Gojko Adzic

Specification by Example

How successful teams deliver the *right* software

MANNING

Table of Contents

10

Validating frequently

> Stray too far out of your lane and your attention is immediately riveted by a loud, vibrating baloop baloop baloop.
>
> —David Haldane[1]

In the 1950s, the California Department of Transportation had a problem with motorway lane markers. The lines were wearing out, and someone had to repaint them every season. This was costly, caused disruption to traffic, and was dangerous for the people charged with that task.

Dr. Elbert Dysart Botts worked on solving that problem and experimented with more-reflective paint, but this proved to be a dead end. Thinking outside the box, he invented raised lane markers, called Botts' Dots. Botts' Dots were visible by day or night, regardless of the weather. They didn't wear out as easily as painted lane markers. Instead of relying just on drivers' sense of sight, Botts' Dots cause a tactile vibration and audible rumbling when drivers move across designated travel lanes. This feedback proved to be one of the most important safety features on highways, alerting inattentive drivers to potential danger when they drift from their lane.

Botts' Dots were introduced to software development as one of the original 12 Extreme Programming practices, called *continuous integration* (CI). Continuous integration alerts inattentive software teams when they start to drift from a product that can be built and packaged. A dedicated continuous integration system frequently builds the product and runs tests to ensure that the system doesn't work only on a developer's machine. By flagging potential problems quickly, this practice allows us to stay in the middle of the lane and take small and cheap corrective action when needed. Continuous integration ensures that once the product is built right, it stays right.

[1] http://articles.latimes.com/1997-03-07/local/me-35781_1_botts-dots

The same principles apply to building the right product. Once the right product is built, we want to ensure that it stays right. If it drifts from the designated direction, we can solve the problem much more easily and cheaply if we know about it quickly and don't let problems accumulate. We can frequently validate executable specifications. A continuous-build server[2] can frequently check all the specifications and ensure that the system still satisfies them.

Continuous integration is a well-documented software practice, and many other authors have already done a good job of explaining it in detail. I don't wish to repeat how to set up continuous-build and integration systems in general, but some particular challenges for frequent validation of executable specifications are important for the topic of this book.

Many teams who used Specification by Example to extend existing systems found that executable specifications have to run against a real database with realistic data, external services, or a fully deployed website. Functional acceptance tests check the functionality across many components, and if the system isn't built up front to be testable, such checks often require the entire system to be integrated and deployed. This causes three groups of problems for frequent validation in addition to those usual for continuous integration with just technical (unit) tests:

- Unreliability caused by environmental dependencies—Unit tests are largely independent of the test environment, but executable specifications might depend heavily on the rest of the ecosystem they run in. Environment issues can cause the tests to fail even if the programming language code is correct. To gain confidence in acceptance test results, we have to solve or mitigate these environmental problems and make the test execution reliable.
- Slower feedback–Functional acceptance tests on brownfield projects will often run an order of magnitude slower than unit tests. I'd consider a unit test pack slow if it runs for several minutes. Anything close to 10 minutes would be definitely too slow, and I'd seriously start investigating how to make it run faster. On the other hand, I've seen acceptance test packs that had to run for several hours and couldn't be optimized without reducing confidence in the results. Such slow overall feedback requires us to come up with a solution to get fast feedback from selected parts of the system on demand.
- Managing failed tests—A large number of coarse-grained functional tests that depend on many moving parts required some teams, especially when they started implementing Specification by Example, to manage failing tests instead of fixing them straight away.

[2] Software that automatically builds, packages, and executes tests when anyone changes any code in the version control system. If you've never heard of one, google CruiseControl, Hudson or TeamCity.

In this chapter, I explain how teams from my research handled these three problems.

Reducing unreliability

An unreliable validation process can undermine a team's confidence in a product and the process of Specification by Example. Investigating intermittent failures that aren't caused by real problems is a huge waste of time. If that happens often, developers will have an excuse not to look at validation problems at all. That will allow real issues to pass undetected, defeating the whole point of continuous validation.

Legacy projects rarely support automated functional testing easily, so executable specifications might need to be automated through unreliable user interfaces or suffer from nondeterminism caused by asynchronous processes. This is especially problematic when developers need convincing to participate in the process and see it only as an improvement on functional testing (in other words, not their problem).

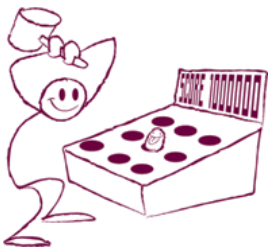
Clare McLennan faced this issue with her team. "Developers didn't care about tests because they weren't stable, but we needed their knowledge to make them stable," she said. This presented a chicken-and-egg problem for her team. To get the developers to participate, she had to show them the value of executable specifications. But to do that, the executable specifications had to be reliable, which required developers to change the system design and make it easier to plug in automated tests.

To get the long-term benefits of Specification by Example, many teams had to invest significant effort into making their validation processes reliable. In this section I present some good ideas for that.

Find the most annoying thing, fix it, and repeat

When: Working on a system with bad automated test support

One of the most important things to understand about making the system more reliable for automated testing is that this won't happen overnight. Legacy systems aren't easy to change; otherwise, they wouldn't be legacy. When something was built without a testable design for years, it won't suddenly become clean and testable.



Introducing too many major changes quickly would destabilize the system, especially if we still don't have good functional test coverage. It would also severely interrupt the flow of development.

Instead of trying to solve a problem with one big hit, a more useful strategy is to make many small changes iteratively.

For example, McLennan's team realized that slow test data processing was causing tests to time out. Their database administrator improved the database performance, which

led them to discover that some tests were starting before the updated test data was processed, so the system was serving old data. They introduced messages to tell them when the latest data from the database was being served, so that they could reliably start the tests after that and avoid false negatives. When that source of entropy was eliminated, they discovered that HTTP cookie expiry was causing problems. They introduced the concept of business time, so that they could change the time the system thinks it's using. (See the "Introduce business time" section later in this chapter.)

As a strategy to achieve stability under automated testing, McLennan advises an incremental approach:

> Find the most annoying thing and fix it, then something else will pop up, and after that something else will pop up. Eventually, if you keep doing this, you will create a stable system that will be really useful.

Improving the stability iteratively is a good way to build a reliable validation process without interrupting the delivery flow so much. This approach also enables us to learn and adapt while we're making the system more testable.

Identify unstable tests using CI test history

When: Retrofitting automated testing into a legacy system

On a legacy system that's not susceptible to automated testing, it's often hard to decide where to start with iterative cleanup because there are so many causes of instability. A good strategy for this is to look at the test execution history. Most continuous-build systems today have a feature that tracks test results over a time.

Once executable specifications are plugged into a continuous-build system, the test run history will allow us to see which tests or groups of tests are the most unstable.

I completely overlooked this feature for years because I was mostly working on greenfield projects built up to be testable up front or systems where relatively small changes introduced stability. In such projects, tracking the history of test executions is useless: The tests pass almost all the time and are fixed as soon as they fail. The first time I tried to retrofit automated testing into a system that suffered from occasional timeouts, networking issues, database problems, and inconsistent processing, seeing the test history helped me focus my efforts to increase stability. That showed me which groups of tests were failing the most often so that I could fix them first.

Set up a dedicated continuous validation environment

If your application needs to be deployed and running for functional testing, the first step to reproducibility is to secure a dedicated environment for deployment.

Continuous validation has to work in a reproducible way to be reliable. In some larger organizations it's harder to get a new set of machines than it is to hire a known poisoner as a chef in the company cafeteria, but fighting for better equipment is well worth it. Many teams tried to use the same environment for demonstrating features to business users, manual testing, and continuous validation. This regularly caused data consistency issues.

Without a dedicated environment, it's hard to know whether a test failed because there's a bug, whether someone changed something on the test environment, or whether the system is just unstable. A dedicated environment eliminates unplanned changes and mitigates the risk of unstable environments.

Employ fully automated deployment

Once we have a dedicated environment, we want to ensure that the software is deployed in a reproducible way. Unreliable deployment is the second most common cause of test result instability. For many legacy systems, deployment is a process done overnight that involves several people, lots of coffee, and ideally a magic wand. When we have to deploy once every year, this is acceptable. When we have to deploy every two weeks, it becomes a major headache. For continuous validation, we might need to deploy several times a day, and magic-aided manual deployment is completely unacceptable.

Without a fully automated deployment that can reliably upgrade a system, we'll frequently get into situations where many tests suddenly start failing and someone has to spend hours troubleshooting to find the culprit, only to hear "but it works on my machine" from the back of the room.

Fully automated deployment will ensure that there's a single standard procedure for upgrading. It will also ensure that all the developers have the same system layout as the test environments.

This eliminates the dependency of executable specifications on a particular environment and makes continuous validation much more reliable. It also makes problems easier to troubleshoot, because developers can use any environment to reproduce problems.

For this to work, it has to be fully automated. No manual intervention should be required—or allowed—at all. (Note that I'm talking about a fully automated deployment that can be executed on demand, not necessarily firing off automatically as well.) Installers that require you to poke around an administration console, half-automated manual scripts, and things like that don't count as fully automated. In particular, this includes automated database deployments.

I've seen many teams who claim to have automated deployment, only to find out that someone has to run database scripts manually afterward.

Fully automated deployment brings other benefits as well, such as being able to upgrade production systems more easily. This will save you a lot of time long term. Frequent deployment is a good practice regardless of Specification by Example.

Create simpler test doubles for external systems

When: Working with external reference data sources

Many teams had problems with external reference data sources or external systems that participated in their business workflow. (By *external* I mean outside the scope of a team, not necessarily belonging to a different organization.) In large enterprises with complex networks of systems, a team might work on only one part of the workflow, and its test system will talk to the test systems of other teams. The problem is that the other teams have to do their own work and testing so their test servers might not be always available, reliable, or correct.

Create a separate fake data source that simulates the interaction with the real system.

Rob Park's team at a large U.S. insurance provider was building a system that looked up reference policy data on an external auto policy server. If the auto policy server went down, all their executable specifications would start failing. For functional testing, they used an alternative version of the external service. The simpler version read the data from a file on a local disk.

This allowed Park's team to test their system even when the auto policy server was offline. Creating a separate reference data source also gave the team full control of the reference data. Expired policies wouldn't be served by the real system, so tests that depended on a policy that had expired would start failing.

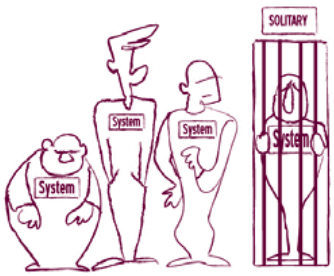
The simpler version of the reference data source served everything from the configuration file, which avoided the temporal issues. They kept the data in an XML file that was checked into a version control system, so that they could easily track changes and package the correct version of the test data with the correct version of the code. This would be impossible with an external system. A local service that reads from a file is also faster than the external system, speeding up the overall feedback.

A risk with test doubles is that the real system will evolve over time, and the double will no longer reflect the realistic functionality. To avoid that, be sure to check periodically whether the double still does what the original system is supposed to do. This is particularly important when the double is representing a third-party system over which you have no control.

Selectively isolate external systems

When: External systems participate in work

Isolating a system completely isn't always a good idea. When a system participates in a larger workflow where the external systems provide more than just reference data, a test double would have to start implementing parts of the real functionality of external systems. This brings significant overhead in development and more maintenance problems.



Ian Cooper's team at Beazley took an interesting pragmatic approach to solve this problem. They selectively turned off access to some services based on the goal of each executable specification. This made their tests significantly faster but still involved the minimal set of real external systems in each test. The solution didn't completely protect them from external influences, but it made troubleshooting a lot easier. If a test failed, it was clear which external dependency might have influenced it.

Selectively isolating some external services can make tests faster and troubleshooting easier.

Try multistage validation

When: Large/multisite groups

With legacy systems, running all the executable specifications often takes longer than the average time between two committed changes to the underlying source code. Because of that, it might be hard to associate a problem with a particular change that caused it.

With larger groups of teams, especially if they're spread across several sites, this can cause problems to accumulate. If one team breaks the database, the other teams won't be able to validate their changes until the problem gets fixed. It might take a several hours to find out that there's a problem, determine what it is, fix it, and rerun the tests to confirm that it's fixed. A broken build will always be someone else's problem, and very soon the continuous-validation test pack will always be broken. At that point, we might as well just stop running the tests.

Employ multistage validation. Each team should have an isolated continuous-validation environment; changes should be first tested there.

First integrate a change with the other changes from the same team. If the tests pass there, the push changes to the central continuous-validation environment, where they're integrated with all the changes from the other teams.

This approach prevents problems of one team influencing the other teams in most cases. Even if the central environment is broken and someone is fixing it, individual teams can still use their own environments to validate changes.

Depending on how long all executable specifications take to run, we can execute the full test pack in both environments or just run a representative subset of tests in the one of the environments to provide a quick smoke test. The Global Talent Management Team at Ultimate Software, for example, runs most of their tests just in the local team environments. Slower tests don't run on the central environment in order to provide quick feedback.

Execute tests in transactions

When: Executable specifications modify reference data

Database transactions can provide isolation from outside influences.

Transactions can prevent our process from influencing other processes running at the same time and make tests more reproducible.

If we create a user during our test, the test might fail the next time we run it because of a unique constraint on the username in the database. If we run that test inside a transaction and roll back at the end of the test, the user won't be stored, so the two test executions will be independent.

This is a good practice in many cases, but it might not work with some transactional contexts (for example, if database constraint checks are deferred until a transaction commit or in the case of nested autonomous transactions). Such advanced transactional topics are outside the scope of this book.

In any case, keep transaction control outside the specifications. Database transaction control is a crosscutting concern that's best implemented in the automation layer, not in the description of executable specifications.

Run quick checks for reference data

When: Data-driven systems

In data-driven systems, executable specifications depend extensively on reference data. Changes to the reference data, such as workflow configuration, might break the test even if the functionality is correct. Such problems are difficult to troubleshoot.

Set up a completely separate group of tests to verify that the reference data is still as we expect it to be.

Such tests can run quickly before executable specifications. If that test pack fails, there's no point in running the others. These tests will also pinpoint reference data problems and allow us to fix them quickly.

Wait for events, not for elapsed time

Asynchronous processes seem to be one of the most problematic areas for executable specifications. Even when parts of a process execute in the background, on different machines, or get delayed for several hours, business users see the whole process as a single sequence of events. At Songkick, this was one of the key challenges to overcome for a successful implementation of Specification by Example. Phil Cowans says:

> "Asynchronous processing has been a real headache for us. We do a lot of background processing that is asynchronous for performance reasons, and we ran into a lot of problems because the tests work instantly. The background processing hadn't happened by the time the test moved to the next step."

Reliably validating asynchronous processes requires some planning and careful design in the automation layer (and often in the production system). A common mistake with asynchronous systems is to wait a specific time for something to happen. A symptom of this is a test step such as "Wait 10 seconds." This is bad for several reasons.

Such tests might fail even when the functionality works correctly but the continuous validation environment is under a heavy load. Running these tests on a different environment might require more time, so they become dependent on a particular deployment. When a continuous validation environment is much more powerful than the

machines that the developers have, the developers won't be able to validate changes on their systems with the same timeout configuration. Many teams set high timeouts on tests to make them more resilient to environment changes. Such tests then delay feedback unnecessarily.

For example, if a test unconditionally waits 1 minute for a process to end, but it finishes in just 10 seconds, we delay the feedback unnecessarily for 50 seconds. Small delays might not be an issue for individual tests, but they accumulate in test packs. With 20 such tests, we delay the feedback for the entire test pack for more than 15 minutes, which makes a lot of difference.

Wait for an event to happen, not for a set period of time to elapse. This will make tests much more reliable and not delay the feedback any longer than required.

Whenever possible, implement such tests to block on a message queue or poll a database or a service in the background frequently to check whether a process has finished.

Make asynchronous processing optional

When: Greenfield projects

When building the system from the ground up, we have the option to design it to support easier testing. Depending on the configuration, the system can either queue a message to process the transaction in the background or directly execute it. We can then configure the continuous validation environment to run all processes synchronously.

One good way to deal with test stability is to make asynchronous processing optional.

This approach makes executable specifications run much more reliably and quickly. But it means that functional acceptance tests don't check the system end to end. If you turn off asynchronous processing for functional tests, remember to write additional technical tests to verify that asynchronous processing works. Although this might sound like doubling the work, it isn't. Technical tests can be short and focused just on the technical execution, not verifying the business functionality (which will be separately checked by functional tests).

For some good technical options on automating validation of asynchronous processes, see *Growing Object Oriented Software, Guided by Tests.*[3]

[3] Steve Freeman and Nat Pryce, *Growing Object-Oriented Software, Guided by Tests* (Addison-Wesley Professional, 2009).

Don't use executable specifications as end-to-end validations

When: Brownfield projects

Many teams, especially those working with existing legacy systems, used executable specifications as both functional tests and end-to-end integration tests. This gave them more confidence that the software worked correctly overall but made the feedback a lot slower and increased the maintenance costs for tests significantly.

The problem with this approach is that too many things get tested at the same time. Such end-to-end tests check the business logic of the process, the integration with the technical infrastructure, and that all the components can talk to each other. Many moving parts mean that a change in any of them will break the test. That also means that we have to run through the entire process end to end to validate every single case, even though we could possibly check what we want in only a part of the flow.

Don't test too many things at the same time with one big executable specification.

Most asynchronous processes consist of several steps that each have some business logic, which needs to be defined using executable specifications. Most of them also include purely technical tasks, such as dispatching to a queue or saving to a database. Instead of checking everything with one big specification that takes long to execute, think about the following when you implement such a process:

- Clearly separate the business logic from the infrastructure code (for example, pushing to a queue, writing to the database).
- Specify and test the business logic in each of the steps separately, possibly isolating the infrastructure, as described earlier in this chapter. This makes it much easier to write and execute tests. The tests can run synchronously, and they don't need to talk to a real database or a real queue. These tests give you confidence that you've implemented the right business logic.
- Implement technical integration tests for the infrastructure code, queue, or database implementations of repositories. These tests can be fairly simple because they don't have to validate complicated business logic. They give you confidence that you are using the infrastructure correctly.

Have one end-to-end integration test that verifies that all the components talk to each other correctly. This can execute a simple business scenario that touches all the components, blocks on a queue waiting for a response, or polls the database to check for the result. This gives you confidence that the configuration works. If you use high-level examples, as suggested in the "Don't look only at the lowest level" section in chapter 5, they're good candidates for this test.

Moving down this list from business logic over infrastructure to end-to-end tests, I'd normally expect to see the number of tests decrease and time to execute individual tests increase significantly. By isolating complicated business logic tests from the infrastructure, we improve reliability.

Getting feedback faster

Most teams found that running the all their executable specifications after every change of the system isn't feasible. A large number of checks (especially if they can only be executed against a website, database, or external services) make the feedback from a full test run too slowly. In order to support development efficiently and facilitate change, most teams changed their continuous validation systems to provide feedback in several stages, so that they get the most important information quickly. Here are some of the strategies that the teams I interviewed used to keep their feedback loop shorter.

Introduce business time

When: Working with temporal constraints

Temporal constraints are a common reason for slow feedback in automated functional testing. End-of-day jobs might normally run at midnight, and any tests that depend on them would produce slow feedback, because we'd have to wait 12 hours on average to see the results. Cache headers might affect whether a document is retrieved from the backend system or not, and to test this functionality properly we might have to wait several days for results.

A good solution for this problem is to introduce the concept of business time into the system, a configurable replacement for the system clock.

The system should use the business clock when it wants to find out the current time or date. This allows us to travel in time easily during a test. That increases the complexity of the system a bit, but it allows us to test it very quickly.

A quick and dirty way to implement business time is to automate clock changes on the test environment, which doesn't require any design changes. Beware of applications that cache time, though, because they might require restarting after system clock updates. Changing the system clock might make test results more confusing if that time is used anywhere else, for example, in test execution reports. A more comprehensive solution is to build the business time functionality into the production software.

Introducing business time as a concept also solves the problem of expiring data. For example, tests that use expiring contracts might work nicely for six months and then suddenly start failing when the contracts expire. If the system supports business

time changes, we could ensure that those contracts never expire and reduce the cost of maintenance.

A potential risk with introducing business time is introducing synchronization problems with external systems that we can't influence. To address this, apply the test double or isolation ideas from earlier in this chapter.

Break long test packs into smaller modules

After several months of years of building executable specifications, many teams ended up with tests packs that take several hours to run. Because hundreds of tests are executed, people take it for granted that the feedback will be slow. They don't notice when something starts taking 20 minutes longer than it normally does. Such problems quickly accumulate, and the feedback gets even longer.

Instead of a big set of executable specifications that takes six hours to run, I'd rather have 12 smaller sets that each take no longer than 30 minutes. Generally, I break those apart according to functional areas.

If I want to fix a problem in the accounting subsystem, for example, I can quickly rerun just the accounting tests to check if the problem is gone. I don't have to wait six hours for all the other tests to finish.

If a single group of tests suddenly starts taking 10 minutes longer, I can easily spot that by looking at the test history for a particular group of tests. A 10-minute increment over six hours isn't as visible as a 10-minute increase over 30 minutes. This allows me to keep the feedback delay under control because I'll start looking for ways to optimize that particular test pack. By splitting a big test pack into several smaller packs, I recently helped a client realize that just one functional area was causing very long delays, and we cut that down from almost one hour to just nine minutes. Divide and conquer!

Avoid using in-memory databases for testing

When: Data-driven systems

Some teams with data-driven systems tried to speed up the feedback by running continuous validation tests against an in-memory database instead of a real database. This allows the system to still execute SQL code but makes all SQL calls run a lot faster. It

also makes test runs better isolated, because each test can run with its own in-memory database. But in practice, many teams found that running tests like this costs more than it's worth long term.

An executable specification that includes a database is most likely a functional acceptance test and an end-to-end integration test at the same time. Whoever wrote it like that wants to check SQL code execution as well. Minor SQL dialect differences between the real production database type and the in-memory database implementations might make test results misleading. This also often requires maintaining two sets of SQL files and managing data changes in two places.

If you use an in-memory database, the end-to-end integration test will validate that the system can work correctly against the in-memory database, not the real database that you'll use in production.

I've already mentioned some better solutions to this problem. Running tests in transactions provides better isolation. Mixing end-to-end integration tests and functional acceptance tests might not be the best idea (as mentioned earlier in this chapter), but if you really want to do that, then use the real database and look for ways to speed it up.

The team at Iowa Student Loan used in-memory databases for testing but gave up on that later. Tim Andersen says:

> We used SQL Server for our database and replaced it with Hypersonic to make it run in memory. That saved us 2 minutes of a 45-minute build. Once we added indexes to the database, SQL Server was actually faster. Hypersonic was more maintenance and didn't improve the build time that much.

If the tests are running slowly on realistic data, it probably means that the system is going to work slowly in production, so improving the performance of the database for testing actually makes sense in the grand scheme of things anyway. Note that the context of this tip is data-driven systems where executing specific database code is often required. In-memory databases can be a perfectly good solution for speeding up database-agnostic checks.

Separate quick and slow tests

When: A small number of tests take most of the time to execute

If a small subset of tests takes the majority of time to run, it might be a good idea to run only the quick tests frequently.

Many teams organized their executable specifications into two or three groups based on the speed of execution. At RainStor, for example, they have to run some of the tests with very large data sets to check the system performance. They have a functional pack that runs after every build and takes less than one hour. They also run customer scenarios overnight, with realistic data obtained from customers. They run long-running suites every weekend. Although this doesn't provide a full validation every time, it significantly reduces the risk of changes introducing problems while still providing relatively quick feedback.

Keep overnight packs stable

When: Slow tests run only overnight

A big problem with delayed execution of slow tests is that any issues in those tests won't get discovered and fixed quickly. If an overnight test pack fails, we find about that in the morning, try to fix it, and get the results the next morning. Such slow feedback can make the overnight build break frequently, which can mask additional problems introduced during the day.

Include tests into packs that run overnight only when they're unlikely to fail.

A good idea to keep the overnight packs stable is to move any failing tests into a separate test pack (see the "Create a known regression failures pack" section later in this chapter).

Another idea is to add tests into overnight packs only when they've been passing reliably for a while. Test-run history statistics (discussed earlier in this chapter) can help us decide when a test is good enough for an overnight pack.

If these tests are too slow to be executed continuously, a possible solution is to execute them periodically on demand until they become stable enough for delayed execution. Adam Knight at RainStor uses this strategy:

> Tests were executed manually until they were deemed reasonably stable and then executed through the nightly test [pack]. Having the separation of tests has benefited us in many ways. If a test fails, we get it fixed. A significant failure becomes our top priority.

By adding specifications into the overnight packs only when they're unlikely to fail, the team at RainStor reduced the risk of the packs with slower feedback failing because of features currently in development. This significantly reduced the maintenance costs for overnight packs. They'll still catch unexpected and unpredictable changes, which appear rarely. Considering that, slower feedback for such features is a good trade-off for lower maintenance costs.

Create a current iteration pack

A common special case of breaking long tests into smaller packs is creating the current iteration pack. This pack contains the executable specifications that are affected by the current development phase.

Having a current iteration pack, clearly separated, allows us to get very quick feedback on the most volatile and the most important area of the system for our current changes.

When the current iteration pack is split from the rest of the tests, we can safely include in it even the tests for the functionality that was planned but not yet implemented. Running all the tests for the current iteration will enable us to track the development progress easier and know exactly when we're finished. The current iteration pack might fail as a whole most of the time, but this won't affect the main regression validation.

A variant of this is to create a pack for the current release if we need to validate such specifications more frequently. Note that many automation tools allow us to create parallel hierarchies so the same specification can belong to multiple packs at the same time.

Parallelize test runs

When: You can get more than one test environment

If you're privileged enough to work in a company where setting up an additional test environment isn't a problem, once you divide a big test pack into several smaller ones, try to run them in parallel. This will give you the fastest feedback possible.

If some tests have to run in isolation and can't be executed in parallel, it's worth splitting them out into a separate pack as well. At LMAX, Jodie Parker organized continuous integration and validation in this way:

> Commit build ran all unit tests and statistical analysis within 3 minutes. If that passed, sequential boxes executed tests that needed to be run one after another or not in parallel. Then 23 virtual machines ran acceptance tests in parallel. After that, a performance test kicked off. They [executable specifications] typically ran between 8 and 20 minutes.

> At the end, if the tests passed a certain amount, the QA instance was available to deploy to run smoke tests and exploratory tests and provide feedback to development. If the commit build failed, everyone had to stop (a complete embargo) and fix it.

If you're not paranoid about security, or don't work under regulatory constraints that prevent you from deploying the code outside your organization, emerging cloud computing services can help with parallel test runs. Deploying the code remotely takes some time, but this allows you to run tests on a very large number of machines at the same time. At Songkick, they use Amazon's EC2 cloud to run acceptance tests. Phil Cowans said that helped them cut down build time significantly:

> Running the full test suite on a single machine would take 3 hours but we parallelize. We just learned how to do this on EC2 to bring it down to 20 minutes.

Some continuous build systems, such as *TeamCity*,[4] now offer test execution on EC2 as a standard feature. This makes it even easier to use computing clouds for continuous validation. There are also emerging services that offer automation through cloud services, such as Sauce Labs, which might be worth investigating.

Try disabling less risky tests

When: Test feedback is very slow

The team at uSwitch has a unique solution for slow feedback of long-running test packs. They disable less risky tests after the features described by those tests are implemented. Damon Morgan said:

> Sometimes you write acceptance tests (which are very good to drive development) that aren't that important to keep around once a feature has been developed. Something that is not moneymaking—e.g., sending delayed emails—is not a truly core part of the site but adds functionality to it.... They [executable specifications] were very good to help us drive the development, but after that, keeping them running as a regression pack wasn't that useful to us. It was more hassle to maintain them than to throw them away. We do have the tests in our source control; they are just not running. If we have to extend the functionality, we can still modify an existing test.

[4] http://www.jetbrains.com/teamcity

For me, this is quite a controversial idea. I'm of the opinion that if a test is worth running, it's worth running all the time.[5]

Most of the tests at uSwitch run through the web user interface, so they take a long time and are quite expensive to maintain, which is what pushed them in this direction. With a system that doesn't make you run tests end to end all the time, I'd prefer trying to automate the tests differently so that they don't cost so much (see the "Automate below the skin of the application" section in chapter 9).

One of the reasons why uSwitch can afford to disable tests is that they have a separate system for monitoring user experience on their production website, which tells them if users start seeing errors or if the usage of a particular feature suddenly drops.

After covering faster feedback and more reliable validation results, it's time to tackle something much more controversial. As the size of their living documentation system grew, many teams realized that they sometimes need to live with failing tests occasionally. In the following section, I present some good ideas for dealing with failing tests that you might not be able to fix straightaway.

Managing failing tests

In *Bridging the Communication Gap*, I wrote that failing tests should never be disabled but fixed straightaway. The research for this book changed my perspective on that issue slightly. Some teams had hundreds or thousands of checks in continuous validation, validating functionality that was built over several years. The calculation engine system at Weyerhaeuser (mentioned in the "Higher product quality" section in chapter 1) is continuously validated by more than 30,000 checks, according to Pierre Veragen.

Frequently validating so many specifications will catch many problems. On the other hand, running so many checks often means that the feedback will be slow, so problems won't be instantly spotted or solved. Some issues might also require clarification from business users or might be a lesser priority to fix than the changes that should go live as part of the current iteration, so we won't necessarily be able to fix all the problems as soon as we spot them.

This means that some tests in the continuous validation pack will fail and stay broken for a while. When a test pack is broken, people tend not to look for additional problems they might have caused, so just leaving these tests as they are isn't a good idea. Here are some tips for managing functional regression in the system.

[5] To be fair, I picked this idea up from David Evans, so if you want to quote it, use him as a reference.

Create a known regression failures pack

Similar to the idea of creating a separate pack of executable specifications for the current iteration, many teams have created a specific pack to contain tests that are expected to fail.

When you discover a regression failure and decide not to fix it straightaway, moving that test into a separate pack allows the tests to fail without breaking the main validation set.

Grouping failing tests in a separate pack also allows us to track the number of such problems and prevent relaxing the rules around temporary failures from becoming a "get out of jail free" card that will cause problems to pile up.

A separate pack also allows us to run all failing tests periodically. Even if the test fails, it's worth running to check if there are any additional failures. At RainStor, they mark such tests as bugs but still execute them to check for any further functional regression. Adam Knight says:

> "One day you might have the test failing because of trailing zeros. If the next day that test fails, you might not want to check it because you know the test fails. But it might be returning a completely wrong numeric result."

A potential risk with a pack for regression failures is that it can become a "get out of jail free" card for quality problems that are identified late in a delivery phase. Use the pack for known failures only as a temporary holding space for problems that require further clarification. Catching issues late is a warning sign of an underlying process problem. A good strategy to avoid falling into this trap is to limit the number of tests that can be moved to a known regression pack. Some tools, such as Cucumber, even support automated checking for such limits.

Creating a separate pack where all failing tests are collected is good from the project management perspective, because we can monitor it and take action if it starts growing too much. One or two less-important issues might not be a cause to stop a release to production, but if the pack grows to dozens of tests, it's time to stop and clean up before things get out of hand. If a test spends a long time in the known regression pack, that might be an argument to drop the related functionality because nobody cares about it.

Automatically check which tests are turned off

When: Failing tests are disabled, not moved to a separate pack

Some teams don't move failing tests into a separate pack, but they disable them so that a known failing test won't break the overall validation. The problem with this approach is that it's easy to forget about such disabled tests.

A separate pack allows us to monitor the problems and ensure that they eventually get fixed or that we don't waste time troubleshooting a similar problem again until the issue gets fixed. We can't do this easily with tests that are disabled. An additional problem is that someone might disable a high-priority failure without understanding that it should actually be fixed straightaway.

If you have tests that are disabled, automatically monitor them.

The team at Iowa Student Loan has an automated test that checks to see which tests are disabled. Tim Andersen said:

> People were turning tests off because we needed a decision or we were writing a new test and weren't sure how this old test fit in. There were conversations that never got followed up on, or people just forgot to turn the test back on. Sometimes the tests were turned off because people were working on it and there was no code behind it yet.
>
> We used FitNesse to find the tests that were turned off, and we had a page that checked all of those test names. We'd use that to list the tests that were intentionally turned off and put a JIRA [an issue-tracking system] ticket next to each test. So a turned-off list acts as another test. It has to match what you say you turned off. At the end of an iteration, we follow up on those tests that were turned off. We could say, "This is no longer applicable, let's just delete the test," or "Oh, there's a difference and we didn't really hear back from the business," and in these cases we'd have to fix the test.

If you opt for temporarily disabling broken tests instead of moving them out into a separate pack, make sure that you can monitor them easily and prevent people from forgetting about the disabled tests. Otherwise, the living documentation system you build will quickly get outdated. Disabling executable specifications is a quick and dirty temporary fix for a broken test, but it defeats the whole point of continuous validation.

Once we have specifications that are continuously validated, it becomes easy to assert what a system does from a functional perspective, at least for the parts of it that are covered with executable specifications. The specifications then become a living documentation system that explains the functionality. In the next chapter, I present some good ideas on how to get the most out of your set of executable specifications by evolving a documentation system.

Remember

- Validate executable specifications frequently to keep them reliable.
- Compared to continuous integration with unit tests, the two main challenges for continuous validation are fast feedback and stability.
- Set up an isolated environment for continuous validation and fully automate deployments to make it more reliable.
- Look for ways to get faster feedback. Split quick and slow tests, create a pack for current iteration specifications, and divide long-running packs of executable specifications into smaller packs.
- Don't just disable failing tests—either fix the problems or move the tests to a pack for low-priority regression issues that's closely monitored.