GUIs. The trick is to design the system so that you can treat the GUI as though it were an API rather than a set of buttons, sliders, grids, and menus. This may sound strange, but it's really just good design.

There is a design principle called the Single Responsibility Principle (SRP). This principle states that you should separate those things that change for different reasons, and group together those things that change for the same reasons. GUIs are no exception.

The layout, format, and workflow of the GUI will change for aesthetic and efficiency reasons, but the underlying capability of the GUI will remain the same despite these changes. Therefore, when writing acceptance tests for a GUI you take advantage of the underlying abstractions that don't change very frequently.

For example, there may be several buttons on a page. Rather than creating tests that click on those buttons based on their positions on the page, you may be able to click on them based on their names. Better yet, perhaps they each have a unique `ID` that you can use. It is much better to

write a test that selects the button whose `ID` is `ok_button` than it is to select the button in column 3 of row 4 of the control grid.

Testing through the Right Interface

Better still is to write tests that invoke the features of the underlying system through a real API rather than through the GUI. This API should be the same API used by the GUI. This is nothing new. Design experts have been telling us for decades to separate our GUIs from our business rules.

Testing through the GUI is always problematic unless you are testing *just* the GUI. The reason is that the GUI is likely to change, making the tests very fragile. When every GUI change breaks a thousand tests, you are either going to start throwing the tests away or you are going to stop changing the GUI. Neither of those are good options. So write your business rule tests to go through an API just below the GUI.

Some acceptance tests specify the behavior of the GUI itself. These tests *must* go through the GUI. However, these tests do not test business rules and therefore don't require the business rules to be connected to the GUI. Therefore, it is a good idea to decouple the GUI and the business rules and replace the business rules with stubs while testing the GUI itself.

Keep the GUI tests to a minimum. They are fragile, because the GUI is volatile. The more GUI tests you have the less likely you are to keep them.

Continuous Integration

Make sure that all your unit tests and acceptance tests are run several times per day in a *continuous integration* system. This system should be triggered by your source code control system. Every time someone commits a module, the CI system should kick off a build, and then run all the tests in the system. The results of that run should be emailed to everyone on the team.

Stop the Presses

It is very important to keep the CI tests running at all times. They should never fail. If they fail, then the whole team should stop what they are doing and focus on getting the broken tests to pass again. A broken build in the CI system should be viewed as an emergency, a "stop the presses" event.

I have consulted for teams that failed to take broken tests seriously. They were “too busy” to fix the broken tests so they set them aside, promising to fix them later. In one case the team actually took the broken tests out of the build because it was so inconvenient to see them fail. Later, after releasing to the customer, they realized that they had forgotten to put those tests back into the build. They learned this because an angry customer was calling them with bug reports.

Conclusion

Communication about details is hard. This is especially true for programmers and stakeholders communicating about the details of an application. It is too easy for each party to wave their hands and *assume* that the other party understands. All too often both parties agree that they understand and leave with completely different ideas.

The only way I know of to effectively eliminate communication errors between programmers and stakeholders is to write automated acceptance tests. These tests are so formal that they execute. They are completely unambiguous, and they cannot get out of sync with the application. They are the perfect requirements document.