Realtime publishers

The Definitive Guide™ To

Quality Application Delivery

sponsored by

MICRO FOCUS®
Leading the Evolution™

Don Jones

Copyright Statement

Chapter 8: Functional Testing—Verifying Quality

I've already mentioned—probably several times—that automated testing cannot, by itself, add more quality to an application. That is, all an automation tool can do is make things faster and less tedious; it cannot inherently make them better. That's why this chapter will focus on *testing* generally and not specifically on tools. You need to have a solid testing methodology in place before you begin automating it; in other words, you need to create the desired level of quality using manual processes, and then you can begin relying on tools to help automate those processes.

Of course, there *is* a caveat to the previous statements: Automated testing *can* allow you to add *more of the same* quality to your application than you could add manually. That is, if you have a good testing plan in place to begin with, then automated tools will allow you to accomplish more testing than you could do manually. More testing, if it's the right kind of testing, does lead naturally to an increase in application quality in most cases. However, it's important to understand that the automation tool is simply helping you get more done in a smaller amount of time—it's *what* you do with that time that's actually important.

Testing to Requirements

First and foremost, remember this important law of software testing:

You cannot begin testing without well-written requirements, and you can only test what is specifically called out in the requirements.

There's simply no way around it. Without solid requirements, you're testing blind; your tests should focus almost entirely on the things which are specifically called out in your software's requirements document. In fact, this is an area where tools can immediately start helping. Requirements-tracking tools allow you to import requirements from documents (such as those created in Microsoft Word), helping you quickly turn a shared document into a set of clearly-defined, trackable requirements. Tools can also help generate test cases based on those requirements, helping to automate the process of creating a test plan and helping to make sure you don't miss any requirements in your testing. Tools also help make requirements more accessible to your entire development team, often by providing role-based security, which allows specified team members to edit requirements, others to view requirements, others to contribute comments, and so forth—and to automatically send notifications when requirements are updated or expanded.

Let's look at exactly how testing should be driven by requirements:

- In creating your requirements, you should be documenting everything you want the application to eventually do.
- Those requirements should essentially become a checklist for the testing phase.
- You can generate test cases and ensure that each requirement is covered by at least one test case.
- You then track which test cases have been tested, which are satisfactorily completed, and so forth.

Risk management comes into play with the requirements. Different organizations have different ways of assigning risk; some will assign a value of 1 to 3 based on the perceived likelihood of the risk occurring, and then a value of 1 to 3 based on the severity of the risk if it does occur. Multiply these two values for a 1 to 9 score of the total risk associated with any given requirement. Why bother? That same risk value can be carried down to the test cases that support each requirement, and you can easily identify those test cases that might need to be run more frequently or in more detail to help prevent the highest-impact, most-likely risks from ever occurring.

> **Note**
> Why didn't I discuss risk assignment in more detail in the chapter on requirements? It's kind of a personal decision on my part. I really believe that one of the great, main purposes of requirements is to drive your testing; without testing, assigning risk isn't as useful. One of the main points of assigning a risk value to a requirement is to help focus appropriate testing resources. This is one of the many ways in which requirements development and testing connect closely to one another.

If you think about it, a close tie between requirements and testing can be an invaluable management tool. Imagine a table like this:

Requirement	Tests to Run	Tests Completed	Test Time Remaining
3.4	34	100%	0 hours
3.5.1	3	33%	2 hours
3.5.2	14	12%	21 hours

This type of table directly tracks each application's requirements, the number of tests that validate the requirements, and how many of those are complete. You can use this to estimate the remaining test time, which helps tell you how long the project has before it will be completed.

Note

Some of the automated test management tools I'll discuss later in this chapter help to automate this kind of reporting, making it easier to pull up these and other useful management views of the project.

Because the application's requirements should directly communicate the business' expectations for the application and the business' view of what will constitute a "quality" application, testing can also provide a valuable insight into the application's current level of quality. Imagine another table like this one:

Requirement	Tests Run	Successful Tests	Quality
3.4	100%	50%	50%
3.5.1	33%	100%	33%
3.5.2	12%	50%	6%
AVERAGE			**30%**

Again, having access to this type of information can drive important management decisions. How far along are we in the project? What level of quality are we at, and how much effort was required to get there? How much effort is likely to be required to raise our quality to 100%? Can we afford that? If we settle for 90% quality, what will we be losing? That last question is especially important, because it offers the business the opportunity to deliberately accept a less-than-perfect application as a tradeoff for using fewer resources such as time and money. It's very costly to create a 100%-perfect, 100%-quality application—so attaching test results and requirements, management can help strike a desirable balance between resources and quality.

Test Data

I've worked on a number of pretty large-scale applications, and while I don't have formal statistics to back it up, my gut tells me that about 75% of the bugs released in production could have been avoided by having better test data. Imagine making your applications have three-quarters fewer bugs—surely that would contribute to a perception of increased quality, no?

There are two phases where I frequently see missed bugs as well as wasted effort: Developers using poor test data in their unit testing is the first. The basic problem here is that developers hate managing test data—honestly, who *likes* doing so?—and so they tend to perform unit tests with pretty poor data. The result is that they miss what would be simple bugs to fix in development, and wind up shipping those bugs to formal testing. I do see organizations who *have* a formal testing process tend to use slightly better test data, and they do catch a lot of these simple bugs, but it's wasted effort: The time to test, find the bugs, ship them back to the developer, fix them, and re-test them is all wasted cycles that could have been avoided by using better test data in the first place.

So what constitutes good test data? It might be easier to look at what constitutes *poor* test data. Take a look at some example data that represents customer records:

ID	Name	Address	City	Phone
1	John Doe	123 Anystreet	City	5551212
2	John2	Street	City	5551212
3	John3	Street	City	5551212

This is test data I've seen—heck, that I've *used*—in a lot of unit tests. It's simple data, and it's fine for doing simple tests to make sure that a UI is adding data to a database properly, for example. It's easy to create, and it is simple enough that it won't require a lot of complicated management. Now look at the data in the following table:

ID	Name	Address	City	Phone
373525	Abdul Irwin	728-3967 Purus. Avenue	Pittsburgh	1-466-816-6750
220368	Yoshio Herman	P.O. Box 147, 9975 Interdum Rd.	Clovis	1-647-941-7454
810956	Dieter Shelton	Ap #634-5883 Est Avenue	Fort Wayne	1-413-617-6786
752662	Anthony D'ffy	P.O. Box 663, 9750 Interdum. Av.	Wilkes-Barre	1-154-882-2798
332754	Curran Decker	6110 Orci. Rd.	High Point	1-432-213-3305
926816	Ashton Long	P.O. Box 846, 4959 Odio, Rd.	Flint	1-222-154-2000

656679	Ian James	Ap #219-642 Sollicitudin Ave	Winston-Salem	1-415-420-3801
355623	Garrison Fisher	248-1121 Ut Rd.	Miami	1-215-860-8460
461819	Lucius Huff	8431 Duis St.	Fairmont	1-273-307-6639
819085	Ishmael Jackson	P.O. Box 912, 3564 Risus. Ave	Canton	1-877-824-2860
674310	Ulric Dejesus	Ap #496-688 Maecenas Street	Catskill	1-140-995-9658
964383	Nicholas Moran	944-9597 Turpis. Rd.	Pottsville	1-758-550-8852

I generated this test data from www.GenerateData.com, using their example script. This data has some higher-quality features:

- There are full names with a variety of characters and lengths, including one with an apostrophe—which is very important in testing the escaping of special characters. Foreign names with umlauts and accents would be desirable, too.
- Addresses are lengthier and contain common punctuation
- City names are longer and includes some with two words and some with punctuation
- Phone numbers are full-length and suitably random.

In addition, the Web site makes it easy to generate hundreds of rows of data like this, making it easier to perform a greater variety of tests during unit testing. Random data like this can be especially useful throughout the testing cycle, but there are some additional things you can do to make it even better.

Deliberate Data

Random data is great, but it's not perfect. Ideally, you should *start* with random data, but then massage it a bit to make sure it covers all the possible boundary conditions. Add weird characters, and add data that *should be rejected* by the application, such as data fields that are too short, too long, out of range, and so on. For example, if a database table has 10 columns, there should be at least 20 rows of data that tests for illegal data: For each column, create one row of data that has legal data in all columns but one. That gives you 10 rows of test data, each one testing the length limitation of a single column, and an additional 10 rows of test data with each one testing the data type of a single column. For example, consider a table that has four columns:

Column Name	Data Type and Length
Name	Character (100)
Address	Character (100)
ID	Integer (5)
Postal	Character (10)

This would require at least eight rows of test data, such as these:

Name	Address	ID	Postal
A a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a A a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a A a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a A a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a	123 Main Street	5	99654
Don Jones	A a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a A a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a A a a a a a a a a a a	17	23455

	a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a A a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a		
Greg Shields	5252 Providence Rd	666666666666666	77342
Chris Gannon	230 Windmill Ln	7394	8888888888888888
7	390 Kelley Ave	7584	74738
Bailey Sory	8	8738	22312
David Knight	300 Elizabeth St	ABCDE	9986
Stonna Edelman	8989 Broadway	88	ABCDE

The first four rows each try to place oversized data into a single column; the second four rows each try to place data of a different type into a single column. There's *still* a need to test a quantity of *valid* data, of course; this is data you might manually add to your test data simply to ensure that *illegal* data doesn't cause any unhandled errors.

> **Note**
> Why not just use a single test row that places illegal data into every column? Doing so makes debugging more difficult. This way, each row causes only a single problem, so any bugs which are found must relate to that single column.

Of course, this is just a simplistic example. The idea is to make sure that your test data is *thorough,* and that it tests a variety of legal and illegal conditions.

> **Note**
> There are tools out there which can examine your database and produce illegal test data automatically. This is obviously far easier—and often more thorough—than producing the data manually, but you should *still* review the data to make sure it's testing every possible condition. Automated tools can't interpret your code to produce test data; they can only (in most cases) analyze your database schema for things like field lengths and data types.

Production Data

Another option is to use production data—real, live data from an existing system. This isn't practical when you're building a brand-new application, of course, but it is practical when building a new system that will replace an existing one, or when building a new version of an existing application.

Using production data needs to be done with some care. There are three main concerns:

- Be aware of any privacy issues related to the data, such as customer information. This data may need to be "disguised" by replacing random characters and digits, or it may need to fall under special security measures to protect it and audit access to it.

  Caution

  Privacy issues are especially important if your data is covered by industry or legislative requirements, such as the Health Insurance Portability and Accountability Act (HIPAA). Check with your organization's security organization before utilizing production data, and rely on tools that can extract the production data and disguise it at the same time. It's almost always more practical to disguise data than to try and manage compliance requirements on test data.

  In the event that you do need to use undisguised data that has privacy implications, you will *definitely* need a test data management tool that can secure the data, audit access to and use of the data, produce audit reports for your organization's auditors, and so forth.

- Be sure that your production data is testing all possible data conditions, including things like punctuation and so forth in various fields. Don't just grab a few production data rows at random; intelligently select rows that will properly and thoroughly test your application.

  Note

  Using production data implies that you have an existing application; review the old bugs that were filed for that application to find scenarios where unexpected data caused problems. Those same scenarios should become part of your new application's tests, and you can create appropriate data to make sure those scenarios occur during your testing.

- Be sure to include test data that includes illegal data, which you're (hopefully) unlikely to obtain from an existing production system.

Typically, you'll extract the desired test data from the production system one time, and then store that test data elsewhere so that it can be easily accessed and re-used. Which brings us to the topic of managing your test data.

Managing Data

There are two simple, easy to understand reasons why developers and even QA testers tend to use poor production data:

- Good data is hard to generate
- Good data is hard to manage (store and re-use)

We've already covered better ways to generate test data, so let's look at some of the management issues. In order for your high-quality test data to be valuable, it's got to be used—both by developers during unit testing and by QA during full-scale functional testing. In order for the data to be used in these scenarios, it needs to have a few characteristics:

- It must be centrally stored and easily accessible
- It must be secured against tampering, so that nobody tries to "dumb down" the data to make testing easier
- It must be updatable by the right people, so that the data set can be expanded to cover newly-discovered boundary conditions
- It must be easy to load the data into a database or other form, or feed it into a user interface, to set up a test scenario

Nearly all of these things are difficult to do manually—which is where tools come into the picture. The right test data management tool can perform all of these tasks, and may even be able to assist with the generation of test data, disguising test data from a production system, and even extracting data from other systems to use in the test environment. Yes, you can absolutely handle all of these tasks without having to obtain tools—but your testing will end up being more thorough, and you'll end up wasting fewer test-debug-retest cycles, if test data is managed and easy for everyone on the team to utilize.

> **Note**
>
> One you start shopping for tools, it pays to consider *all* of the tools you'll need. A good test data management tool, for example, might be able to feed data to an automated testing tool that can enter the data into a graphical user interface. That's a desirable feature, but to get it you'll need to make sure you're selecting tools that fit together in that way. Start by looking at tool vendors that offer suites of integrated tools, or vendors who make tools specifically designed to integrate with tools from other leading vendors.

Unit Testing

Unit testing is a method where individual programmers test the individual units of code that they are writing. A unit test is generally considered to be the smallest unit of testing that can be performed against an application: It might be a single function or object method, for example. Stepping through the code in a debugger may be the only way to conduct unit tests on some portions of code; in other cases you may have to create a *test harness,* a separate set of code that is designed solely to load and test the code you're actually interested in.

Unit tests should, ideally, be some of the most thorough testing performed against an individual unit of code. In other words, developers *should* be testing every single code path within a module, using appropriate test data, to make sure that the entire unit of code is completely tested. Other testing later in the project's lifecycle may be just as thorough, but limited resources often mean that later "full" or *functional* testing may not be able to test every combination of code paths offered by numerous integrated units of code. Figure 8.1 is a flowchart that illustrates the logic for a given unit of code; having a flowchart like this helps to ensure that each code path is tested.

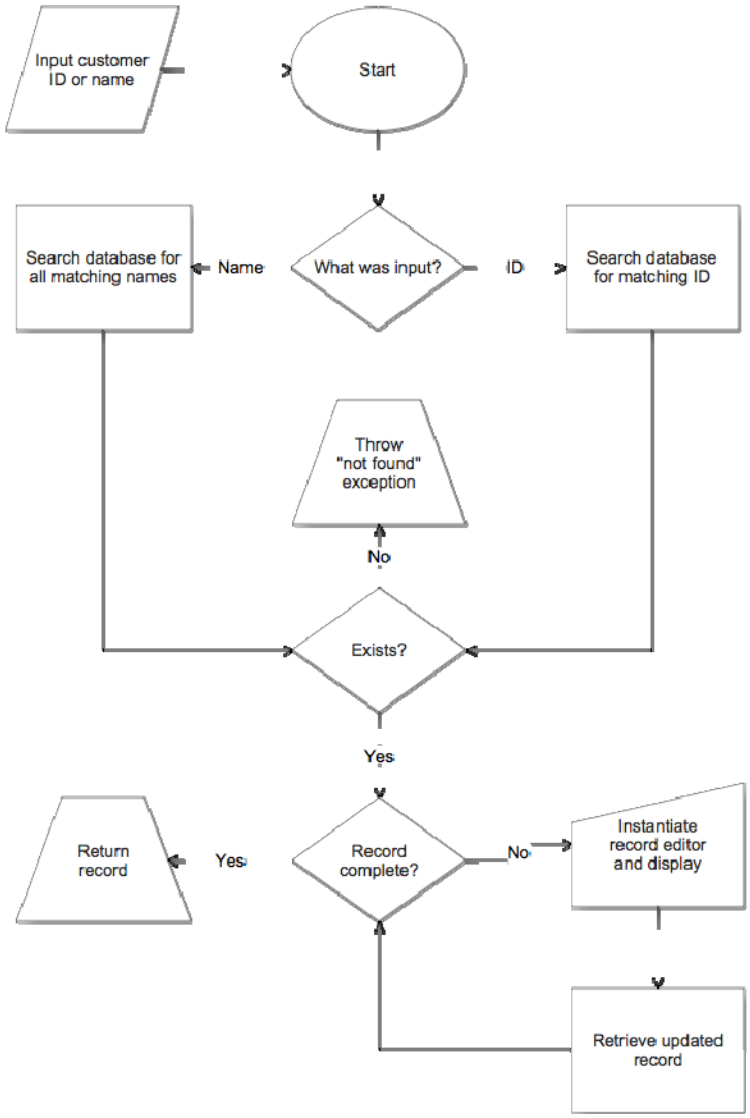

Figure 8.1: Flowcharts help direct and guide unit testing.

The flowchart not only depicts the conditions necessary to enter each code path, but provides a sort of checklist. As shown in Figure 8.2, you can highlight each path as you test it, and even indicate which test data sets you used to test individual paths.

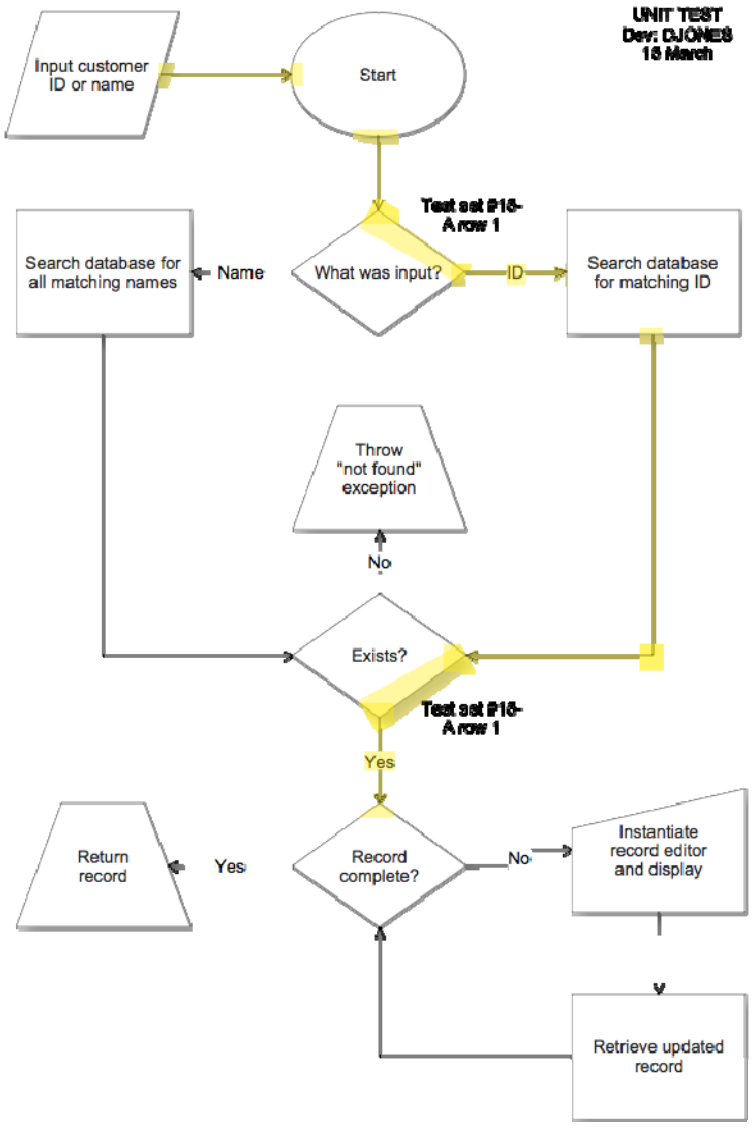

Figure 8.2: Tracking the results of unit testing on a flowchart.

Of course, this type of flowchart isn't just useful for manual unit testing—it's also very helpful in setting up *automated* unit tests. The flowchart helps make sure that an automated test is addressing every code path, and helps document the specific test data that will be used during various aspects of the test.

It's often difficult to test a single unit of code such as an object method, because code tends to rely on other code—and the depended-upon code may not be complete or fully-tested. Ideally, unit tests should focus on a *single unit* of code, with no outside dependencies. This may mean developing test harnesses to instantiate and activate the code, and it may also mean creating "mock objects" to replace depended-upon code. These mock objects can be very simple pieces of code that return static results, allowing the unit test to be completely self-contained and predictable. Again, automated testing tools can help make this overall process less painful, more consistent, and much faster.

Microsoft's Visual Studio documentation provides a good definition of unit testing with regard to isolation:

> The primary goal of unit testing is to take the smallest piece of testable software in the application, isolate it from the remainder of the code, and determine whether it behaves exactly as you expect. Each unit is tested separately before integrating them into modules to test the interfaces between modules. Unit testing has proven its value in that a large percentage of defects are identified during its use.

If you don't isolate units, you can substantially increase debugging time. For example, suppose you have two units, 1 and 2, and you decide to test them together. When an error occurs, you'll have to determine:

- Is the error in unit 1?
- Is the error in unit 2?
- Is the error in both units?
- Is the error due to the interface between the units?
- Is the error due to a defect in the test?

By isolating unit 1, and having it depend on a mock object (also called a *stub* by some developers) rather than depending on unit 2, you eliminate many of these possibilities. You're left with only two:

- Is the error in unit 1?
- Is the error due to a defect in the test?

This makes tracing the error and fixing it much easier and more efficient.

Functional Testing

Functional testing—also called *integration testing, system testing,* and numerous other terms by developers I've worked with—is the act of testing a complete system, or a complete subsystem of a larger application. I've also seen it referred to as *black box* testing, because it should require little or no knowledge of the application's inner workings (unlike unit testing, which may require that knowledge to create test harnesses, stubs, and so forth).

I have a firm opinion regarding functional testing and bugs: Functional testing *will* reveal bugs in your application, but it will not find *all* of them. The *main* purpose of functional testing is to ensure that each and every business requirement is met; finding bugs is an inevitable side benefit. Why is this so? It's a question of complexity. A given unit of code might have a half-dozen code paths, each of which must be tested in order to discover all of the bugs within that unit. If every unit of code in the application averages six code paths, and the application consists of two hundred such units, then you have an enormous number of discrete code paths to test—it's not practical to think that a functional test, no matter how well-automated, can catch them all.

> **Note**
> I tend to think of unit testing as the bug-catching phase, and functional testing as the requirements-checking phase, but obviously there's overlap. Some business requirements can be checked at the unit level, depending on the unit, and as I've said you'll almost always uncover bugs during functional testing, even if that's not your specific goal.

There are many goals for functional testing; the article at http://en.wikipedia.org/wiki/System_testing lists a nice set:

- GUI software testing
- Usability testing
- Performance testing
- Compatibility testing
- Error handling testing
- Load testing
- Volume testing
- Stress testing
- User help testing
- Security testing
- Scalability testing

- Capacity testing
- Reliability testing
- Recovery testing
- Installation testing
- Idempotency testing
- Maintenance testing
- Accessibility testing

Yes, that's a lot of work—but most of these things actually relate to specific business requirements that are common to most applications.

> **Note**
> I tend to use this list of functional tests as a sanity check. If I've created all the test cases for all of my business requirements, and I'm not seeing test cases for one of these bulleted items, then I have a problem. If, for example, my complete list of functional tests don't include some load testing, then I start to wonder why my business requirements don't specifically state any capacity or load requirements.

Another article I like, at http://www.devbistro.com/articles/Testing/Requirements-Based-Functional-Testing, has a very succinct definition for functional testing:

> The objective of function test is to measure the quality of the functional (business) components of the system. Tests verify that the system behaves correctly from the user / business perspective and functions according to the requirements, models, storyboards, or any other design paradigm used to specify the application. The function test must determine if each component or business event: performs in accordance to the specifications, responds correctly to all conditions that may be presented by incoming events / data, moves data correctly from one business event to the next (including data stores), and that business events are initiated in the order required to meet the business objectives of the system.

This stresses that the focus of functional testing is not on bugs, but on *functionality* (that's actually why I prefer the term *functional test* over something like *integration test* or *system test*).

Functional testing should be regarded as a distinct entity that is not necessarily related to unit testing. Unit testing must often be planned and implemented as developers write code, since things like test harnesses and stubs often can't be fully defined in advance. Functional testing, however, *can* be fully planned in advance, because it's driven solely by the business requirements for the application. Functional testing doesn't care about the code; it only cares whether the finished code is meeting the requirements.

That said, planning for functional testing isn't always as simple as creating one test case for each business requirement (in fact, it's rarely that simple). *Partitioning* is often necessary to break complex requirements down into testable units; this is a process also referred to as *functional decomposition*. The idea is to break down the specific functional areas of the application so that they can be tested—perhaps before the entire application is complete—against the relevant business requirements.

Rethinking Testing

It's time to completely re-think the way you test. Even if you're already using automated testing tools—which, by themselves, are not a guarantee of quality—you can probably improve your testing processes and bring a great deal of additional quality to your applications. Here are some things to consider:

- Are your tests mapping directly to the original application requirements? Tests should give management an idea of the application's overall quality, as well as an idea of the application's progress—based on the business requirements that have been met.
- Do you have metrics in place which can be easily viewed? In other words, can project managers track the project's overall quality, overall completeness, and so forth?
- Are metrics in place to help balance risk and resources? Can management quickly target riskier elements of the application to have more testing, thus avoiding risk, and see which elements of the application are meeting or failing their tests?
- Are you using production-quality, varied test data that is centrally accessible and easily usable? Are you also using deliberately illegal data to test boundary conditions and the application's handling of errors?
- Do you have a thorough set of test cases? In other words, do you have test cases that test every possible code path, and test for compliance with every relevant business requirement? Can you quickly identify business requirements which are *not* being tested by a specific test case?
- Can any developer or tester quickly access the necessary test data?
- Are developers conducting thorough unit tests that are self-contained, and documenting the results of those tests?
- Have you planned for functional tests that address every single business requirement?

If this seems like a lot of work—well, that's because it is. My experience is that testing—including test case development, creation of test harnesses and stubs, and so forth—can easily be as much effort as the programming itself. However, unlike programming—which pretty much has to be done by human beings—testing involves a lot more repetition and management, which opens an opportunity for tools to help reduce the amount of human effort required.

The Role of Tools

I've already described some of the ways in which tools can help make testing and requirements tracking easier. There are other features which are commonly available in requirements-tracking tools which you may find desirable in your environment—use this list to help construct a sort of "shopping list" when you're comparing solutions:

- Integration with project management tools, such as Microsoft Project, Microsoft Excel, and so forth; you may also opt for a solution which has its own built-in project management capabilities
- Automated generation of test cases based on requirements
- Analysis tools that help you prioritize work and manage trade-offs between workload and resources
- Role-based security that allows the entire team to participate and have access to requirements
- Notifications to keep the entire team informed of changes (especially if you are using an Agile-based development framework, which relies heavily on constant communications for success)
- The ability to create (or store externally-created) visual models from your requirements, including process models, flow charts, data flows, and so on
- Reporting to help maintain compliance with industry or legislative requirements
- Change control, so that changes to requirements can be reviewed and rolled back at any time

You may wonder why I'm discussing these features in this chapter, rather than in the chapter where I first discussed requirements. The reason is simple: Tools with these features help automate a process that you generally must master manually. In other words, you shouldn't immediately turn to tools to start improving quality. As I've written earlier, you should master the process manually, and then use tools to automate it. There's another reason, too: Testing is where requirements sort of meet reality. Yes, requirements should drive the development process, the design process, and any other phases of the application lifecycle. But testing is where you *prove* that those processes have (or have not) been doing so. Testing essentially represents the end of the software development lifecycle (from there, the application moves to deployment in most cases), and so testing "closes the loop" and is your opportunity to make sure you've met your requirements. In my mind, *requirements development* and *requirements testing* are two faces of the same thing; it makes sense for me to discuss testing tools and requirements management tools because I truly think they're just about the same thing.

My students and consulting clients often give me a little push-back about that, often stating that testing is making sure the application works—and that testing tools, therefore, are primarily about test data management, testing automation, and so forth. They're absolutely correct in that those things are *part* of what testing is all about—but they're revealing some weaknesses in their quality maturity by not more solidly linking testing and requirements.

Of course, testing automation is still extremely valuable in terms of making it easier to go through all of your test cases, and plenty of vendors offer tools to fill this space. I tend to organize these into four major feature sets:

- Requirements management, which I've already discussed, is responsible for tracking requirements and (generally) generating and tracking the test cases which will tell us if the application meets the requirements or not.
- Test management, which is responsible for centralizing test plans, test design, and test execution, often involving a library of re-usable test assets like scripts, data, and so forth.
- Test automation, which actually executes tests against your application's code. Tools in this category offer a variety of visual metaphors and workflows to help make automation easier, because it is in effect a type of programming all on its own.
- Test data management, which helps manage sets of test data that can feed automated test execution.

Note

My goal with the screen shots here are to help you visualize what a tool might offer. To help remove any vendor-specific focus, I'm cropping these to focus tightly on the specific element I'm discussing; you'll find that many vendors adopt similar visuals for specific key features.

In most cases, a given vendor will offer some mix of these capabilities within a single solution or within a solution suite.

In the category of test management, I tend to look for features like these:

- Requirements-driven. Tools should integrate with your requirements management tools so that you can generate test plans that derive directly from your requirements. This is an excellent way to help validate the completeness of your requirements, too: If there are conspicuously-missing test plans, you need to ensure there are actually requirements to drive them. Figure 8.3 shows how requirements can be managed, and how you can use the tool to track completion percentages and other details.

Name	Coverage (%)	Risk	Tests	Passed	Failed	Defects
Requirements Center (2)	100%	Ignore	200	189	11	4
Layout (3)	100%	Above Avg	7	7	0	0
Help (2)	100%	Above Avg	4	4	0	0
Detail View Actions Menu	100%	Above Avg	12	12	0	0
Detail View Attachments Actions	100%	Above Avg	12	12	0	0
Detail View Creating Requirements	100%	Above Avg	26	25	1	0
Detail View Deleting Requirements	100%	Above Avg	2	2	0	0
MT_REQM000510 Main Menu and...	100%	High	1	1	0	0
MT_REQM000530 Project Explore...	100%	High	1	1	0	0
Detail View Links Actions	100%	Above Avg	10	7	3	0
Detail View Right Click Menu	100%	Above Avg	7	6	1	0

Figure 8.3: Tracking requirements for an application.

- Risk-driven. Frankly, you can't ever afford to test as much as you'd like. Good management tools help you focus on high-risk elements of your application, concentrating your resources where they'll do the most good. Figure 8.4 shows how a tool can help. Here, tests are being allocated by time, with different test loads scheduled for different categories of risk. At the bottom, the tool indicates how much coverage this plan offers in terms of hitting all the requirements, helping you decide where to focus your resources and showing you the potential consequences.

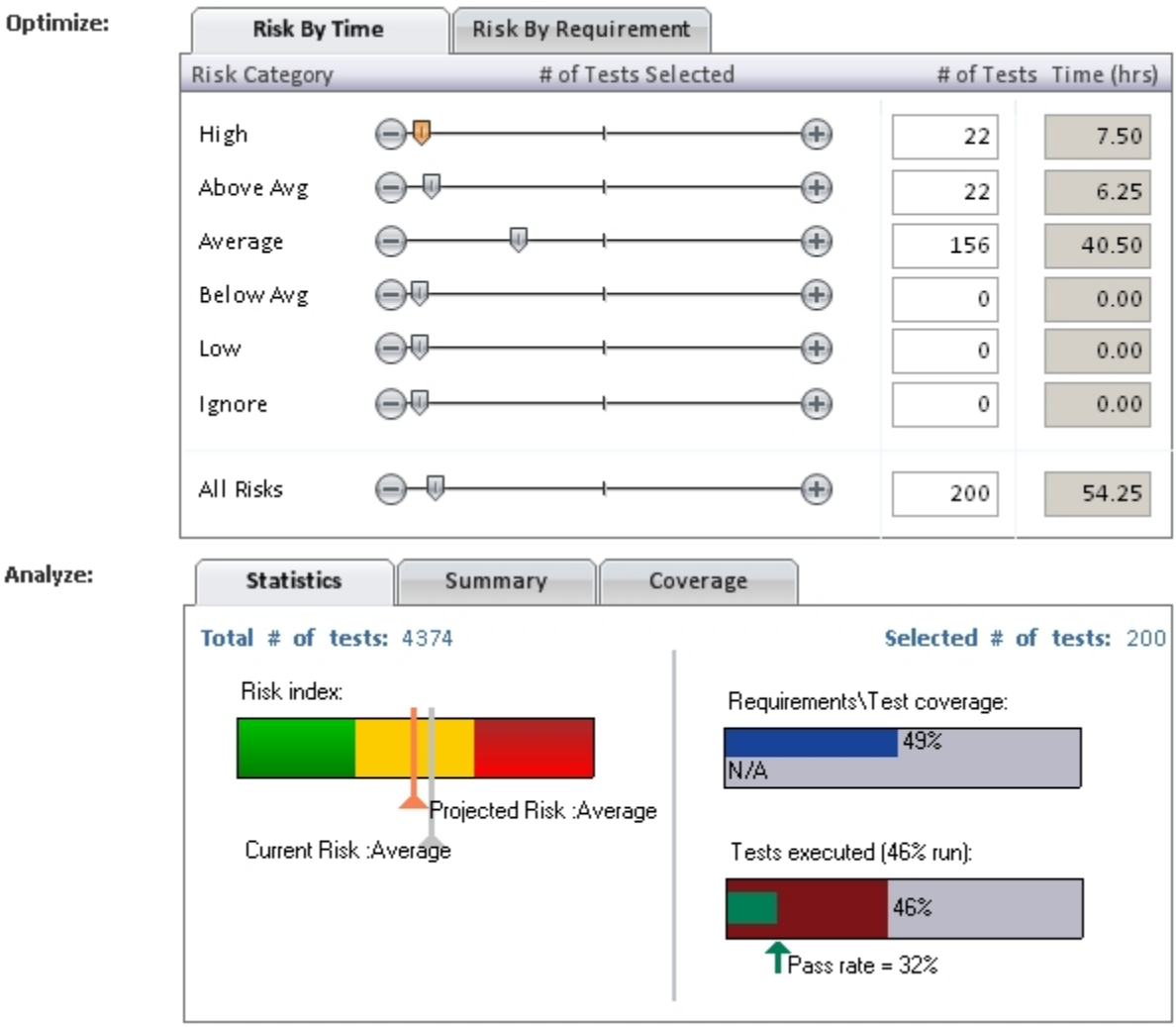

Figure 8.4: Managing requirements based on priority, risk, and other factors.

- Dashboards. A good management tool should provide high-level management views that help reveal the application's quality level, what requirements have been tested and met, what requirements haven't yet been tested, how much testing is left (and how long it will take), and so forth.

When it comes to the more detailed task of testing automation, I look for features like these:

- Test-building should involve as little programming as *possible.* Visual metaphors like storyboarding and visual workflow generators should help minimize scripting; they can rarely eliminate it entirely, but the best tools may be able to test certain applications without any kind of actual programming. Figure 8.5 shows how a testing automation tool can help automate information entry and testing by using a visual test design interface.

Figure 8.5: Visually designing a test: The application's GUI is on the left, and the test steps on the right.

- Support for sophisticated test cases. These should include the ability to make logical decisions to help test various code paths, to examine the state of the application and data, and so forth.
- Easy maintenance of test assets. Most testing tools rely on an understanding of things like database layout, application screen layouts, and so forth. Tools that can help automatically update this information to accommodate a changing, under-development application, help these test assets remain viable and usable for longer, with less ongoing work.
- Scripting support. While you want to minimize the amount of programming you have to do in order to run tests—programming invariably introduces bugs, and having bugs in your test suites creates a lot of complexity to worry about—you *do* want the *ability* to manually script tests when needed. This feature is often provided through proprietary scripting languages, or through more widely-used standards like Visual Basic.

Finally, in the last category of tools we have data management. I discussed the importance of this earlier in this chapter, and of course it's an area where good tools can make all the difference in terms of consistently implementing your data management plans. Good tools should support major features like these:

- Complete data-level security, so that you can maintain compliance with legislative or industry requirements that may relate to your test data
- The ability to extract, transform, and load data on demand to set up databases and other data stores for specific testing scenarios
- The ability to work across many types of data store, including different brands of database software, different file formats, and so forth
- The ability to generate specific types of common test data, such as customer names or random ID numbers
- Analyze test data and report on invalid values, uniqueness, and so forth
- Data modification, offering the ability to de-personalize personal information which may have been taken from a production database, while still maintaining the proper data formats. Figure 8.6 shows an example of how an application might do this: It shows potentially sensitive information like customer IDs and names, and specific "disguise" methods that de-personalize the data.

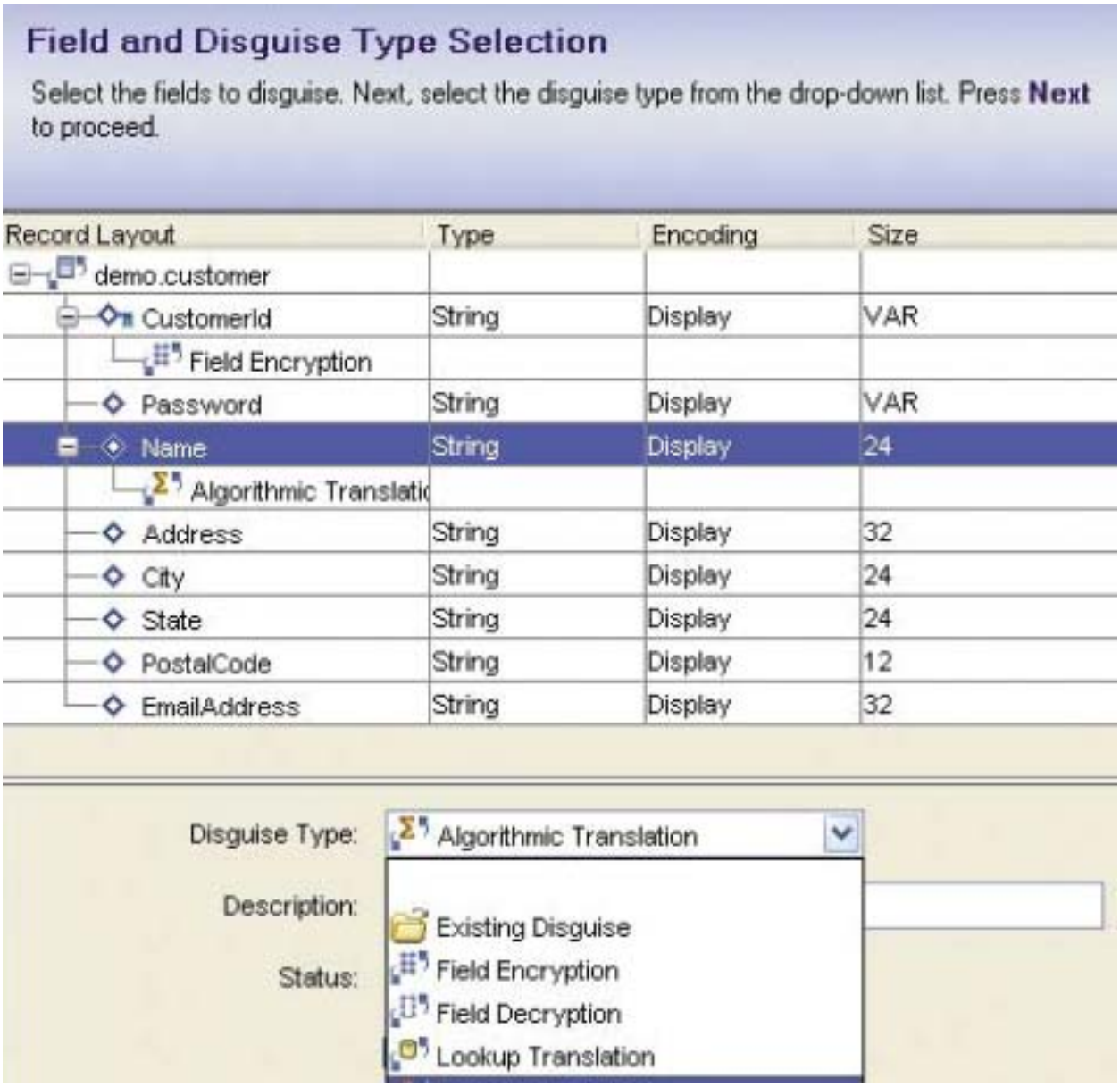

Figure 8.6: Disguising data to maintain privacy.

- Data aging, giving you the ability to increment values, dates, and other data automatically

I tend to find that data management tools fall into two categories: Those which provide a *good* feature set across a very broad range of databases and formats, and those which provide *excellent* features for perhaps a single relational database management product. In other words, as you get more specific with the database platform, the data management tool can offer more specific features. You'll have to look at your data environment and decide where the right tradeoff is for you.

It's Functional: What's Next?

Once you've verified the *function* of your software, and made sure that it has all the capabilities required, do you need to test anything else?

Absolutely. If you recall from Chapter 5, there are numerous "non-functional" requirements, chief of which are performance and security. These don't necessarily impact what the software does, but they most certainly impact how the software does it, and performance and security are, in particular, major contributors to what end-users and the business as a whole perceive as "quality" in an application. We'll make non-functional testing the subject of our next chapter.