The Developer's Guide to Test Automation

Agile Development Conference East
Better Software Conference East

Half-day Tutorial
Monday, November 11, 2013 — 1:00pm - 4:30pm

Dale Emery
@dhemery
http://dhemery.com/

George Dinwiddie
@gdinwiddie
iDIA Computing, LLC
http://blog.gdinwiddie.com/
http://idiacomputing.com/

Contents

Essential Vs. Incidental

I want an automated test not only to test a system responsibility, but also to *describe* the responsibility it is testing. To make tests expressive and maintainable, we must separate the *essential* details from the *incidental* ones. And to do that, we must know how to *distinguish* essence from incidentals.

Let's explore the idea of essence at two scopes: the essence of the system as a whole, and the essence of a given responsibility.

> **The essence of a planned response system** is the set of responsibilities allocated to the system.
>
> **The essence of a responsibility** is the obligation to respond to a specified event in a specified context by producing planned results.

The essence of a responsibility and the essence of the system are defined in terms of three concepts:

- *Event:* An occurrence of interest to the system.
- *Result:* An effect produced by the system.
- *Context:* A set of conditions inside and outside the system.

Essence is Independent of Implementation Technology

Note that none of the concepts that make up our definitions of essence – event, result, and context – depend on what technology we use to implement the system. And none of them constrain our choices of technology. This independence from technology is the key distinction between essential and incidental:

> **Essence is independent of implementation technology.**

Whenever we speak of the technology used to implement a system, we are speaking not of the essence of the system, but of incidental details.

With these distinctions in mind, we can define essential and incidental:

> **Essential:** Inherent in the definition of the responsibility.
>
> **Incidental:** Chosen to satisfy the implementation.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

A Subtle Distinction: Technology at the System Boundary

When I talk about essence being independent of implementation technology, I mean specifically *the technology inside the boundary of the system.*

Technology at the boundary. Sometimes the definition of the system's responsibilities requires the use of specific technology *at the boundary* of the system.

This is a subtle distinction that matters when our system has the *responsibility* to interact with other systems, either to sense signals from them or to send messages to them.

Sensing signals. Suppose the events that trigger our system's responsibilities occur inside upstream systems. When an event occurs in the upstream system, it sends some signal through some technology. Our system, in order to perform its resposibilities, must sense those signals. So *the boundary of our system* must use a technology that can sense the signals. The technology at the boundary of our system depends on the technology of the upstream system.

Sending messages. Suppose our system has the responsibility to send a message to another system. In this case, *the boundary of our system* must use a technology that can send a message to the downstream system. The technology at the boundary of our system depends on the technology of the downstream system.

Test Implementation and System Implementation

In general, we want our top-level test code (the code that directly expresses the responsibility being tested) not to refer to incidental details of technology. This of course includes references to the technology used to implement the system were testing. But we also want to avoid references to the lower-level technology we are using to implement the tests.

Techniques to Distinguish Essence from Incidentals

Here are some techniques I use to determine whether a detail in my test code refers to inessential technology.

Incidental By Definition

We can apply the definition of *incidental* directly by asking a few questions about any given detail:

> Does this detail refer to any element of the technology used to implement the system?

If so, the detail is an incidental detail. Common incidental details of this sort include buttons, text fields, screens, keyboard, mice, and any other references to elements of a graphical user interface (GUI).

> If we were to use a different test technology to implement this test, would this name still be appropriate?

Note the subtle difference between this question and the previos one. We're asking not whether the test code *refers* to an element of technology, but whether it *names* one. Every detail in an automated test must of necessessity refer to elements of test technology. So what starts to matter here is the names we use, and what ideas the names evoke in readers' minds.

To clarify, let's look at two small examples, each of which submits an expense item to an expense reporting system. Here's the first example:

```
selenium.sendKeys(ITEM_NAME, "Zeno's Diet Chocolate Brownies");
selenium.sendKeys(ITEM_PRICE), "$1.49");
selenium.click(SUBMIT);
```

This code uses the names `ITEM_NAME`, `ITEM_PRICE`, and `SUBMIT`, each of which nicely refers to an essential detail of the responsibility being tested.

The code also names two concepts that come from the choice of system technology: *typing keys* and *clicking.* Our earlier question would identify these as incidental details, so let's ignore them here.

There is one more name, a troublesome one: `selenium`. This name refers not to the system's implementation, or to an essential detail of the responsibility. It names *an element of test technology.* If we were to change our test code to use Sahi or WebRat instead of Selenium to interact with the system, this name would no longer be appropriate.

Here's a second example:

```
submitExpense("Zeno's Diet Chocolate Brownies", "$1.49");
```

This example does not mention Selenium or its methods. a Java method (defined somewhere) called `submitExpense`. The key here is that the "submit expense" is the name of a domain concept, and not (merely) a test technology concept.

So perhaps another way to phrase this second question:

The Fantasy of Perfect Technology

Another way to distinguish essence from incidentals is to indulge in **The Fantasy of Perfect Technology**. Imagine a system implemented using perfect technology. Then ask yourself some questions about the quality attributes of the system.

- *How fast would it respond?* If it were made of perfect technology, of course it would respond instantly, with zero delay.
- *How many users could use it at once?* An infinite number of users.
- *How much information could it store?* An infinite amount.
- *How often would it break?* It would never break.
- *How long does it take to start up?* None, because it's always on and always available.
- *How much energy would it use?* It would use no energy; heck, it might even generate energy for free.

The one glaring flaw of perfect technology is that it does not exist. Real-world technology is imperfect. That's what makes this exercise a fantasy. But it's a useful fantasy, because it helps us to separate the system's essential responsibilities from the temporary constraints of current technology.

Note that we apply the Fantasy of Perfect Technology only inside the boundary of the system. Even in our fantasy, the world outside of the system is made of real, imperfect stuff, with which the system will have to interact.

Now apply the fantasy to your own system. What responsibilities would your system have even if you could implement it using perfect technology? That set of responsibilities is your system's essence.

This suggests a subtle question that we can ask whenever our test code refers to some element of system technology:

> Is this element inside the boundary of the system?

If it is *absolutely required* in order to interact with some other system, then it may be *on the boundary* of our system. And it *may* be okay to cite it in the test. Maybe. This conclusion always warrants extra scrutiny.

If the element is not *absolutely required* by some external system, it is likely an incidental detail. Here is the final test:

> Is this element of technology perfect?

If the technology exists today in our universe, you know how to answer that one.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

The Dale's Brothers Heuristic

Imagine a warehouse full of parts for agricultural machinery – fertilizer sprayers, wood chippers, small tractors, and so on. Imagine that you are responsible for managing the inventory in the warehouse. Imagine that you had some kind of system that could help you manage the inventory.

General responsibilities. Here is a small sample of the things you might want the system to do for you:

- Record the location of each part.
- Record each transaction that moves parts into or out of the warehouse.
- Record how many of each part are currently in stock.
- Notify you when the stock for a given part falls below some threshold, so that you can order more.
- Identify the suppliers of each part.

Common features of implementation technology. If you were going to implement the system, there are some common features that the implementation technology would have to provide:

- A means of receiving input from the user and sending information to the user. The system must have an interface of some kind through which the system and its users can interact.
- An information processor. The system must have some means of, for example, calculating whether a transaction reduces the stock level of a part below the reorder threshold.
- A means of storing information. The system must have some means of remembering information, and later retrieving it for use in calculations and user interactions.

Essential tests. Suppose you had to write automated tests for this system? What tests would you write for such a system?

Before you answer, consider that you don't yet know anything about the technology that will be used to implement the system. So you will have to write your tests *with no knowledge of implementation technology.*

What tests would you write? How would you write them?

Possible implementation technologies. Now imagine a variety of technologies that you might use to implement such a system.

- A mainframe computer with text-based terminals.
- A Ruby on Rails web application.
- A Windows .NET application with a graphical user interface.
- A client/server system with a web service back end and mobile clients for iOS, Andriod,

and Google Glass.

H. L. Emery, Inc. My father ran such a warehouse in southern Maine. The manufacturers whose parts he warehoused repeatedly begged him to install their preferred high-tech inventory management systems. But my father was satisfied with the features of his current system, which he had used for thirty years.

My father's system was not computerized, but the technology he used to implement the system did have all of the features we listed above:

- **Information storage:** *Manila cards in a metal bin.*
- **Input and output.** *Pencils.*
- **Processor.** *My brothers Glenn and Gregg.*

The Dale's Brothers Heuristic. I uses a heuristic to help me notice when my tests make assumptions about implementation technology:

> How would I write this test if I knew absolutely nothing about the technology used to implement the system?

Here is another way to state the heuristic. I call it the *Dale's Brothers Heuristic*:

> How would I write this test if I did not know whether the system would be implemented by computers or by my brothers wielding pencils?

If you don't have brothers, you are welcome to borrow mine.

Super-intelligent magical monkeys. If the differences between computers and my brothers don't help you distinguish essence from incidentals, make up your own fanciful technnologies.

> How would I write this test if I did not know whether the system would be implemented:
>
> - by computers
> - by a horde of super-intelligent monkeys?
> - by a magical fairy with a magic wand?
> - by super-intelligent magical monkeys with magic wands?

Incidental details

In our tests, we need to make a decision about what details to highlight, and what details to hide. We want our tests to

- highlight the aspects they are checking
- communicate the rules being checked to someone who doesn't know them
- be obviously correct to someone who knows the rules.

Large amount of incidental details

When we have a large number of details, it's hard to pick out which ones are significant.

```
Scenario: Reimbursement for travel meals
  When I spend $3.49 for an egg sandwich at breakfast
  And I spend $1.49 for a cup of coffee at breakfast
  And I spend $0.30 for tax at breakfast
  And I spend $1.00 for tip at breakfast
  And I spend $7.99 for a ham sandwich at lunch
  And I spend $1.99 for french fries at lunch
  And I spend $1.99 for a soda at lunch
  And I spend $0.72 for tax at lunch
  And I spend $2.00 for tip at lunch
  And I spend $18.99 for meat loaf and mashed potatoes at dinner
  And I spend $1.99 for iced tea at dinner
  And I spend $6.50 for a glass of wine at dinner
  And I spend $1.65 for tax at dinner
  And I spend $5.00 for tip at dinner
  Then I should receive $43.00 in reimbursement
```

What is this test testing? Is it significant that I had an egg sandwich? Or that it cost $3.49? Are tax and tips important details? The **incidental details obscure the essential details**. When we read the test, we can't determine which is which.

Does this test represent the business requirements? I can't tell from this test what those requirements are. Someone who knows the requirements would likely have to get out a calculator to verify the test. It would be easy for an error in the calculation to go unnoticed.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Reducing incidental details

When we eliminate a lot of incidental details, the essential ones are more visible.

```
Scenario: Reimbursement for travel meals
  When I spend $6.28 for breakfast
  And I spend $14.69 for lunch
  And I spend $30.20 for dinner, including $6.50 for a glass of wine
  Then I should receive $43.00 in reimbursement
```

Now, this is better. Most of the details are gone. We can assume that, for this test, the amount for each meal, as well as which meal it is, is apparently important.

What about that glass of wine? What's significant about that? Could it be a bottle of wine? A glass of beer? A Pepsi-Cola?

```
Scenario: Reimbursement for travel meals
  When I spend $6.28 for breakfast
  And I spend $14.69 for lunch
  And I spend $30.20 for dinner, including $6.50 for alcohol
  Then I should receive $43.00 in reimbursement
```

In this case, the essential detail is that it's an alcoholic drink. What type of alcoholic drink is incidental. Having removed the incidental details, the business rule is discernible and the test is obviously correct.

Wait! Is that right? Where did the $43.00 come from? Maybe there are some essential details not shown.

When a lot of incidental details are supplied, it's harder to notice that essential details are missing. It's a classic case of not being able to see the forest because of all the trees in the way.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Making essential details explicit

In our previous example, some **essential details about the context of the test are hidden**. We cannot understand this test without knowing those. Rather than have to look elsewhere for this information, we can specify it right here.

Let's specify the reimbursement limits in the test. Then we can check the numbers.

```
Scenario: Reimbursement for travel meals
  Given the maximum reimbursement for breakfast is $5.00
  And the maximum reimbursement for lunch is $15.00
  And the maximum reimbursement for dinner is $25.00
  And alcohol is not reimbursable
  When I spend $6.28 for breakfast
  And I spend $14.69 for lunch
  And I spend $30.20 for dinner, including $6.50 for alcohol
  Then I should receive $43.00 in reimbursement
```

Do the numbers make sense now? Let's see… Breakfast is limited to $5.00; lunch is below the limit, and so is dinner if we subtract the glass of wine. So, $5.00 + $14.69 + $30.20 - $6.50 = $43.39. It takes some figuring, but that looks closer.

What happened to the 39 cents? Oh, sales tax is 6% and we're not getting reimbursed for the tax on the wine, either. I'll add that detail to the the test.

```
Scenario: Reimbursement for travel meals
  Given the maximum reimbursement for breakfast is $5.00
  And the maximum reimbursement for lunch is $15.00
  And the maximum reimbursement for dinner is $25.00
  And alcohol is not reimbursable
  And sales tax is 6%
  And sales tax on alcohol is not reimbursable
  When I spend $6.28 for breakfast
  And I spend $14.69 for lunch
  And I spend $30.20 for dinner, including $6.50 for alcohol
  Then I should receive $43.00 in reimbursement
```

Now we have our rules clearly defined. A little math shows that the test is checking for the correct answer. It does take some math, though. If one of these numbers was changed, would we notice the problem? If this test did not pass, would we know where our code might be wrong?

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Splitting out conditions

This test is testing a lot of conditions all at once. When the test indicates a problem, it could be a problem with any of these conditions. We would have to do some debugging or write some more tests to determine which condition had the problem.

If we split each condition into a separate test, we can determine where the problem lies by which test fails.

```
Scenario: Reimbursement limit for breakfast
  Given the maximum reimbursement for breakfast is $5.00
  When I spend $6.28 for breakfast
  Then I should receive $5.00 in reimbursement

Scenario: Reimbursement limit for lunch
  Given the maximum reimbursement for lunch is $15.00
  When I spend $14.69 for lunch
  Then I should receive $14.69 in reimbursement

Scenario: Reimbursement for dinner
  Given the maximum reimbursement for dinner is $25.00
  And alcohol is not reimbursable
  And sales tax is 6%
  And sales tax on alcohol is not reimbursable
  When I spend $30.20 for dinner, including $6.50 for alcohol
  Then I should receive $23.31 in reimbursement
```

When I do this, I notice that I'm testing only one condition for each meal. What about a test where lunch is over the reimbursement limit?

```
Scenario: Lunch under reimbursement limit
  Given the maximum reimbursement for lunch is $15.00
  When I spend $14.69 for lunch
  Then I should receive $14.69 in reimbursement

Scenario: Reimbursement limit for lunch
  Given the maximum reimbursement for lunch is $15.00
  When I spend $15.69 for lunch
  Then I should receive $15.00 in reimbursement
```

Repeating the **Given** condition in each test is a duplication I don't need. I intend for all of these to be the same, so I'll move it out to a common fixture setup. These are essential details

associated with the suite of tests for this fixture, rather than with the individual tests. We'll want to name the Feature to refer to the fixture setup.

```
Feature: Reimbursement rules and limits

    As a stingy middleman with a monopoly on corporate contracting
    I want to nickle and dime consultants with byzantine rules
    In order to minimize payouts and maximize profits.

    There are per-meal limits on the amount of reimbursement.
    Alcohol-related expenses are never reimbursable.

  Background:
    Given the maximum reimbursement for breakfast is $5.00
    And the maximum reimbursement for lunch is $15.00
    And the maximum reimbursement for dinner is $25.00
    And alcohol is not reimbursable
    And sales tax is 6%
    And sales tax on alcohol is not reimbursable

  Scenario: Reimbursement limit for breakfast
    When I spend $6.28 for breakfast
    Then I should receive $5.00 in reimbursement

  Scenario: Lunch under reimbursement limit
    When I spend $14.69 for lunch
    Then I should receive $14.69 in reimbursement

  Scenario: Reimbursement limit for lunch
    When I spend $15.69 for lunch
    Then I should receive $15.00 in reimbursement

  Scenario: Reimbursement for dinner
    When I spend $30.20 for dinner, including $6.50 for alcohol
    Then I should receive $23.31 in reimbursement
```

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Clarifying more conditions

We can continue to write scenarios for each condition, each meal both within and over the reimbursement limit, but we can also generate these variations from a table. Using a table makes it clearer to notice whether or not we've covered all the significant variations.

```
Background:
  Given the maximum reimbursement for breakfast is $5.00
  And the maximum reimbursement for lunch is $15.00
  And the maximum reimbursement for dinner is $25.00
  And alcohol is not reimbursable
  And sales tax is 6%
  And sales tax on alcohol is not reimbursable

Scenario outline: Reimbursement limits
  Given I spend <cost> on <meal>
  Then I should receive <amount> for <reason>

  Examples:
| meal      |  cost | amount | reason                |
| breakfast |  4.99 |  4.99  | breakfast under limit |
| breakfast |  5.01 |  5.00  | breakfast over limit  |
| lunch     | 14.99 | 14.99  | lunch under limit     |
| lunch     | 15.01 | 15.00  | lunch over limit      |
| dinner    | 24.99 | 24.99  | dinner under limit    |
| dinner    | 25.01 | 25.00  | dinner over limit     |

  Scenario outline: No reimbursement for alcohol
Given I spend <cost> on <meal>, including <alcohol_cost> for alcohol
Then I should receive <amount> for <reason>

  Examples:
| meal      |  cost | alcohol_cost | amount | reason                              |
| dinner    | 30.29 | 5.00         | 24.99  | dinner under limit without alcohol |
| dinner    | 29.25 | 4.00         | 25.00  | dinner over limit without alcohol  |
```

Do these examples describe the functionality more clearly? We can certainly see the effect of each limit individually.

Do we need examples that show the addition of multiple meals? What about alcohol with breakfast or lunch? Would those be treated the same as for dinner? Or might they result in

disciplinary action; we can't tell from these examples.

How do you feel about the <reason> field? This is included solely for commentary. It identifies the purpose of each example, and could be used by the step definition to better describe a failure. Otherwise it's not functional. If we wrote individual scenarios, the sceneario title would fill this need. Is there a better way to handle this need?

What might we be missing?

When I receive a bug report, I like to write a test that illustrates the bug. This lets me verify that I understand the conditions that exhibit the problem, and signals when the problem is fixed.

```
Scenario: Bug #24, reported 24Nov2009
  When I spend $6.28 for breakfast
  And I spend $14.69 for lunch
  And I spend $30.20 for dinner, including $6.50 for alcohol
  Then I should receive $43.00 in reimbursement
```

Often I will leave such a test in the suite. I *really* ***really*** hate it when I introduce the same defect I've fixed before. In this case, I'm pretty sure that my final set of scenarios cover all of the conditions that this bug-report scenario illustrates. But *could* there be an interaction between the meal limits? Could there be a problem with adding up a combination of within-limit and over-limit meal purchases, perhaps exacerbated by the calculation of sales tax on the alcohol?

On the other hand, if there are hundreds of these, they might slow the tests down considerably. In that case, I would probably want to delete it, or at least move it to a separate suite that is run less frequently.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Using Data Values in Automated Tests

We would like each automated test to describe its intentions, to describe the responsibility it tests. An important factor that affects the expressiveness of our tests is the way we select, name, and expose or hide the data values we use.

Question Literal Values

When I examine my clients' test code, I often see test steps like this:

```
logIn("j-fred-muggs");
```

This step uses the string literal `"j-fred-muggs"`. When I see literals in the code, my reaction is to wonder what makes that specific value important to the code. So I begin asking questions. And depending on the answers, I might suggest changing the code in some way. From the context I can guess that `"j-fred-muggs"` is a user name, but I don't know why this test is logging in as this user.

My first question:

> *Why that value?*

In this case: Why `"j-fred-muggs"`?

A common answer is that the test programmer needed to supply some value in order to log in, and knew that the system already had an account for `"j-fred-muggs"`. So that was the easiest one to use.

I interpret this to mean that the only important characteristic of this value is that it names a user for whom the system already has an account. More abstractly: This value is important because it is a member of some category that matters to the test. It matters that the user is an existing user. I test my interpretation by asking:

> Would *any* member of the category work just as well in this test?

For this test, the question is: Would *any* existing user work just as well? I'm checking to see whether there are any other important characteristics of this name. There are three possibilities here:

- **Any existing user would work just as well.** In this case, the important thing about our data value is that it is a *representative member* of some category.
- **Some other existing users would work just as well,** but some would not. In this case, our data value has some other characteristic that matters to the test, but we haven't yet

named the important characteristic. We will want to *refine the category*.

- **No other value would satisfy the purpose of the test.** In this case, our data value is a *magic literal*, a value that has a specific, unique meaning to the system we're testing.

Representative Members

If the important characteristic of a literal value is that it is a representative of some category, I ask a few more questions.

> Does this *category* matter to the responsibility I'm testing?

That is,

I have a few choices about what to do with the value:

- If the category matters to the responsibility we're testing, extract the value into a variable, name the variable to identify the category, and use the name in the test.
- If the category doesn't matter to the test, hide it inside a well-named helper method or object.

The category is *existing users*, so I will name the constant `EXISTING_USER`. Now the step looks like this:

```
logIn(EXISTING_USER);
```

This code expresses more clearly the intent of that step: Log in as an existing user.

I might go a step further. If our intention is to log in, we will of course have to log in as an existing user. So maybe we can remove that information entirely from the test step:

```
logIn();
```

In this case, I let the login method select a reasonable default value.

Have we lost anything by removing `EXISTING_USER` from the test code? Maybe. That depends on the intent of the test. Two key questions here are:

> Does the system have a responsibility to do something different for existing users than for other users?

and:

> Is the purpose of this test to test that responsibility?

If the test is otherwise written clearly, it will be apparent to us whether the test cares about the prior existence of the user. If it isn't obvious, we may find clues by reading nearby tests – other tests in the same test class, or other scenarios in the same Cucumber feature.

Suppose a second test logs in as `"phoebe-b-beebe"`. And through the same question and answer approach we refactor the login step in the second test to look like this:

```
logIn(NEW_USER);
```

We can now clearly see that one test logs in as a new user, and one as an existing user. The two tests differ in this way. Does this difference matter to the tests?

Suppose these two tests are trying to test that the system responds differently when a new user logs in than when an existing user logs in. When a new user logs in, the system must immediately ask the user to supply a new password.

In this case, it matters to the test whether the user is a new user or an existing one. Each test needs a member of a specific category.

Refining Categories

Sometimes we discover that a literal value represents a *subset* of a more general category. Suppose J. Fred Muggs is not only a user with an account, he is a *chimpanzee* with an account. And suppose the responsibility we are testing cares about the species of the user. Maybe we present one kind of user interface for chimpanzees, and a different kind of user interface for humans and other lesser primates. In that case, the variable name `EXISTING_USER` would not sufficiently convey the important distinction. We will need a name that conveys the distinction:

```
loginAs(A_CHIMPANZEE);
```

Before we settle on this name, we can ask a few other questions to find a name that says *everything that is important* about the value and *nothing that is unimportant*. The first question is one we have asked before, but now it becomes more specific:

> Would **any** chimpanzee do?

We call this *The Any Monkey Question.*[1] If the answer is no, we search for a further distinction. This may be a further subcategory, such as *bonobo*. Or we may need to introduce a separate dimension, such as *wild* or *in-captivity*.

If the answer is that, yes, any monkey will do, we can probe in the other direction, to see whether a more general category would suffice for our test. We can ask a few questions:

> Must it be a chimpanzee, specifically?

This can help us avoid using a category that is overly restrictive. Overly restrictive categories can confuse readers by suggesting constraints that the responsibility does not care about.

> What is a chimpanzee a kind of?

It's a kind of ape, which is a kind of primate, which is a kind of mammal. Would those more general categories suffice?

Magic Literals

Sometimes we choose a given literal value because the value has a specific, unique meaning to the system we're testing. Suppose, for example, we have a business rule that says:

> Preferred customers receive a 10% discount on purchases over $1000.

This business rule would create numerous responsibilities for the system:

- Apply a 10% discount to preferred customers' purchases over $1000 dollars.
- Do not apply the discount to non-preferred customers, even for purchases over $1000.
- Do not apply the discount to purchases of $1000 or less, even for preferred customers.

Each of these responsibilities involves one or more specific data values. How might we represent these values in a test? Here's one way:

```
Feature: Do not discount purchases of $1000 or less
  Given that the customer is a preferred customer
  When the customer makes a purchase of $1000
  Then the system does not apply the preferred customer discount

Feature: Discount preferred customers' purchases over $1000 by 10%
  Given that the customer is a preferred customer
  When the customer makes a purchase of $1100
  Then the system applies a preferred customer discount of $110
```

What advantages and disadvantates do you see in putting these specific values directly into the tests?

Some questions that I would ask: Can the reader understand what each magic literal means? Can the reader easily trace these magic literals to the business rules that give rise to them?

Should we try to remove these literal values from the tests, and instead use meaningful names? Let's try it and see what we think.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

The first step is to find meaningful names. What does each value mean? Let's call the 10% the *preferred customer discount*, and the $1000 the *preferred customer discount threshold.* (You may be able to think of better names.)

Now the tests look something like this:

```
Feature: Do not discount purchases at or below the preferred customer threshold
  Given that the customer is a preferred customer
  When the customer makes a purchase of exactly the preferred customer threshold
  Then the system does not apply the preferred customer discount

Feature: Discount preferred customers' purchases that exceed the threshold
  Given that the customer is a preferred customer
  When the customer makes a purchase that exceeds the threshold
  Then the system applies the preferred customer discount
```

Is this an improvement over the original tests with the magic values? In this case, I don't think so. Given that the business rule states the magic literals explicitly, it is easier for readers to map the test to the business rule if the test uses the literal values.

But suppose the business rule were stated this way:

> Preferred customers receive a discount on purchases that exceed the preferred customer discount threshold. The threshold and the amount of the discount are updated from time to time, and their current values are stored in the Customer Loyalty Policies database.

Now the business rule does not state particular values, and we can see that the values may change from time to time. In this case, I am more inclined to write the tests to use meaningful names from the business domain instead of literal values.

Relationships and Literal Values

In addition to understanding the individual steps in a test, we also want to understand the *relationships* among the steps.

Let's take another look at one of our preferred customer discount tests:

```
Feature: Discount preferred customers' purchases over $1000 by 10%
  Given that the customer is a preferred customer
  When the customer makes a purchase of $1100
  Then the system applies a preferred customer discount of $110
```

What do the numbers in this test have to do with each other? What does the `$1100` value have

to do with this test? The important thing about that value is that it exceeds the threshold at which the discount applies. That affects the next line, where we expect the system to apply the discount. Is the meaning of the value apparent in the test code? I think so, but judge for yourself.

What about the `$110`? What does that have to do with the rest of the feature? With a little calculation, we can see that $110 is 10% of $1100. That is, it is the dollar amount of the preferred customer discount for a purchase of $1100. Is that clear from the test? I wouldn't say that it's apparent *at a glance*, but I think this is clear to a typical person who understand the business rule.

For examples of magic literal values that are not only unclear, but which acutally *obscure* the meanings of tests, see the dollar amounts in the early examples in "Incidental Details" elsewhere in this handout.

For more about general idea of making relationships clear in test code, see "Tests as Examples of System Responsibilities" elsewhere in this handout.

When Are Literal Values Okay?

In his "Refactoring with Ben Orenstein" video[2], Ben Orenstein says:

> I never put a value other than 1 or 0 into the code without a name to it.

Though I'm tempted to give his strict rule a thorough tryout, I don't usually go quite so far as Ben. Sometimes I find that I'm okay with a literal value in the code.

The key for me is whether the literal value makes the meaning of the test clear, and whether the test code makes the meaning of the literal value clear.

In some of the examples above, the literal values are actually clearer than the best abstract names we can think of.

There is a danger here. It is very easy to conclude, when looking at test code that you yourself have written, that of course the meanings of the literal values are clear in the code. Before you conclude that a literal value communicates your intentions, consider showing your code to a few colleagues and asking:

- What makes this specific value important to the responsibility I'm testing?
- How does this specific value relate to other parts of the test?

If their answers are not what you intended, or if it takes a reader more than 10 seconds to give the right answer, your code is not yet clear enough. And if the meaning is unclear, refactor the test code by extracting the values into variables or methods that make the meaning clear.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

For more questions to assess the clarity of literal values, see “Tests as Examples of System Responsibilities” elsewhere in this handout.

Tests as Examples of System Responsibilities

I like to think of system responsibilities as consisting of three parts:

- A **stimulus** to which the system is obligated to respond
- The **results** that the system is obligated to produce in response to the stimulus
- The **context** in which the system is obligated to produce those results in response to that stimulus.

I want automated test code to make clear what responsibility it is testing. To do that, the test code will typically have to identify all three parts (context, stimulus, and results) and will have to describe them directly in the code.

Further, I want it to be obvious to readers of the code which code establishes the context, which code provides the stimulus, and which code evaluates the results.

Other Common Three-Part Models

Testers and test automators use a variety of terms and models to refer to these concepts. Here are some of the popular ones.

Three A's. William Wake offered the popular Three A's model: *Arrange, Act, Assert.* This maps nicely onto the parts of a responsibility. First the code *arranges* the context for the test. Then the code *acts* on the system by sending some stimulus, such as by calling a method. Finally, the code makes *assertions* about the results.

Gherkin. The Cucumber tool includes a simple testing language called Gherkin. With Cucumber, each test is called a feature, and each feature is made up of one or more steps. Each step typically begins with one of the three primary Gherkin keywords: *Given, When, Then.* When used as intended by the Gherkin language designers, these keywords also map nicely onto the parts of a responsibility. A *Given* step establishes context for the feature. A *When* step simulates some user command or request to the system, and stimulates the system to respond. A *Then* step evaluates a result produced by the system.

Parts of a Test. Testers sometimes think of tests in terms of *Setup, Action, Expectations* (or *Validation*). We perform some *setup* steps to establish the context. Then we take some *action* that we're trying to test, which stimulates the system to respond. Then we *validate* the results produced by the system by comparing them to our *expectations*.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

A Three-Part Checklist

You can use any of these models as a checklist, to help determine whether a given test describes all three parts of the responsibility it tests, and to help remember to describe all three parts. Your checklist might look like this:

- Does the test code clearly describe the relevant *context* for this responsibility?
- Does the test code clearly describe the *stimulus*?
- Does the test code clearly describe the desired *results*?

A Three-Part Test Code Template

Some of my clients use test code templates, like this:

```
@Test
public void test {
    // Arrange

    // Act

    // Assert
}
```

Describe the Responsibility

I prefer to use terms that focus on the system's responsibility instead of on the test. To my mind, the *arrange, act, assert* terminology of the Three A's focuses on what the *test* is doing. The *test* arranges something. The *test* acts. The *test* asserts something. Similarly, the common *setup, action, expectations* model focuses on the tester's activities.

I find that Gherkin's *given, when, then* scheme does a better job of directing my attention the system's responsibility. In Cucumber, the Gherkin keywords often allow me to write tests steps that describe the system responsibility clearly and directly:

```
Given that "Fred" reports to "Mr. Spacely"
When "Mr. Spacely" opens his direct reports org chart
Then the system displays "Fred" on the org chart
```

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Describe Each Part of the Responsibility

As you write each test, ask yourself whether the test code expresses each of the essential parts of the responsibility. If a part is missing, add it.

For examples of tests with and without each part, see “Essential Details” in this handout.

Highlight Relationships Among the Parts

The essence of a responsibility comes not only from its parts – the context, stimulus and results – but also from the *relationships* among the parts. How does the context affect the way the system must respond? How to the inputs influence the outputs?

The meaning of the test comes largely from these relationships. A well-written test *highlights* these relationships and makes them clear and obvious to the reader.

We want to see not only the specific details that differentiate this responsibility from other responsibilities, but also the *relationships* among these details.

For examples, see “Essential Details” in this handout.

What Interferes with Clarity

As we read test code, we want the essential details of the responsiblity not only to be *present*, but also to be *apparent*. We would like to be able to tell *at a glance* what responsibility each test is testing.

Even when all the parts are present, several factors can interfere with our ability to see them at a glance.

Incidental details. Incidental details in the code compete with essential details for the reader’s attention. Anything that competes with essence interferes with clarity.

We have plenty to say about incidental details elsewhere (and nearly everywhere) in this handout.

Overloaded parts. Complex setups and verifications can make it harder to see the essence of a test. Often an overloaded part is a sign that an important concept is lurking unnamed in the code.

Do these these three setup steps work together to establish a single condition? *Name the condition.* This makes the essence of the condition clearer and makes it easier to understand

the surrounding test code.

Do these three assertions work together to assess a single compound criterion? *Name the criterion.* This makes the essence of the criterion clearer and makes it easier to understand the surrounding test code.

Run-On Tests. Sometimes a test is really testing multiple responsibilities. See “Complex Tests” (in this handout) for examples. When we bunch multiple responsibilities into a single test, we make it harder to see the tests’ responsibilities at a glance. Further, the added bulk of the test code makes it harder to understand the code even with study.

Assessing Clarity and Coherence

Assessing the Setup. Identify the test steps that establish the context for the test. As you read these steps, note:

- The *conditions* established by the steps.
- The *data values* used to establish or describe the conditions.

For each **condition** established by the setup steps, ask:

- What makes this specific condition important to the test?
- How does this condition relate to the input values used in the stimulus steps?
- How does this condition affect the expected results?

For each **data value** used in a setup step, ask:

- What makes this specific value important to the test?
- How does this value affect the expected results?
- How does this value relate to the input values used in the stimulus steps?

Assessing the Stimulus. Identify the stimulus steps, the test steps that trigger the system to respond. As you read the stimulus steps, note:

- The *input mechanisms* used to trigger the system.
- The *input values* sent to the system as part of the stimulus.

For each **input mechanism** used in a stimulus step, ask:

- What makes this specific input mechanism important to the test?
- How does this mechanism relate to the conditions established by the setup steps?
- How does the use of this mechanism affect the expected results?

Are the answers clear in the code?

Copyright © 2003-2013 Dale Emery and George Dinwiddie

For each **input value** used in a stimulus step, ask:

- What makes this specific input value important to the test?
- How does this input value relate to the conditions established by the setup steps?
- How does this input value affect the expected results?

Assessing the Results. Identify the tests steps that verify the results produced by the system. As you read the results steps, note:

- The *assertions* made by the results steps.

For each **assertion**, ask:

- What result does this assertion verify?
- What makes this specific result imporant to the test?
- How is this result affected by the conditions established in the setup steps?
- How is this result affected by the input values used in the stimulus steps?

Assessing Code Clarity. These final questions may be the most important questions of all:

- ***Are the answers to these questions obvious in the code?***
- ***How do you know?***

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Essential Details

We discovered in our discussion of incidental details that it's easy to become distracted by them and not notice that we're missing essential details. It's also possible to hide essential details without that distraction.

Categories of Essential Details

As discussed in "Tests as Examples of System Responsibilities", there are three parts to the system responsibilities: conditions, stimulus, and results. All three of these parts generally have essential details that we want to make explicit. We also want to make clear the relationships between details in these three areas.

Explicit stimulus

If we were to write our scenario as

```
Scenario: Reimbursement for travel meals
  Given the maximum reimbursement for breakfast is $5.00
  And the maximum reimbursement for lunch is $15.00
  And the maximum reimbursement for dinner is $25.00
  And alcohol is not reimbursable
  And sales tax is 6%
  And sales tax on alcohol is not reimbursable
  When I spend $51.17 for meals, including $6.50 for alcohol
  Then I should receive $43.00 in reimbursement
```

then there would be insufficient essential details describing the stimulus. We can't determine if $43.00 is the correct reimbursement without knowing the amounts for individual meals.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Explicit results

```
Scenario: Reimbursement for travel meals
  Given the maximum reimbursement for breakfast is $5.00
  And the maximum reimbursement for lunch is $15.00
  And the maximum reimbursement for dinner is $25.00
  And alcohol is not reimbursable
  And sales tax is 6%
  And sales tax on alcohol is not reimbursable
  When I spend $6.28 for breakfast
  And I spend $14.69 for lunch
  And I spend $30.20 for dinner, including $6.50 for alcohol
  Then I should receive the correct reimbursement
```

This scenario does not specify what “correct reimbursement” means. It may be that, under the covers in the step definition, the correct value is indeed specified. That’s not helpful when looking at this test, though.

Explicit Conditions

When we specified our test as

```
Scenario: Reimbursement for travel meals
  When I spend $6.28 for breakfast
  And I spend $14.69 for lunch
  And I spend $30.20 for dinner, including $6.50 for alcohol
  Then I should receive $43.00 in reimbursement
```

then we failed to specify the conditions under which we would expect the answer $43.00 for these expenditures.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

We improved this by changing the scenario to

```
Scenario: Reimbursement for travel meals
  Given the maximum reimbursement for breakfast is $5.00
  And the maximum reimbursement for lunch is $15.00
  And the maximum reimbursement for dinner is $25.00
  And alcohol is not reimbursable
  And sales tax is 6%
  And sales tax on alcohol is not reimbursable
  When I spend $6.28 for breakfast
  And I spend $14.69 for lunch
  And I spend $30.20 for dinner, including $6.50 for alcohol
  Then I should receive $43.00 in reimbursement
```

In doing so, we defined our rules more clearly. A little math shows that the test is checking for the correct answer. It does take some math, though. If one of these numbers was changed, would we notice the problem? If this test did not pass, would we know where our code might be wrong?

Explicit relationships

We would like the essential relationships between values to be plain upon inspection. If we specify

```
Scenario: Reimbursement limit for breakfast
  Given the maximum reimbursement for breakfast is $5.00
  When I spend $6.28 for breakfast
  Then I should receive $5.00 in reimbursement
```

then the relationship between the reimbursement limit and the reimbursement is plainly visible. We can intuit the reason for the expected value. Similarly, in

```
Scenario: Reimbursement limit for lunch
  Given the maximum reimbursement for lunch is $15.00
  When I spend $14.69 for lunch
  Then I should receive $14.69 in reimbursement
```

we quickly notice the relationship between the stimulus and result. Again, the can understand the reasoning for the test by simple inspection, without any calculation.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Implementation Details

Workflow details

Sometimes I see tests written as if they're telling some remote programmer how to accomplish the test.

```
Scenario: Reimbursement for travel meals
  Given I login to the Expense Report system
  When I navigate to the Enter Expense Page
  And I enter "6.28" for "breakfast"
  And I enter "14.69" for "lunch"
  And I enter "30.20" for "dinner"
  And I submit the form
  And I print the form
  And I take the form to the client and get it signed
  And I photocopy the receipts
  And I fax the signed form and photocopied receipts to "800-555-1212"
  Then I should receive $43.00 in reimbursement within 3 months
```

Tests written in this way are often called ***imperative scenarios*** because they tell you what to do. Favor ***declarative scenarios*** that describe the system behavior. Not only do declarative scenarios make better ***executable specifications***, but they are easier to maintain. The precise procedure for telling the system how much I spent for lunch might change in the future. It's better that this detail be implemented once, and called by every step definition referring to lunch spending, than that it be repeated numerous times in multiple scenarios.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

System details

Even more fragile are tests like the following:

```
Scenario: Reimbursement for travel meals
  Given I navigate to http://example.com/
  And I click "//div/form//input[@id='login_name']"
  And I enter "j-fred-muggs"
  And I click "//div/form//input[@id='password']"
  And I enter "SuP3rS3kRiT"
  And I click "//div/form//input[@type='submit' and @value='Login']"
  And I navigate to http://example.com/expense_report
  And I click "//div/form//input[@name='description']"
  And I enter "breakfast"
  And I click "//div/form//input[@name='amount']"
  And I enter "6.28"
  And I click "//div/form//input[@type='submit' and @value='next']"
  And I click "//div/form//input[@name='description']"
  And I enter "lunch"
  And I click "//div/form//input[@name='amount']"
  And I enter "14.69"
  And I click "//div/form//input[@type='submit' and @value='next']"
  And I click "//div/form//input[@name='description']"
  And I enter "dinner"
  And I click "//div/form//input[@name='amount']"
  And I enter "30.20"
  And I click "//div/form//input[@type='submit' and @value='submit']"
  Then I should receive $43.00 in reimbursement within 3 months
```

This test is littered with references to the specific implementation, in this case, the document structure of a web page. It will only take the slightest change to the page, e.g., a small search form added to the top of the `div`, to break all of these XPATH expressions. Again, details such as this should be isolated to one place, and therefore made easy to change in the future. These implementation details also hide the intent of the test.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Driver details

I couldn't think of a good example of this problem using Cucumber, so I wrote a JUnit example, instead.

```
public static void testReimbursementForTravelMeals() {
  selenium.findElement(By.id("login_name")).sendKeys("j-fred-muggs");
  selenium.findElement(By.id("password")).sendKeys("SuP3rS3kRiT");
  selenium.findElement(By.xpath("//div/form//input[@type='submit' and @value='Login']").
  selenium.findElement(By.name("description").sendKeys("breakfast");
  selenium.findElement(By.name("amount").sendKeys("6.28");
  selenium.findElement(By.name("description").sendKeys("lunch");
  selenium.findElement(By.name("amount").sendKeys("14.69");
  selenium.findElement(By.name("description").sendKeys("dinner");
  selenium.findElement(By.name("amount").sendKeys("30.20");
  selenium.findElement(By.xpath("//div/form//input[@type='submit' and @value='submit']")
  assertEquals("43.00", selenium.findElement(By.xpath("//div[@class='reimbursement']//sp
}
```

Again we find our test crowded with incidental details that obscure the essential ones. In this case, the incidental details are those of the driver whe're using to connect our test to the system we're testing. It's better to keep the expression of the test driver-agnostic. Not only are we not testing the driver, itself, but we may want to change it for a different driver in the future.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Context setup

When we specify the ‘Given’ conditions that are assumed by the following scenarios (or later in the same scenario), we could be

- Stating conditions embodied in the system, either by code or configuration
- Verifying conditions embodied in the system, either by code or configuration
- Setting conditions within the system, either dynamically or by configuration.

For example:

```
Background:
  Given the maximum reimbursement for breakfast is $5.00
  And the maximum reimbursement for lunch is $15.00
  And the maximum reimbursement for dinner is $25.00
  And alcohol is not reimbursable
  And sales tax is 6%
  And sales tax on alcohol is not reimbursable
```

Stating conditions

If the background merely states conditions that we know to be true, we leave ourselves open to future problems. Sure, the maximum reinbursement for breakfast may be $5.00 now, but the penny-pinching middleman is likely to reduce it to $3.00 in the future. If this happens, our tests will start giving us incorrect results. I find this far too fragile to live with.

On the other hand, we might think that alcohol and associated sales tax is unlikely to become reimbursable. This business logic may be embedded in the code without a way to configure or verify it. In that case, having the background merely state our assumptions about the system may be reasonable for documentation purposes, even though it’s not functional.

```
Then(/^alcohol is not reimbursable$/) do
  # This policy is unlikely to change
end
```

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Verifying conditions

If a pre-condition is not settable, or it's not convenient to set it, we may still want to verify that our assumptions are true.

```
Then(/^sales tax is (\d+)%$/) do |tax_rate|
  expect(page.tax_rate).to eq tax_rate
end
```

This alerts us when, in the future, the tax rate changes. The error will tell us that the tax rate doesn't match our expectation, rather than leaving us puzzled about how we broke the functionality for calculating reimbursements.

Setting conditions

It's preferable if we can control our context.

```
Then(/^sales tax is (\d+)%$/) do |tax_rate|
  application.configure(:tax_rate=>tax_rate)
end
```

This way we can test the logic, without being concerned about the values being used in production.

One of these things is not like the other

The 6% sales tax is not "corporate policy" like the rest are. I have included it in the Background because it makes the calculations more understandable. I'm a little uncomfortable with this, though. I think that it not fitting with with the other items and the fact that the tests calculate the alcohol sales tax both contribute to this uneasiness.

In situations like this, sometimes I live with the discomfort until I find a better solution.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Complex Tests

Run-On Test

A client once complained to us, “But all of your examples are so small. Our tests are more complex than that. Why don’t you show an example of a *complex test*?” We asked for an example, but never got to see it. I suspect it was something functionally similar to this:

```
Scenario: Check the ability to control file access
  Given I login as an administrator
  And I set the file access to "read/write" for "j-fred-muggs"
  And I logout
  When "j-fred-muggs" logs in
  Then "j-fred-muggs" can read the file
  And "j-fred-muggs" can write the file
  And "j-fred-muggs" logs out
  And I login as an administrator
  And I set the file access to "read_only" for "j-fred-muggs"
  And I logout
  When "j-fred-muggs" logs in
  Then "j-fred-muggs" can read the file
  And "j-fred-muggs" cannot write the file
  And "j-fred-muggs" logs out
  And I login as an administrator
  And I set the file access to "deny" for "j-fred-muggs"
  And I logout
  When "j-fred-muggs" logs in
  Then "j-fred-muggs" cannot read the file
  And "j-fred-muggs" cannot write the file
  And "j-fred-muggs" logs out
```

This is called a **Run-On Test**, similar to a run-on sentence that your English teacher might have hated. It does one thing and then another and then another and when it fails, you have to discern what about it failed. Also, when it fails, you don’t have any information about later parts of the scenario, as they’re “blocked” by the failure.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

By breaking these into separate tests, we get clearer and more information in the case of a failure.

```
Scenario: Check read/write access
  When the file access is set to "read/write" for "j-fred-muggs"
  Then "j-fred-muggs" can read the file
  And "j-fred-muggs" can write the file

Scenario: Check read-only access
  When the file access is set to "read_only" for "j-fred-muggs"
  Then "j-fred-muggs" can read the file
  And "j-fred-muggs" cannot write the file

Scenario: Check denied access
  When the file access is set to "deny" for "j-fred-muggs"
  Then "j-fred-muggs" cannot read the file
  And "j-fred-muggs" cannot write the file
```

Sidebar: End to end testing

What does “End to end” testing mean to you? I’ve come across two main interpretations.

One is is a series of interactions with the system. Often this includes everything the user might do from logging on to logging off, and everything in between. Sometimes, as in our Run-On Test example, above, there are multiple users involved. This sort of test simulates real interaction with the system.

The other is a test that exercises system code from one external boundary to another. This might be from the GUI down to the database. This sort of test makes sure that the layers of subsystems work properly in concert.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Writing Maintainable Automated Acceptance Tests

Dale Emery[3]

Test Automation is Software Development

Test automation is software development[4]. This principle implies that much of what we know about writing software also applies to test automation. And some of the things we know may not be apparent to people with little or no experience writing software.

Much of the cost of software development is maintenance—changing the software after it is written. This single fact accounts for much of the difference between successful and unsuccessful test automation efforts. I've talked to people in many organizations that attempted test automation only to abandon the effort within a few months. When I ask what led them to abandon test automation, the most common answer is that the tests quickly became brittle and too costly to maintain. The slightest change in the implementation of the system—for example, renaming a button—breaks swarms of tests, and fixing the tests is too time consuming.

But some organizations succeed with test automation. Don't they experience maintenance costs, too? Of course they do. An important difference is that where unsuccessful organizations are surprised by the maintenance costs, successful organizations expect them. The difference between success and failure is not the maintenance costs per se, but whether the organization expects them. Successful organizations understand that test automation is software development, that it involves significant maintenance costs, and that they can and must make deliberate, vigilant effort to keep maintenance costs low.

The need to change tests comes from two directions: changes in requirements and changes in the system's implementation. Either kind of change can break any number of automated tests. If the tests become out of sync with either the requirements or the implementation, people stop running the tests or stop trusting the results. To get the tests back in sync, we must change the tests to adapt to the new requirements or the new implementation.

If we can't stop requirements and implementations from changing, the only way to keep the maintenance cost of tests low is to make the tests adaptable to those kinds of changes.

Developers have learned—often through painful experience—that two key factors make code difficult to change: Incidental details and duplication. You don't want to learn this the hard way.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Acceptance Tests and System Responsibilities

An acceptance test investigates a system to determine whether it correctly implements a given responsibility. The essence of an acceptance test is the responsibility it investigates, regardless of the technology used to implement the test.

Suppose we are testing a system's account creation feature. The `create` command creates a new account, given a user name and a password. One of the account creation feature's responsibilities is to *validate passwords.* That is, it must accept valid passwords and reject invalid ones. To be valid, a password must be from 6 to 16 characters long and include at least one letter, at least one digit, and at least one punctuation character. If the submitted password is valid, the `create` command creates the account and reports *Account Created*. If the password is invalid, the `create` command refrains from creating the account and reports *Invalid Password*.

That's the essence of the responsibility. No matter how the system is implemented—whether as a web app, a GUI app, a set of commands to be executed on the command line, or a guy named Bruce wielding a huge pair of scissors to snip off the fingers of anyone who submits an invalid password—the system must implement that responsibility.

Incidental Details

Listing 1 shows a poorly written automated acceptance test[5] for the `create` command's password validation responsibility.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Listing 1: A poorly written acceptance test

```
** Test Cases **
The create command validates passwords
    ${status}=  Run  ruby app/cli.rb create fred 1234!@$^
    Should Be Equal  ${status}  Invalid Password
    ${status}=  Run  ruby app/cli.rb create fred abcd!@$^
    Should Be Equal  ${status}  Invalid Password
    ${status}=  Run  ruby app/cli.rb create fred abcd1234
    Should Be Equal  ${status}  Invalid Password
    ${status}=  Run  ruby app/cli.rb create fred !2c45
    Should Be Equal  ${status}  Invalid Password
    ${status}=  Run  ruby app/cli.rb create fred !2c456
    Should Be Equal  ${status}  Account Created
    ${status}=  Run  ruby app/cli.rb create fred !2c4567890123456
    Should Be Equal  ${status}  Account Created
    ${status}=  Run  ruby app/cli.rb create fred !2c45678901234567
    Should Be Equal  ${status}  Invalid Password
```

This test has numerous problems, the most obvious being that it is hard to understand. We can see from the second line—the name of the test—that it tests the `create` command's validation responsibility. But it's hard to make sense of the details of the test among the flurry of words and "syntax junk" such as dollar signs and braces.

With a little study we can pick out the passwords—such as `1234!@$^`. And with a little more study we might notice that some passwords lead to a status of *Invalid Password* and others lead to *Account Created*. On the other hand, we might just as easily not notice that, because the connection between passwords and statuses is buried among the noise of the test. What do dollar signs, braces, and the words `Run`, `Ruby`, and `fred` have to do with passwords and validation? Nothing. Those are all *incidental details*, details required only because of the way we've chosen to implement the system and the test.

Incidental details destroy maintainability. Suppose our security analysts remind us that six-character passwords are inherently insecure. So we change one of the key elements of the responsibility, increasing the minimum length of a password from six to ten. Given this change in requirements, what lines of this test would have to change, and how? It isn't easy to see at a glance.

Let's consider a more challenging requirements change. We want system administrators to be able to configure the minimum and maximum password length for each instance of the system. Now which lines of the test would have to change? Again, the answer isn't easy to see at a glance.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

That's because the test does not clearly express the responsibility it is testing. When we cannot see the essence of a test, it's more difficult and costly to understand how to change the test when the system's responsibilities change. Incidental details increase maintenance costs.

So the first step toward improving maintainability is to hide the incidental details, allowing us to more easily see the essence of the test. In this test, most of the details are about how to invoke the `create` command. This system is implemented as a set of command line commands, written in the Ruby programming language. The first highlighted line in the test tells Robot Framework to run the computer's Ruby interpreter, telling it to run the `app/cli.rb` file (the system we're testing), and telling *it* in turn to run its `create` command with the user name `fred` and the password `1234!@$^`. And at the end of it all, Robot Framework stuffs the `create` command's output in a variable called `${status}`. Whew!

The highlighted second line is easier to understand. It compares the returned status to the required status `Invalid Password`. But it's awkwardly worded and includes distracting syntax junk, a form of incidental detail.

Robot Framework allows us to extract details into *keywords*, which act like subroutines for our tests. A keyword defines how to execute a step in an automated test.

So let's create a keyword to hide some of the incidental details.

One useful approach is to ask yourself: How would I write that first step if I knew nothing about the system's implementation? Even if I knew nothing about the system's implementation, I know it has the responsibility to `create` accounts — that's the feature we're testing, after all. So I know it will offer the user some way to create an account. *Create Account*, then, is an essential element of the system's responsibilities. I also know (from other requirements) that in order to create an account, the user must submit a user name and a password.

Given all of that, I might write the test step like this:

```
Create Account fred 1234!@$^
```

I still have some concerns with this test step[6], but I'll deal with those later.

Now let's look at the second highlighted step. It seems to be verifying that the `create` command returned the appropriate status: *Invalid Password*. How might I rewrite this step if I knew nothing about the system's implementation? Here's one possibility:

```
Status Should Be Invalid Password
```

So together, those two steps now look like this:

```
Create Account fred 1234!@$^
Status Should Be Invalid Password
```

That's much clearer. Without all of the incidental details, it's easier to spot the connection between the two lines: The system must tells us that the given password is invalid.

Now if we try to run the test, it will fail, because Robot Framework doesn't know the meaning of `Create Account` or `Status Should Be`. We haven't defined those keywords yet. Let's do that now:

Listing 2: Keywords to create an account and check the status

```
** Keywords **
Create Account ${user_name} ${password}
    ${status}=  Run  ruby app/cli.rb create ${user_name} ${password}
    Set Test Variable   ${status}

Status Should Be ${required_status}
    Should Be Equal  ${status}  ${required_status}
```

The highlighted line introduces a new keyword called `Create Account`, and describes it as taking two pieces of information as input – a user name and a password. The next two lines tell Robot Framework how to execute the keyword. Notice that the first indented line looks a lot like the first highlighted line of our original test. This is where we hid the incidental details.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

You may also notice that we introduced yet more syntax junk, yet more dollar signs and braces. How is this an improvement? The benefit is this: By extracting all of the incidental details out of the test steps and into the keyword, we've cleaned up our test steps, making them easier to understand. The benefit becomes more apparent if we rewrite *all* of our test steps using the new keywords:

Listing 3: The test rewritten to remove incidental details

```
** Test Cases **
The create command validates passwords
    Create Account fred 1234!@$^
    Status Should Be Invalid Password
    Create Account fred abcd!@$^
    Status Should Be Invalid Password
    Create Account fred abcd1234
    Status Should Be Invalid Password
    Create Account fred !2c45
    Status Should Be Invalid Password
    Create Account fred !2c456
    Status Should Be Account Created
    Create Account fred !2c4567890123456
    Status Should Be Account Created
    Create Account fred !2c45678901234567
    Status Should Be Invalid Password
```

Now our test reads much more cleanly. At the expense of a little bit of syntax awkwardness in the keyword definition, we've gained a lot of clarity in the test. It's a tradeoff well worth making.

Duplication

So far we've improved the test noticeably by extracting incidental details into reusable keywords. But there are still problems. One, mentioned earlier, is the troublesome `fred` in every other step. A bigger problem is duplication. Every pair of lines submits an interesting password and verifies that the system emits the appropriate status message. From one pair to the next, only two things change: the password and the desired status. Everything else stays the same. Everything else is duplicated from one pair to the next.

Duplication destroys maintainability. Suppose our usability analysts remind us that none of our other systems ask users to create an account. Instead, they ask users to *register*. So the language of this system – *create account* – is inconsistent with others. The usability analysts insist, and now we need to change our system's terminology.

One possibility is to simply change the name of the command line command from *create* to *register*, and leave our tests the way they are. If we were to do that, then every time we tried to talk about the acceptance tests with the business folks, we would have to translate between the language of the tests and the language of the business. That path leads to confusion.

To keep the language consistent, it would be better to change the tests to use the common terminology. This is where duplication rears its ugly head. We have to scan all of our tests, identify every mention of *create*, and change it to *register*. With our revised test, that's not especially onerous. We mention *create* only ten times: eight[7] times in the test and twice in the keywords. But imagine if we had hundreds of tests[8]. Duplication increases maintenance costs.

Duplication often signals that some important concept lurks unexpressed in the tests. That's especially true when we duplicate not just single steps, but *sequences* of steps. In our test, we duplicate pairs of steps – one step in each pair creates an account with a significant password, and the next checks to see whether the system reported the correct status.

Consider the first two steps in Listing 3. What do they do? What is the essence of those two steps? Taken together, they verify that the `create` command rejects the password `1234!@$^`. How about steps nine and ten? Those two steps verify that the `create` command accepts the password `!2c456`. *Accept* and *reject*. Those concepts are the essence of the responsibility we're testing, yet they're cowering in the shadows of our test steps.

Let's make the concepts explicit by creating two new keywords[9]:

Listing 4: Keywords for accepting and rejecting passwords

```
** Keywords **
Accepts Password ${valid_password}
    Create Account arbitraryUserName ${valid_password}
    Status Should Be Account Created

Rejects Password ${invalid_password}
    Create Account arbitraryUserName ${invalid_password}
    Status Should Be Invalid Password
```

These keywords not only allow us to rewrite our test, they also define the *meaning* of accepting and rejecting passwords. To accept a password means that when we try to create an account with the password, the system reports that the account has been created[10]. To reject a password means that when we try to create an account with that password, the system reports that the password is invalid.

Now we can rewrite our test to reduce the duplication, and also to directly express the essential responsibility of accepting and rejecting passwords[11]:

Listing 5: Test rewritten to reduce duplication

```
** Test Cases **
The create command validates passwords
    Rejects Password 1234!@$^
    Rejects Password abcd!@$^
    Rejects Password abcd1234
    Rejects Password !2c45
    Accepts Password !2c456
    Accepts Password !2c4567890123456
    Rejects Password !2c45678901234567
```

By analyzing duplication in the test, we identified two essential system concepts – the system accepts valid passwords and rejects invalid ones. By defining keywords, we *named* those concepts. Then we rewrote the test to refer to the concepts by name. By putting names to those concepts, and using the names throughout the test, we made the test more understandable and thus more maintainable.

Naming the Essence

Now that the test more clearly talk about accepting and rejecting passwords, one last bit of unclarity becomes more apparent. As we look at each invalid password, it isn't immediately obvious what's invalid about it. And what about the valid passwords? Why do we test two passwords? And why those two? What's so special about them? With time you could figure out the answers to those questions. But here's a key point: *Any* time spent puzzling out the meaning and significance of a test is maintenance cost. This may seem like a trivial cost, but multiply that by however many tests you need to change the next time someone changes a requirement. As many of my clients have discovered, these "trivial" maintenance costs add up, and they kill test automation efforts.

As I designed the test, I chose each password for a specific purpose. The essence of each password is that it tells me something specific that I want to know about the system. Take the password `1234!@$^` as an example. I chose this password because it is missing one of the required character types: it contains no letters. The essence of this password is that it lacks letters.

I'd like to give that essence a name. Robot Framework offers a feature to do that: variables. I can create a variable, give it an expressive name, and assign it a value that embodies that name. Here's how to create a variable:

```
** Variables **
${aPasswordWithNoLetters}   1234!@$^
```

Now I can use that variable in my test. In the interest of space, let's assume that I've created variables for all of the passwords, each named to express its essence, its significance in the test[12]:

Listing 6: Test rewritten to name significant values

```
** Test Cases **
The create command validates passwords
    Rejects Password ${aPasswordWithNoLetters}
    Rejects Password ${aPasswordWithNoDigits}
    Rejects Password ${aPasswordWithNoPunctuation}
    Rejects Password ${aTooShortPassword}
    Accepts Password ${aMinimumLengthPassword}
    Accepts Password ${aMaximumLengthPassword}
    Rejects Password ${aTooLongPassword}
```

Now the test is nearly as clear as we can make it. I'll take one more step, and break the test into multiple tests, each focused on a particular element of password validation:

Listing 7: Test rewritten to name significant values

```
** Test Cases **
Rejects passwords that omit required character types
    Rejects Password    ${aPasswordWithNoLetters}
    Rejects Password    ${aPasswordWithNoDigits}
    Rejects Password    ${aPasswordWithNoPunctuation}

Rejects passwords with bad lengths
    Rejects Password    ${aTooShortPassword}
    Rejects Password    ${aTooLongPassword}

Accepts minimum and maximum length passwords
    Accepts Password    ${aMinimumLengthPassword}
    Accepts Password    ${aMaximumLengthPassword}
```

Now when I read these tests, I can understand *at a glance* the meaning and significance of each test and each step. Each important requirements concept is expressed clearly, and expressed once.

Now suppose we change the requirements for minimum and maximum password length. Because each requirements concept is expressed clearly in the tests, I can quickly identify which tests would have to change. And because each concept is defined once—and given a name—I can quickly change the tests.

Putting the Tests to the Test: A Major Implementation Change

So all of our work has made the tests more adaptable to requirements changes. But how about implementation changes? To find out, let's change a few implementation details of the system and see how our tests fare. By "a few implementation details," I mean let's rewrite entire system as a web app. Now, instead of typing the `create` command on the command line, users visit the account creation web page, type the user name and password into text fields on a web form, and click the *Create Account* button. And the system, instead of printing the status to the command line, forwards the user to a web page that displays the status.

The big question: How would our tests have to change?

Remember that earlier we hid many incidental details inside keywords – `Create Account` and `Status Should Be`. Those keywords still contain the arcane steps to issue commands on the command line. So clearly those keywords will have to change. Let's rewrite those keywords to invoke the web app instead of the command line app:[13]:

Listing 8: Rewriting keywords to invoke the new web app

```
Create Account ${username} ${password}
    Go To    http://localhost:4567/create
    Input Text    username    ${username}
    Input Text    password    ${password}
    Submit Form

Status Should Be ${required_status}
    ${status}=          Get Text    status
    Should Be Equal     ${required_status}  ${status}
```

Okay, so we've changed the keywords that directly interact with the system. And we've added another eleven lines of test code as described in the footnote. What's next? What else do we have to change?

Nothing. We're done.

We've changed a few lines of test code, and our tests now run just fine against a new implementation of the system using entirely changed technology.

Meanwhile, Back in the Real World

In the real world, you will likely have more work to do to respond to such a major implementation change. For example, you will have to change more than two keywords. But if you've created low-level keywords that isolate the rest of your test code from the details of how to interact with the system, you will have to change only those low-level keywords. The tests themselves continue to work, unchanged.

And real world implementation changes may require more radical changes in the tools you use to run the tests.

But even when that's true, you can still use modern open source testing tools[14] to remove duplication from your tests, and to write tests that clearly and directly express the essence of the system responsibilities they are testing.

The bottom line is this: If you write automated tests so that they express system responsibilities clearly and directly, and if you remove duplication, you will significantly reduce the maintenance costs that arise from both changes in requirements and changes in system implementation. That could mean the difference between successful test automation and failure.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Four Layers in Automated Tests

Dale Emery[3]

I've known for a while that when I automate tests, layers emerge in the automation. Each chunk of automation code relies on lower-level chunks. In Robot Framework[15], for example, tests invoke "keywords" that themselves invoke lower-level keywords.

The layering *per se* wasn't a surprise, because automated tests are software, and software tends to organize into layers. But lately I've noticed a pattern. The layers in my automated tests center around four themes:

- Test intentions
- System responsibilities
- Essential system interface
- System implementation interface

Test intentions

Test names and suite names are the top layer in my automation. If I've named each test and suite well, the names express my test intentions. Reading through the test names, and seeing how they're organized into suites, will give useful information about what I tested and why.

For example, in my article on "Writing Maintainable Automated Acceptance Tests"[16] I was writing tests for a system's account creation feature, and specifically for the account creation's responsibility to validate passwords. I ended up with these test names (see Listing 7):

- Rejects passwords that omit required character types
- Rejects passwords with bad lengths
- Accepts minimum and maximum length passwords

In an excellent video followup[17] to my article, Bob Martin organized his tests differently, using Fitnesse[18]. He grouped tests into two well-named suites, "Valid passwords" and "Invalid passwords." Each suite includes a number of relevant example passwords, each described with a comment that expresses what makes the example interesting.

Every test tool that I've used offers at least one excellent way to express the intentions of each test. However expressed, those intentions become the top layer of my automated tests.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

System responsibilities

A core reason for testing is to learn whether the system meets its responsibilities. As I refine my automation, refactoring it to express my intentions with precision, I end up naming specific system responsibilities directly.

In my article, I'm testing a specific pair of responsibilities: The account creation command must accept valid passwords and reject invalid ones. As I refactored the duplication out of my initial awkward tests, these responsibilities emerged clearly, expressed in the names of two new keywords: Accepts Password and Rejects Password. Listing 7 shows how my top-level tests build on these two keywords.

Essential system interface

By system interface, I mean the set of messages that the system sends and receives, whether initiated by users (e.g. commands sent to the system) or by the system (e.g. notifications sent to users).

By essential I mean independent of the technology used to implement the system. For example, the account creation feature must offer some way for a user to command the system to create an account, and it must include some way for the system to notify the user of the result of the command. This is true regardless of whether the system is implemented as a command line app, a web app, or a GUI app.

As I write and refine automated tests, I end up naming each of these essential messages somewhere in my code. In my article, Listing 2 defines two keywords. "Create Account" clearly identifies one message in the essential system interface. Though the other keyword, "Status Should Be," is slightly less clear, it still suggests that the system emits a status in response to a command to create an account. (Perhaps there's a better name that I haven't thought of yet.) Listing 4 shows how the higher-level system responsibility keywords build upon these essential system interface keywords.

System implementation interface

The bottom layer (from the point of view of automating tests) is the system implementation interface. This is the interface through which our tests interact most directly with the system. Sometimes this interaction is direct, e.g. when Java code in our low-level test fixtures invoke Java methods in the system under test. Other times the interaction is indirect, through an intermediary tool, e.g. when we use Selenium[13] to interact with a web app or FEST-Swing[19] to interact with a Java Swing app.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

In my article, I tested two different implementations of the account creation feature. The first was a command line application, which the tests invoked through the “Run” keyword, an intermediary built into Robot Framework. Listing 2 shows how the Create Account keyword builds on top of the Run keyword (though you’ll have to parse through the syntax junk to find it).

The second implementation was a web app, which the tests invoked through Robot Framework’s Selenium Library[20] an intermediary which itself interacts through Selenium, yet another intermediary. Listing 8 shows how the revised Create Account keyword builds on various keywords in the Selenium Library.

Translating Between Layers

Each chunk of test automation code translates an idea from one layer to the next lower layer. Listing 7 shows test ideas invoking system responsibilities. Listing 4 shows responsibilities invoking messages in the essential system interface. Listings 2 and 8 show how the essential system interface invokes two different system implementation interfaces.

Each of the acceptance test tools I use allows you to build layers like this. In FitNesse, top-level tests expressed in test tables may invoke “scenarios,” which are themselves written in FitNesse tables. And scenarios may invoke lower-level scenarios. In Cucumber[21] top-level “scenarios” invoke “test steps," which may themselves invoke lower-level test steps. In Twist[22] “test scenarios” invoke lower-level “concepts” and “contexts.” Each tool offers ways to build higher layers on top of lower layers, which build upon yet lower layers, until we reach the layer that interacts directly with the system we’re testing.

In the examples in my article, I chose to write all of my code in Robot Framework’s keyword-based test language. I defined each keyword entirely in terms of lower-level keywords. I could have chosen otherwise. At any layer, I could have translated from the keyword-based language to a more general purpose programming language such as Java, Ruby, or Python. The other tools I use offer a similar choice.

But I, like many users, find these tools’ test languages easier for non-technical people to understand, and sufficiently flexible to allow users to write tests in a variety of ways. In general, I want as many of these layers as possible to be meaningful not just to technical people, but to anyone who has knowledge of the application domain. So I like to stay with the tool’s test language for all of these layers, switching to a general purpose programming language only at the lowest layer, and then only when the system’s implementation interface forces me to.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

A Lens, Not a Straightjacket

When I write automated tests for more complex applications, there are often more layers than these. Yet these four jump out at me, perhaps because each represents a layer of meaning that I always care about. Every automated test suite involves test ideas, system responsibilities, the essential system interface, and the system's implementation interface. Though other layers arise, I haven't yet identified additional layers that are so universally meaningful to me.

These layers were a discovery for me. They offer an interesting way to look at my test code to see whether I've expressed important ideas directly and clearly. I don't see them as a standard to apply, or a procrustean template to wedge my tests into. They are a useful lens.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

What Do You Want From Tests?

Dale Emery[3]

Automated tests are software. At first glance, this seems like a non-blinding non-flash of non-insight. But I'm learning a lot about testing by applying this non-insight mindfully.

One thing I'm learning is how often I forget that automated tests are software. When I'm writing tests, I often neglect to apply all of the principles help me to write software well. What if I were to apply some of those principles mindfully?

A key principle is that **we write software in order to serve some specific set of needs for some specific set of people**. When I'm trying to understand what software to write, I apply this principle in the form of a few questions: *Whose needs will the software serve? What needs will trigger those people to interact with the software? What roles will the software play in satisfying those needs?*

Let's apply this principle to the tests we write: **Whose needs will these tests serve? What needs would trigger those people to interact with the tests? What roles will the tests play in satisfying those needs?**

These days, I write software mostly for my own needs. And mostly I write the software alone. So the "whose needs" question is an easy one: When I write tests, I'm writing them mostly for me, for my own needs.

More enlightening for me — as a solo software developer writing tests solely for my own needs — are other questions. What needs trigger me to interact with the tests, either by running them or by reading test code? What roles do the tests play in satisfying those needs? Here's a partial list of answers:

I want to *know whether my software is ready to deliver.*

- I want test code to **help me understand which parts of the system are tested and which are not.**

I want to *know whether there are defects in the software I'm writing.*

- I want tests to **expose defects.**

I want to *know how to correct defects.*

- I want tests to **direct me to the defective part of the software.**

I want to *understand the meaning of the test results.*

- I want each test's code to **indicate clearly how the test stimulates the software, and in what conditions.**
- I want test reports to **describe the test stimulus, the relevant test conditions, and the software's response.**

When I'm adding a feature, I want to *know when I'm done.*

- I want tests to **tell me which of the feature's responsibilities the software fulfills, and which it does not.**

When I'm editing software, I want to *know whether my edits are having unintended effects.*

- I want tests to **detect changes in the behavior of the surrounding software.**

When I'm preparing to edit software, I want to *know what the existing code does*, so that I don't inadvertently break it.

- I want test code to **describe clearly what the existing software does.**

That's a partial list needs for a single stakeholder. I'm sure you can think of additional needs that you have when you run tests or read test code, and additional ways that you want tests to help you satisfy those needs. And if we were to consider other people who might interact with our tests, we would discover even more needs. And then there are all of the people who do not interact with the tests and yet are affected by them.

That's a lot of stakeholders, and a lot of needs. I'm more likely to satisfy all of these people's needs (including my own) when I'm aware of what the needs are. And I'm more likely to be aware of the needs when I ask questions like the ones I've used here. And I'm more likely to ask these questions when I remember that tests are software.

What do *you* want from tests?

Copyright © 2003-2013 Dale Emery and George Dinwiddie

What Do You Want From a Diagnosis?

When the system fails a test, we would like the failure message to help us correct the problem at minimal cost. In his book *Principles of Software Engineering Management*, Tom Gilb identifies ten distinct activities involved in the process of correcting software errors:

- Recognize the problem
- Assign someone to correct the fault
- Collect tools
- Analyze the problem
- Propose a correction
- Inspect the proposed correction
- Implement the correction
- Test the implementation
- Evaluate the test results
- Recover from the effects of the errors

Each of these activities costs time, money, and effort. The first activity, recognizing the problem, is one of the primary purposes of running tests. We would like to discover the problems in our software before they cause damage.

Once we recognize that the system has a problem, the next challenge is to analyze the problem, to understand the problem so we can correct it. We want to trace the problem from its symptoms to its source.

Understanding the problem is often the most costly and variable part of the correction process. To help reduce these costs, and potentially the costs of the subsequent activities, we test automators can make the effort to gather potentially useful information and present it in ways that can guide us from symptoms to source.

Failures and Faults

When I'm analyzing a problem in software, I find it helpful to distinguish between failures and faults.

A **failure** is an observed difference between the results we want and the results the system produced. This is the starting point of our analysis. We have observed a failure, and we don't know the cause of the failure. Actually, at this point we don't know which system is failing – the system we're testing, the test system itself, or some other system that affects the test results. All we know is that something has produced a result we don't want.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

A **fault** is a defect in the system that, under certain conditions, produces a failure. When we analyze a problem, our goal is to identify the fault so we can fix it.

A third helpful distinction is **conditions**. Under what *conditions* does the fault lead to a failure? Often in our analysis we experiment by changing the conditions, observing the results (perhaps including additional failures), forming hypotheses and models about the system, and changing conditions again.

Information From Automated Tests

To trace a failure to a fault, we need information. Some of the information we want comes in the form of relatively static descriptions of the system – for example, requirements documentation, design diagrams, and source code. Other information comes from running our tests. We can write tests to provide some of the information we need, in a way that helps us understand failures.

Our automated tests may not be able to point directly to the fault, but they can describe three very important things: The **responsibility** being tested, the **failure**, and the **interesting events** that occurred during the test.

Describe the Responsibility

The test code is part of the diagnostic message. That's one good reason to make your test code as clear and descriptive as it can be. Here are some ways to make your test code more useful for analyzing failures.

Test Suite Name

Different test tools offer different ways to specify suites. Unit testing tools typically use object packaging mechanisms from their native programming languages specifically classes and packages, to organize tests into suites.

Cucumber uses *feature* files. Robot Framework uses your disk's directory structure to organize tests into suites.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

In Rspec, you use *describe* blocks to identify the things being tested, and *context* blocks to identify the context in which the thing is being tested. And these blocks can be nested:

```
describe Stack do
    context "when the stack is empty" do
        describe "#push" do
            # Tests for push with an empty stack go here
        end
        describe "#pop" do
            # Tests for pop with an empty stack go here
        end
    end
end
```

Each of these *describe* and *context* blocks identifies a suite of tests. The outer *describe* contains all of the tests for the `Stack` class. The *context* block is a suite that contains all of the tests for an empty stack. The inner *describe* blocks each contain the tests for what a particular method must do when called with an empty stack.

Whatever mechanism your test tool uses, most test tools display the suite name in some way when tests fail.

Name your test suites so that they identify the general area of the responsibility tested by the tests in the suite. This ensures that the tool's display indicates which areas of responsibility were satisfied and which produced the failures. This information can quickly guide problem solvers to the relevant code in the system.

Test Name

Different test tools use different mechanisms to specify individual tests. Unit testing tools typically specify tests as individual *methods*. And most unit testing tools identify test methods by means of a naming convention (e.g. *methods whose names begin with* `test`) or by some form of annotation.

In Cucumber, each *feature* is an individual test. In Rspec, you use an *it* statement to write an individual test.

Name each test so that it identifies the specific responsibility being tested. Often you can use the suite name to identify the component being tested, so individual test names need not include this information. In *Naming Unit Tests* I describe how to name tests to indicate the *context, stimulus, and result*.

Sometimes you can put the context into the suite name, so that you need not include the

information in the individual tests. The Rspec syntax lends itself nicely to this idea. You can put the context in a *context* block, and the individual stimuli and results in the *it* blocks. For example:

```
describe Stack do
    describe "#push" do
        context "when the stack is empty" do
            it "places the pushed item on the top of the stack" do
                ...
            end
        end
    end
end
```

If this spec fails, the failure message includes all of the information from these nested pieces:

> Stack#push when the stack is empty places the pushed item on the top of the stack

Test Setup Name

Many test tools allow you to put common setup code for a suite into a single place in the suite. Unit test tools typically allow you to mark a method as a *setup* method. Cucumber has a *background* block where you establish the starting conditions before each feature. In Robot Framework, you can include a `Test Setup` setting that identifies keywords to run before each test.

Whatever the mechanism, you can give your setup code a name that describes the conditions it establishes. For example, if you are testing a web application, and a setup method in your JUnit class leaves you logged in on the home page, you can name the setup method like this:

```
@Before
public void loggedInOnHomePage() {
    // setup code goes here
}
```

When a failure occurs in a method in this suite, problem solvers will quickly be able to understand the conditions that were established before the test ran.

Test Code

The test code itself can guide problem solvers by describing three parts of the responsibility being tested. See *Tests as Examples of System Responsibilities* for ideas about writing expressive test code.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Describe the Failure

For many of your tests, failures will be detected by the assertion statements that you write. See *Diagnostic Assertions* for how to write assertions so that the failure messages aid in diagnosis.

Don't Repeat Yourself

If you use informative names, and you write the test code to describe the essence of the test clearly, there is no need to repeat that information in the assertion failure message.

Unhandled Exceptions

Some failures will be caught in ways other than via your tests' assertions. It is common in automated tests for failures to come from the tests' runtime environment, or from the drivers and other library code used by the tests.

Here is an example of a failure triggered by a driver. Suppose your test uses WebDriver to access html elements in a browser's DOM. If your test tries to access an element that does not exist in the DOM, the Seleniuim server will return an error code, and the WebDriver instance in your test will throw a "No Such Element" exception.

The tests' runtime environment (such as the Java Virtual Machine, or JVM) can also trigger failures. A common type of failure is a *null pointer exception*, or NPE. Your test code (or code in some library used by your test) attempts to act on some object, but the object does not exist. When the code tries to dereference the object, the JVM throws a null pointer exception.

If your code does not catch an exceptions thrown by drivers, libraries, and the runtime environment, the exception will bubble up to the test framework, which will report a failed test.

Exceptions thrown by these sources are usually less informative than exceptions thrown by the assertions you write. When you write assertions, you know the context and the intent of each assertion, and you can write your assertions so that failure messages include the most helpful information.

But the code that throws these externally-generated exceptions cannot know the intent and context of your test. They know the low-level details of what went wrong, but they don't know what your test was trying to accomplish at the time.

You may choose to write your test code to catch these exceptions and suppliment them with further information. Whether that's worth the effort depends on how often your tests experience such externally-generated exceptions, and how much additional time you spend diagnosing the failures because the exceptions don't give you the specific information you need.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Describe Interesting Events

Often as we're solving a problem we need to know more than what responsibility we were testing and what failed. We also need to know what interesting events happened along the way.

Screen Capture

If you are testing through a graphical user interface (GUI), you may want to capture screenshots or videos of the application's display.

Screenshots. Screenshots are generally easier to obtain than videos.

- The driver you are using to interact with the GUI may offer a facility for taking screenshots. For example, some of the browser drivers included in Selenium can take snapshots of the browser window.
- You can likely find open-source or third-party libraries that can capture images of the computer screen or a specific window. With a little effort you can incorporate these into your test code.
- It may be possible to use your computer's screen capture facilities. For example, on a Mac you can make your test programmatically issue a keystroke to capture an image of the entire screen.

A minor disadvantage of screenshots is that you have to decide when to take them during the execution of a test.

A way around this disadvantage is to arrange for all GUI interactions to flow through an adapter that automatically takes a snapshot before issuing any command that might change the display. Sauce Labs On Demand offers just such a feature. Every time your code sends a Selenese command to the Sauce Labs Selenium server, the server takes a screenshot before performing the command.

Videos. Video capture is increasingly popular for automated tests. Many test tools offer no direct support for videos, but you can typically find a third-party application chat your tests can use to record an entire test run.

Discarding Screen Captures. Screen images and videos can take up a lot of space on your disk. Common practice here is to retain screen captures only for a short time. When you are confident that you will not need them, throw them away. My clients often retain screen captures for failed tests for up to a week, but discard screen captures for successful tests after a day or a few hours. If they are still working on a problem after a week, they make a copy of the relevant screen captures so that they won't be purged.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Logs

When a test fails, we often want to know not only the nature of the failure, but also the relevant conditions and events that occurred while the test was running. A listing of such events is called a *log.*

There are two primary ways to log information about a test run: bare printouts, and logging systems.

Bare printouts. One way to create this information is to add statements to the tests to print information to the console:

```
login(FRODO_BAGGINS);
System.out.println("Logged in");
```

Bare printouts have several drawbacks in tests. First, they always print out, whether we want the information or not. This can clutter the console with noise. We could solve this problem by surrounding the printouts with conditional statements, but that clutters the test code, making it more difficult to understand.

A second drawback is that all printouts appear as if they had the same level of significance. Printouts of major events and of minor events look more or less the same. How important is it to know that we've logged in? We could solve this problem by inserting information into the printout, but that clutters both the test code and the printed information.

A third drawback is that a bare printout lack some of the context that would help us understand the meaning of the printouts. If we see a line that says "Number of search results: 37" on the console, it may not be easy to understand why this matters unless it's preceded by other printouts that establish the context.

Logging systems. To overcome some of the drawbacks of bare printouts, you can use a *logging system*. It's a good bet that there are numerous freely available logging systems for whatever programming language you are using.

In Java, I use a logging API called SLF4J[23] and an implementation called Logback[24], but there are literally scores of Java logging systems that you could use.

Logging systems typically have several features that bare printouts do not:

- A way to indicate the significance of each log event.
- A way to categorize each log event by associating it with a category in a tree.
- A way to indicate, when running tests, which log messages you want to see. You can filter by significance, by category, or both.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

- A way to indicate, when running tests, how you want each log message to be formatted.

Categories. With SLF4J, you write each log event using a *logger.* You can have as many loggers as you like in your system, and each logger represents a *category* of information. Typical use is to create one logger per class, and to use the Java package hierarchy as the categorization scheme.

```
package com.dhemery.foo.bar;
...
public class MyClass {
    Logger log = LoggerFactory.getLogger(getClass());
    ...
}
```

This creates a logger called “com.dhemery.foo.bar.MyClass”. It also places the logger into the “com.dhemery.foo.bar” category, which is itself in the “com.dhemery.foo” category, and so on.

These categories are useful for filtering. (See below.)

Significance. To write a message to the log, you first decide the *log level* of the event that you want to describe. The log level indicates the significance of the event. A typical logging system has a hierarchy of log levels:

- *FATAL* (the most significant level) indicates that the event is so severe that execution cannot proceed.
- *ERROR* indicates a serious problem that will likely subsequently affect subsequent operations.
- *WARN* indicates a condition or event that may adversely affect subsequent operations.
- *INFO* indicates a major event in the life cycle of the test.
- *DEBUG* Indicates that information this event is likely to be useful when debugging a problem.
- *TRACE* (the least significant level) reports information that is of very limited use even when debugging.

The log level of an event is useful for filtering. (See below.)

Many logging systems have logging methods dedicated to each of these common log levels. To log an event at a given level of significance, you call the log method associated with the log level:

```
public void myMethod() {
    ...
    login("frodo-baggins");
    log.debug("Logged in as frodo-baggins");
}
```

The `debug()` method logs this event as a *debug* level event.

Depending on how the logging system is configured, the message printed to the console might look like this:

> DEBUG - Logged in as frodo-baggins - myMethod() - MyClass.java line 33

Note that the logging system automatically discovered and displayed some of the *context* of the event: the Java file, `method, and line number of the log statement. This means that you need not clutter your test code by writing those context etails explicitly into your log messages.

Filtering, formatting, and routing. Logging systems provide three great benefits over bare printouts:

- Ability to *filter* which log messages are displayed.
- Ability to *format* log messages.
- Ability to *route* different log messages to different destinations.

These features are typically configured at runtime, independently of the test code that logs the events. This means that when you run the tests, you can configure filtering, formatting, and routing to suit your problem solving needs.

Filtering. You can filter by log level. For example, you can set a filter to display messages that are DEBUG level or worse.

You can filter by category. For example, you could configure the logging system to display log events only from the “com.dhemery.foo.bar” category.

Formatting. You can format the log output by configuring the logger to include different elements of the *context* of a log event. You can, for example, include or exclude:

- The date and time of the event.
- The debug level of the event.
- The category of the event.
- The the name of the class (the simple name or the full package-scoped name) in which the event was logged.
- The name of method in which the event was logged.

Some logging systems support a variety of log file formats, such as plain text, XML, or HTML output.

Routing. Some logging systems allow you to route logging information to a variety of destinations, each independently configurable. Some of my clients log only INFO-level or higher events to the console, but log ALL events to HTML files that are stored with other test results for use in debugging. And they log each test class's events to a separate file.

Get to know your logging system. Logging systems are all different, and they're all complex. But the information you can produce with a logging system is extremely helpful when you are trying to track down a test failure.

Get to know your logging system.

Test Your Diagnostic Messages

When you write an assertion, try to run it in conditions where it fails. This allows you to see what an error looks like from the assertion, so you can evaluate whether it gives you the information you need, expressed in a helpful way.

Run the test so that the assertion fails, and look at the failure message. Does it tell you everything you want to know about the failure? Is it easy to interpret? If not, adjust your test code to give a more useful diagnosis.

Test-First Diagnosis

It is easier to evaluate the usefulness of failure messages if you develop the system test first. You write a test, and because you haven't yet implemented the feature you are testing, the test fails.

Before you write the code to pass the test, take a look at the failure message. If it doesn't give you the diagnostic information you need, fix the assertion before you implement the feature.

Fakey Breaky Tests

If you're writing a test for a feature that already works, consider testing each assertion. Temporarily put an error either into your test or into the system you're testing so that the assertion fails.[25]

If the assertion message doesn't give you the diagnostic information you need, fix the assertion.

And remember to correct the temporary error before committing the code to version control.

Final Words

Rule #1: Make your automated diagnoses informative and trustworthy.

Rule #2: Never (fully) trust an automated diagnosis.

Diagnostic Assertions

When we write an assertion statement, naturally we want to make the code expressive. We want the assertion statement to include all of the information that matters to the assertion, and no extraneous information. We want it to be clear to the reader just what we are asserting.

When writing assertion statements, test automators often neglect an important consideration: the diagnostic value of the *message* that is displayed if the assertion fails.

Given how carefully we craft our assertion statements, it's a shame when the assertion failure message throws away important information, or when it swamps us with noise that hides the information that we want most.

Some assertion mechanisms naturally lose information when they create failure messages. Other mechanisms allow us to make the failure messages as clear and informative as the assertion statements themselves.

I will demonstrate using Java, JUnit, and a few other libraries. You will find similar assertion mechanisms and features in other programming languages and libraries.

Example

Let's set up an example that we can use to explore the diagnostic value of different kinds of assertions.

A class. Here is a very simple class, an item with text:

```
public class Item {
    private final String text;

    public Item(String text) {
        this.text = text;
    }

    public String text() {
        return text;
    }
}
```

A test. Here is a test:

```
public void findItemReturnsAnItemWithTextFoo() {
    Item item = someObject.findItem();
    assert item.text() == "foo";
}
```

The test asks `someObject` to find an item, then asserts that the item's text is equal to `"foo"`.

A fault. The code being tested has a fault, and gives the item the incorrect text value `"bar"` instead of the desired value `"foo"`.

A failure. Given that our item's text is *not* `"foo"`, but `"bar"`, our assertion should fail. When the assertion statement executes, it will detect that the value is incorrect and throw an exception to indicate an assertion failure. Then JUnit will display the exception, including any diagnostic message contained in the exception.

Assertion Styles

There are many ways to express our desired assertion in Java, especially if we use assertion mechanisms from third-party libraries, such as JUnit, Hamcrest, and Hartley. Each of the following assertion statements correctly evaluates whether our item's text is equal to `"foo"`, and each correctly throws an exception if the text does not have the desired value.

Let's look at each of these styles, and notice what information is included in the failure message, and what information is lost.

The Java *assert* Statement

Our example test espresses an assertion using a Java assertion statement:

```
assert item.text().equals("foo");
```

Notice that this short statement gives a *lot* of information about our intentions:

- `assert` says that we will make an evaluation, and that the evaluation is so important to our test that we will mark the test as *failed* if the result is not as we've specified.
- `item` says that we will evaluate an object called *item* (or some aspect of *item*).
- `text()` says that we will evaluate some text.
- The `.` between `item` and `text()` says that we will evaluate the text obtained from `item`.

- `.equals()` says that we will compare whether `item`'s text is equal to some value.
- `"foo"` says that we will compare the item's text to the literal value `"foo"`.

Each of those six pieces of information is essential to the assertion. If we remove any piece of information, the assertion loses its meaning. (Technically, we could extract the text into a local variable, so that our assertion need not refer to `item`, but let's assume that we've chosen this particular phrasing because we want to make the *source* of the text crystal clear.)

If we run this test and execute this assertion, the assertion throws an exception, and JUnit emits this message:

> java.lang.AssertionError

followed by a stack trace. The stack trace indicates the Java file name and line number from which the exception was thrown. That is, it indicates the file name and line number of our assertion statement. So if we want to understand the source of the failure, we have to navigate to the source code and read the assertion statement. It's a good thing we took pains to make the assertion statement so clear, because that's all of the information we have as we begin our investigation of the failure.

The (Mostly) Uninformative Failure Message

Note that the failure message itself tells us only that some assertion failed. It gives us no information at all (in the message itself) about what aspect of our assertion went wrong. Remember that our assertion statement expressed at least six important pieces of information. The failure message conveys only *one* of these: This thing that went awry was an assertion.

That's a critical piece of information, of course, but what happened to the other five pieces, which we took such pains to express so clearly in the code?

They were thrown away by the nature of the assertion mechanism. The Java assertion statement consists of two parts: the `assert` keyword and a boolean expression. To execute an assertion statement, Java first evaluates the boolean expression. Then it assesses the result (`true` or `false`). If the result is `true`, execution continues with the following statement. If the result is `false`, the statement throws an exception.

In evaluating the boolean expression, the computer loses the information the original expression, and saves only the `true` or `false` result. By the time the statement throws its exception, all of the other information about the expression has been lost.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

The JUnit *assertTrue()* Method

Now let's try another style of assertion, the `assertTrue()` method from JUnit:

```
assertTrue(item.text().equals("foo"));
```

This method produces the same error message as the bare Java `assert` statement:

> java.lang.AssertionError

again supplemented by a stack trace that points to the assertion in the code.

The JUnit `assertTrue()` method has one advantage over the Java `assert` statement: You don't have to tell the JVM to enable it. If you want the JVM to execute `assert` statements, you have to pass a special argument to the JVM. If you don't tell the JVM to enable `assert` statements, it (quietly) ignores them.

Otherwise, the effects of `assert` and `assertTrue()` are similar. Each throws an `AssertionError`, and neither gives you any of the other information that you so carefully crafted into your assertion.

Improving the Java *assert* Statement with an Explanation

Both JUnit and Java offer a mechanism to compensate for the limitations of the bare `assert` and `assertTrue()` assertions: *the explanatory message* (which I'll shorten to *explanation*).

With the Java `assert` statement, you add an explanatory message like this:

```
assert item.text().equals("foo") : "Item text should be foo";
```

If this assertion fails, Junit displays this message:

> java.lang.AssertionError: Item text should be foo

followed by the same stack trace as before.

That's much more informative than before. In fact, our failure message tells us nearly *everything* that the code tells us.

Am I happy? Nope.

The big problem with this explanation mechanism is that *it requires you to express the same idea twice*, once in the boolean expression, and again in the explanation. As with other forms of comments in code, it is very easy (and common) for the comment to diverge from the code it describes. You end up with failure messages that mislead.

And even if the comment stays current with the code, it is a form of duplication. If the code changes, you also have to do the extra work of changing the comment.

Adding Explanations with JUnit *assertTrue()*

JUnit includes a form of `assertTrue()` method that takes an explanation as its first parameter:

```
assertTrue("Item text should be foo", item.text().equals("foo"));
```

I find this expression more awkward then the `assert` statement. With the `assert` statement, the explanation follows the entire assertion. With `assertTrue()`, the explanation interrupts the left-to-right phrasing of the assertion.

Still, this is better than no explanation at all The `assertTrue()` explanation has the same effect as the `assert` statement explanation. If the assertion fails, JUnit displayes this message:

> java.lang.AssertionError: Item text should be foo

Still Missing: The Actual Value

By adding explanatory messages to our assertions, we can restore the information that is lost when the assertion evaluates our boolean expression.

But now I notice another bit of information that I wish were displayed in the failure message: The actual value of the item's text. Thanks to our explanations, we know the value is not the desired `"foo"`, but what *is* the value?

Of course, if we're writing our own explanations, we could easily include the actual value in our explanatory message. But that takes extra work. Granted, it's not a big burden, but it is extra work.

What if there were a way to get that information (almost) for free? The actual value is assessed somewhere during the evaluation, but it is then thrown away after the comparison is made and we've determined the boolean result of the evaluation. What if we could catch the value before it was thrown away?

We can do that. The key is to express the assertion in a way that retains both the actual value and the desired value, even after comparing them.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

The JUnit *assertEquals()* Method

JUnit offers another style of assertion, a style that, when evaluated, retains the separation between the value you are evaluating and the value you are comparing it to. If you want to compare values for equality, the appropriate method is (appropriately) called `assertEquals()`. It looks like this:

```
assertEquals("foo", item.text());
```

Though this shifts the phrasing (the concept of equality now appears earlier in the expression, and the desired value now appears before the retrieval of the actual value), we still have all six of the pieces of information that matter to our assertion.

Our new expression is okay, if a little clumsy English-wise. At least it has all the pieces, and it expresses each piece only once.

What happens when we run it? JUnit displays this message:

> org.junit.ComparisonFailure: expected:<[foo]> but was:<[bar]>

Very interesting. By passing the two pieces of information separately to the method, we enable the method emit a more helpful failure message. We give it two pieces (the expected value and the actual value) and it gives those two pieces back to us in the message.

We now have a piece of information that we didn't (and couldn't) express directly in our test code: The actual value of the item's text.

Note also that we no longer get a barely informative `AssertionError` that tells us only that something went wrong, Now we get a more specific exception, a `ComparisonFailure`. I find this slightly maddening. We invoked a method dedicated to equality, but the error message seems to remember only that we were doing some kind of comparison.

Still, this is way better than, "Hey, dude, something's busted. *You* figure it out."

And best of all: We got it (nearly) for free. We didn't have to duplicate any information in our statement. We had only to rephrase our assertion, to rearrange the same six important pieces of information.

So that's two and a half of our six pieces of information that now appear in the failure message:

1. We're asserting something.
2. We're asserting equality (this appears only partially in the failure message, so it counts only half).
3. We're comparing someting to `foo` (note that we've lost the information that it's a string,

Copyright © 2003-2013 Dale Emery and George Dinwiddie

but let's be generous and give this full credit).

And remember that we also get this bonus piece of information:

1. The actual value that we were evaluating.

This is an important addition. Let's say that we now would like all seven of those pieces of information. And we're getting three and a half of them.

JUnit *assertEquals()* with Explanation

JUnit offers a form of `assertEquals()` that takes an explanatory message. Let's use that to restore a few of the pieces that still don't get for free:

```
assertEquals("Item text", "foo", item.text());
```

Now the failure message is:

> org.junit.ComparisonFailure: Item text expected:<[foo]> but was:<[bar]>

We're duplicating the ideas of *item* and *text*, and I think we've done it in a way that also expresses the relationship between the two.

But there's far less duplication than our earlier explanatory messages, so this is progress.

Can we do better?

The Hamcrest *assertThat()* Method

Every assertion involves a comparison of some kind. Steve Freeman and Nat Pryce took the very helpful step of separating the comparison from the assertion method. They accomplished this by introducing the concept of a *matcher*. A matcher is an object that represents some set of criteria, and knows how to determine whether a given value matches the criteria. Nat and Steve created a library of widely useful matchers called *Hamcrest*.[26]

Hamcrest also includes an `assertThat()` method that applies a matcher as an assertion:

```
assertThat(item.text(), equalTo("foo");
```

The first parameter to `assertThat()` is the value that we want to evaluate. The second is a matcher object. Typically you supply a matcher by calling a *factory method* such as `equalTo()`. Hamcrest factory methods are named so that assertion expressions read informatively in the code. The assertion above says: *Assert that item's text (is) equal to "foo"*.

When a Hamcrest `assertThat()` assertion fails, it throws an exception with a message like this:

> java.lang.AssertionError:
> Expected: “foo
> but: was ”bar"

This failure message is similar in content to the JUnit `assertEquals()` failure message. The great advantage of Hamcrest-style assertions is that you can easily extend the set of available assertions. We will leave that as an exercise for the reader.

Hamcrest also has a version of `assertThat()` that takes an explanation as a parameter. Normally all we need to do is describe the subject of the evaluation:

```
assertThat("Item text", item.text(), equalTo("foo"));
```

When this assertion fails, it emits a message similar to the one from JUnit `assertEquals()`:

> java.lang.AssertionError: Item text
> Expected: “foo”
> but: was “bar”

As with JUnit’s `assertEquals()`, at the expense of a little bit of redundancy, we have made our failure message express all of the information that matters to the assertion.

So our assertion messages read very similarly to JUnit’s assertion messages. But Hamcrest’s assertion statements read far more expressively in the code.

The Hartley assertThat() Method

For my own tests, I often take a step beyond Hamcrest. I often want to evaluate not only some subject, but more specifically some attribute or feature or property of the subject. Each of the examples in earlier sections evaluates not only `item`, but more specifically the item’s *text* as returned by its `text()` method.

I often find it helpful to write assertion statements that separate the subject from the feature. I have created an assertion method[27] to help me do that:

```
assertThat(item, text(), is("foo"));
```

This assertion method takes three parameters. Each parameter is an object. The first object is the *subject* of the assertion. In this case, we are evaluating `item`.

The second parameter is the *feature* being evaluated. In the same way that Hamcrest matchers

are objects that evaluate other objects, Hartley *features* are objects that extract values from other objects. The `text()` method is a factory method that produces a feature object that can retrieve the text from an item.

The third parameter is a matcher that compares the extracted feature to the desired criteria.

In the code, this reads just the same as the Hamcrest assertion:

> Assert that item's text (is) equalTo "foo"

But notice what happens when this assertion fails:

> java.lang.AssertionError: Expected: Item text "foo"
> but: was "bar"

With no redudancy in the assertion statement itself[28], we now have a failure message that includes every important detail of the assertion.

Naming Unit Tests

Dale Emery[3]

Last year I read Brian Button's[29] wonderful article "Double Duty" in *Better Software* magazine (the February, 2005 issue). One of the things I learned is that Brian is the world's best namer of unit tests. I visited Brian's web site for more of his ideas and found an article called "TDD Defeats Programmer's Block–Film at 11"[30] In this article, Brian describes using the Test Driven Development process to write a "continuous integration system" (a tool that automatically (re)builds software systems when programmers change the source code). Here are some examples of his unit test names:

- Starting Build With No Previous State Only Starts Build For Last Change
- Previous Build Number Is Incremented After Successful Started Build
- Last Build Failing Leaves Last Build Set To Previous Build

What makes these names so good? I analyzed a few dozen of Brian's test names and found this pattern: **stimulus and result in context**. Let's examine these names to identify the parts.

Starting Build With No Previous State Only Starts Build For Last Change:

- Context: There is no previous state (i.e. no previous builds were done).
- Stimulus: Start a build.
- Result: A build was started for only the last change.

Previous Build Number Is Incremented After Successful Started Build:

- Context: There were zero or more previous builds.
- Stimulus: Request a build that will succeed.
- Result: The build number is one more than before the build.

Last Build Failing Leaves Last Build Set To Previous Build:

- Context: There were previous builds, the most recent of which is recorded in the system as the last build.
- Stimulus: Request a build that will fail.
- Result: The previously identified last build is still identified as the last build.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

One of Brian's tests from a different system—an "animal factory" (a concept better left unexplained)—is called **Default Animal Is Cow**.

- Context: No animal type has been identified as the desired type of animal for the system to manufacture.
- Stimulus: Request that the system manufacture an animal.
- Result: A new cow exists.

Now that I've learned the pattern that makes Brian's test names so useful, I can use it deliberately.

Using the context-stimulus-result scheme increases the value of tests as documentation. The resulting names make clear what specifically is being tested and under what specific conditions. This helps the reader to understand quickly what each test does, and what is covered by each set of tests.

Another benefit is that **the context-stimulus-result naming scheme encourages you to clarify your thinking about each test.** Each unit test establishes some set of starting conditions, or context. Each stimulates the system. Each compares the result to a desired result. In order to name these elements you will have to think about the specifics of each and clarify them well enough that you can describe each in a few words.

If you're having difficulty naming a test using this scheme, that may indicate a problem in your test. Perhaps the test is doing too much work, or your test suite is doing too little. For example, suppose you're testing software to manage bank accounts, and one test is called *Withdrawal Test*. We can tell from this name that the test tests the withdrawal feature in some way. But we don't know what specific aspects of withdrawals this test is testing.

Does *Withdrawal Test* test only that a withdrawal of less than the account balance reduces the balance by the proper amount? If so, calling this test "Withdrawal Test" may indicate that your suite of tests for the withdrawal feature is missing many important test cases. The name of the test gives readers an overly broad sense of what the test actually tests. Does *Withdrawal Test* test a score of different stimuli under a dozen different conditions? If so, it's probably doing too much work. The name of the test does not quickly tell readers what is being tested.

Whether *Withdrawal Test* is doing too much work or too little, we can improve the test by applying the context-stimulus-result scheme. If *Withdrawal Test* is doing too much, we can use the scheme to identify how to break the test into smaller, more focused tests with more descriptive names. If *Withdrawal Test* tests only one tiny aspect of withdrawals and leaves other aspects untested, we can use the scheme to create a better name for the test and to identify other tests to write.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Testing the Tests

When I'm automating a test, I often want to ensure that my test is actually testing the thing I want it to test. So I make the test wrong in some small but meaningful way, then run it with the expectation that it will fail. For example:

- If the system is supposed display "42" in the "meaning-of-life" field, I'll change the the test to assert that it displays "43".
- If the system is supposed to display "42" only when the current user is "Ford Prefect," I'll change the test to log in as "Zaphod Beeblebrox".
- If the system is supposed to display "42" only after Ford visits the "Restaurant at the End of the Universe" page, I'll change the test to omit the "visit the restaurant…" step.

I'll do one of these at a time. I'll make the test wrong, run it, and notice whether it still passes. If the test now fails, and it makes sense to interpret the failure as due to my change, I'll conclude that the original test got that detail right, and change it back.

If the test still passes, that means the detail I changed does not have the effect I wanted, so I'll need to do some debugging.

If the test fails, but the failure isn't clearly attributable to the way I broke the test, I'll need to do some debugging.

Once I've tested the test in those ways, and once the system passes the test,

I treat the test as being reasonably reliable until I discover some reason to doubt it. Here are some occasions where I might begin to doubt that my test is testing what I thought it was testing.

- The test reports a failure. This might indicate a problem either in the test or in the system.
- I believe that the system or its environment is broken, yet the test reports success.
- The requirements have changed and the test no longer reflects the current requirements.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Testing in Depth

George Dinwiddie[31], iDIA Computing, LLC[32]

In the late 1970s, in the *Co-Evolution Quarterly*, the magazine successor to *The Whole Earth Catalog*, Peter Warshall stated that geodesic dome houses **always** leak. This was a bold and surprising statement at the time, coming from a man who was considered one of the finest builders of dome houses–ones that didn't leak.

Why did he make this statement?

He went on to explain, that the design of a dome house depended on a single skin being perfect waterproofing technology. Traditional houses, by comparison, have multiple imperfect layers. There are overlapping shingles, which keep most of the water out. Below that there's a layer of tar paper, which keeps out most of what reaches it. Then there's the plywood sheathing, which blocks or absorbs most of what penetrates the tar paper. Then the attic insulation….

No single layer of this system has to be perfect.

Software testing works the same way. If you depend on one method of testing, you're going to leak bugs into production.

From the developers' viewpoint, one of the things I've found is that **good, simple design** helps to minimize the places where such problems can hide. By minimizing coupling and maximizing cohesion, the things that change together tend to be visible together, where it's obvious what needs to change. In many designs I see, a change here ripples through the system and lots of places need to accommodate a new parameter or a new property of an object. To me, this is a point of pain, and I'll want to clean it up. Perhaps I'll create a parameter object that encapsulates the tuple that needs to be passed around. Perhaps I'll move behavior depending on the object's properties into the object, so the object's collaborators don't have to worry about the internal changes. These things are the right thing to do, not because someone said so, but because they reduce the incidence of errors.

If you do unit testing (or micro-testing, as Mike "GeePaw" Hill calls it[33] of small chunks of code at a low level, you'll catch most of the coding mistakes. These tests **make sure the code does what you think it does**. If you test-drive your code into existence as a design technique, then you get these tests as a side-effect.

While you're at that level, **add "negative tests" to your unit test suites**. In other words, add tests that verify that things that shouldn't happen, don't. This is more of a testing frame of mind than a design one, though it may drive the creation of code that prevents bad things. A careful developer will check edge conditions and perhaps some invalid input, depending on the circumstances.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Below this, do **small scale integration tests** of the code that talks to other systems (such as the database). Make sure these interfaces work correctly. At each system boundary, I generally use an adapter or mediator class to “impedance match” the other system’s API into one preferred by my application. In my unit tests and negative tests, I use fake adapters to give control and visibility of the system under test to the test code. When I do this, I make assumptions about the way the other system responds. I test these assumptions by writing tests of my real adapter, talking to the other system, that validate my assumptions.

Then add some **tests to ensure that the system works together as a whole**. These are termed system, or integration, or acceptance, or customer tests. Certainly you want the correct operation of the main features to be tested with automated scripts. You want to test the expected failure modes, also. The negative tests at this level, however, would lead to combinatorial explosion if you tried to cover them exhaustively.

Hence the need for **exploratory testing**. Exploratory testing is interactive testing by a human tester who is trying to find ways to surface problems. Its proponents like to point out that it’s the best way to find bugs. I believe them, because the bugs surfaced by automated regression tests would already be fixed. At this point, if things are done well, there’s little need for testing that things are working correctly. Instead, the good tester will concentrate on things that are possible, but likely not anticipated by the developers. Uncovering the implicit assumptions, and finding holes in them, is the value of a good interactive tester.

Well-known security technologist Bruce Schneier says[34]

> “One of the basic philosophies of security is defense in depth: overlapping systems designed to provide security even if one of them fails.”

You can’t protect your network from malicious people by relying on perimeter security. You have to bake security into other levels of access, too.

The same is true of testing to protect your application from bugs. Errors must be attacked at different levels: design, implementation, and integrations. Errors must be attacked from different viewpoints: what’s intended and what’s not intended. No one technique, viewpoint, or level will ever be sufficient.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Testing Classes in Isolation and in Collaboration

Dale Emery[3]

When I'm talking to programmers about writing tests for their own code, one of the questions that comes up often is: *Should we test classes in isolation from each other, or in collaboration with each other?*

I like both kinds of tests. Here's why.

I like tests that isolate classes. When a failure occurs, the tests tell me specifically what class failed, and what method failed. That guides me more directly to the fault — the specific code that is broken — and saves a ton of debugging.

I like tests that exercise collaborations. When a failure occurs, the tests tell me that:

- one class or the other is not fulfilling its responsibilities, or
- the collaborators disagree about each other's responsibilities, or
- some other class (the "electrician" class that connects the collaborators with each other) has wired the collaborators together improperly.

If the individual classes are well tested, I can focus my collaboration testing specifically on wiring and agreements. And if the individual classes are tested well, collaboration test failures tell me about disagreements and improper wiring.

When I test classes in isolation, failures guide me quickly to faults.

When I test classes in collaboration, failures tell me where the classes disagree about each other's responsibilities.

So I write both kinds of tests.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Test-driving those “non-functional” stories

George Dinwiddie[31], iDIA Computing, LLC[32]

Questions often arise about “technical stories,” especially when developing frameworks. I’ve been test-infected for quite some time, and I don’t find much need for “technical stories.” I find that driving the development with user stories keeps things on track much better. So, for developing a framework, I suggest developing a rudimentary client for that framework in parallel. It’ll help you drive things from an end-user point of view. It will also help your framework become usable by and useful for client code.

Generally I start with the story test (or part of it, if the story is a little big). I’ll make that test pass in a trivial way, and then use unit tests to drive completion of the story.

That works for functional needs, but what about the non-functional requirements such as performance? I find that non-functional stories, such as performance, aren’t good candidates for directly driving the development of code. Instead, I’d approach such a story in this manner.

I would write a story for the performance criteria and an acceptance test checking the performance criteria. That doesn’t mean that the story is a good one for driving development, but it’s still a business need. The “non-functional” requirements don’t make for easy TDD.

If the performance test doesn’t meet the acceptable criteria, I would profile the system and see where the time is being spent. Most of the time, the culprit is rather well defined. Often the solution (well, a solution) is pretty obvious. Sometimes it’s damn difficult. A new algorithm might have to be developed.

In any event, I would use TDD to drive the new solution. That TDD would drive from a technical basis, however, not from the story test. The story test would just verify if the target performance had been met.

For a simple example, the story might be “Display the sorted list of froobles in less than 1 second.” A performance test written for the frooble display might show it takes 2.5 seconds. Profiling the application shows that 1.9 seconds of that time is spent in the bubble sort routine. Perhaps the decision is made to use a merge sort. I would test drive the writing of the merge sort routine. Then I would substitute the merge sort for the bubble sort and run the acceptance test. Great, it now takes 0.9 seconds to display the sorted list of froobles. We’re done, for now. Non-functional requirements have a way of popping up again in the future, as the system grows and evolves.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Design for Testability

George Dinwiddie[31], iDIA Computing, LLC[32]

I first encountered the issues of testability when working with integrated circuit design in the 1980s. The same issues apply to software systems.

1. You want to be able to easily put the system into a known state. It's best if you can get to the state you want for your test without going through a number of interactions or other states on the way.

2. You want to be able to drive internal nodes of the system so that you can test parts in isolation. It's much harder to test everything through the GUI, public API, IC pins, or whatever is the normal interface.

3. You want to be able to sense internal nodes of the system so that you can test parts in isolation. Like being able to drive internal nodes, this reduces the combinatorial complexity.

4. The special access you add for testability does add some complexity, can provide new failure modes, and might leave some paths untested. Be aware of this and consider what these issues are for your system.

Your work gets a bit easier if you consider this as part of building the system, rather than just as part of testing it after the fact. If you build your tests first, the tests will specify the state and access you need for testability. Running these tests while the system is built will result in testability appearing almost magically.

For ICs, the first level of making a circuit testable was to the internal state predictable and discernible. Sometimes this was accomplished simply, by having the power-up state known, rather than random, and by being able to clock internal nodes into a shift register to be read out on an output pin in a special test mode. That was enough to make it testable, but not generally enough to make it easy to test.

Being able to drive internal nodes to various known states gave a lot more power, allowing "unit testing" of various blocks of circuitry. This generally required some additional hardware added just for testability, but paid handsome dividends in reduced time to test units in production.

With ICs, much of the expense is in the packaging, and that expense was significantly related to the number of pins. Testing equipment evolved to do the equivalent of "the bed of nails" used on circuit boards, but applied to pads on the circuit die that were never bonded out to pins. This allowed easier access to internal nodes, both for driving state and for reading it, prior even to slicing the wafer into individual chips. The heads that probed the circuit had a small nozzle to spray dye on failed circuits so they could be discarded before packaging.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

With software, the ability to drive and access internal nodes is much easier–often not requiring any additional logic. Sadly, many programs manage to make these internal nodes inaccessible, in spite of the lack of cost to do otherwise. While the hardware world acknowledged the cost savings of adding transistors (and chip area) to reduce the cost of testing, many in the software world will argue against making a method “protected” instead of “private” so that it can be overridden for the purpose of testing.

Well designed code, like well designed circuits, contain the complexity better and are therefore easier to adequately test. This is, to me, an important point. We’re very likely to miss stuff. Let’s make it as hard as possible to miss stuff. And let’s make it as easy as possible to notice when we’ve missed stuff.

In the words of C. A. R. Hoare,

> *There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.*

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Appendices

The testers get behind at the end

George Dinwiddie[31], iDIA Computing, LLC[32]

It's a very common complaint, such as this one left on the Scrumdevelopment yahoogroup[35]

> Usually in different phases, workload for tester and dev is different. E.g. when a project is coming to the end, most of the tasks will be test.

There are a couple of big red flags waving at me in those two sentences. One is *"different phases."* Why should a software development project, especially one only a couple weeks to a couple months long, have phases? The other is, at *"the end, most of the tasks will be test."* Postponing testing to a phase at the end has never worked very well. It *always* results in the testers being behind at the end.

Does this situation sound somewhat familiar to you? If so, what can we do about it?

Many teams try to live with it. I've seen teams institutionalize "being behind" by implementing in one iteration and testing in the next. When they do that, they generally find that problems are found in the testing, so the implementation really drags across two iterations. The rework in the second iteration bumps new work planned for that iteration, so things continue to slide. And since it's hard to tell when a bit of work really is completely done, it's hard to know how much work fits into an iteration and when the release will be ready. Often things feel the same as they did before going to time-boxed iterative development. That's no surprise, because this really isn't time-boxed iterative development. It's waterfall phases in sheep's clothing.

Just don't do it. Get things completely done and tested before moving on to the next thing.

But how? Don't you have to wait until the code is written before you can test it?

No, you don't. Start when the product owner is describing what is to be done. Distill that down to the essence of the user story. Make sure that the product owner agrees that this *potentially automatable* distillation is what is desired.

Then, while the developers are hard at work implementing the functionality, the testers should be automating the test that verifies it. To be sure, this is often something the testers, and the organization they work for, find unfamiliar. It'll go a little slow and shaky while you learn. It's completely understandable that testers will need some assistance from the developers as they automate the tests–after all, test automation is a form of programming.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

The bottom line is that the story isn't ready for acceptance until both the implementation and the test are done, and the test passes. That's the minimum, but take a further look, too.

Over time, this will get easier. The testers will learn to do more test automation with less help. The resulting set of regression tests will grow, giving quick feedback that functionality that worked before, still works. Instead of the testers getting further and further behind as they continue to check the same functionality iteration after iteration, they'll get further ahead because running these tests takes about the same amount of time. They'll have more time to do exploratory testing, and less of that exploratory testing will have to cover basic functionality.

It's hard work to make automated acceptance testing a success. In the end, though, it's the only thing that can make testing a success in time-boxed iterative development. If you don't make it work, I guarantee the testers will get further and further behind.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Planned Response Systems

Dale Emery[3]

I first learned about the idea of planned response systems from III, a colleague and friend of mine. I later read about the idea in depth in McMenamin and Palmer's profound book *Essential Systems Analysis*.[36]

The idea of planned response systems is fundamental to how I think about programming and testing. I encourage you to notice what happens when you think about software systems as planned response systems.

A **planned response system** is a system that responds in planned ways to events in its environment.

For example, a software system is a planned response system — it responds in planned ways to users' actions.

In an object-oriented software systems, each object is a planned response system — it responds in planned ways to messages sent by other objects.

Planned response systems produce two general **kinds of results**: They send messages to entities outside of the system boundary, and they make changes to the essential memory of the system.

An **event** is a significant change in the system's environment. A change is **significant** to the system if the system is obligated to respond to the change in a planned way.

Events fall into two broad categories: Changes initiated by entities in the system's environment (e.g. users or other systems), and temporal events caused by the passage of of time.

For example, an ATM is obligated to respond in a planned way to a user's request to withdraw cash. The user's request is an event.

A **system responsibility** is a system's obligation to respond to each notification of a specified kind of event under specified circumstances by producing a specified set of planned results.

The specification of a system responsibility consists of three parts: A specification of a kind of event, a specification of a set of circumstances, and a specification of the set of planned results that the system is obligated to produce in response to being notified of an event of that kind under those circumstances.

A system becomes **obligated to respond** to an event when a system designer allocates that responsibility to the system.

The **essence** of a planned response system is the set of responsibilities allocated to the system, independent of the choice of technology used to implement the system.

The definition a system's essence makes no mention whatever of technology inside the system, because the system's essential responsibilities would be the same whether it were implemented using software, magical fairies, a horde of trained monkeys, or my brothers Glenn and Gregg wielding pencils and stacks of index cards.

One way to identify the essence of a system is to indulge in **The Fantasy of Perfect Technology**. Imagine a system implemented using perfect technology. Then ask yourself some questions about the quality attributes of the system.

How fast would it respond? *If it were made of perfect technology, of course it would respond instantly, with zero delay.* How many users could use it at once? *An infinite number of users.* How much information could it store? *An infinite amount.* How often would it break? *It would never break.* How long does it take to start up? *None, because it's always on and always available.* How much energy would it use? *It would use no energy; heck, it might even generate energy for free.*

The one glaring flaw of perfect technology is that it does not exist. Real-world technology is imperfect. That's what makes this exercise a fantasy. But it's a useful fantasy, because it helps us to separate the system's essential responsibilities from the temporary constraints of current technology.

Note that we apply the Fantasy of Perfect Technology only inside the boundary of the system. Even in our fantasy, the world outside of the system is made of real, imperfect stuff, with which the system will have to interact.

Now apply the fantasy to your own system. What responsibilities would your system have even if you could implement it using perfect technology? That set of responsibilities is your system's essence.

The *essential memory* of a system is the set of data that the system must remember in order to fulfill its obligations — that is, in order to respond as planned to future events.

For example, an ATM must remember users' account balances in order to determine whether to satisfy users' requests to withdraw money.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

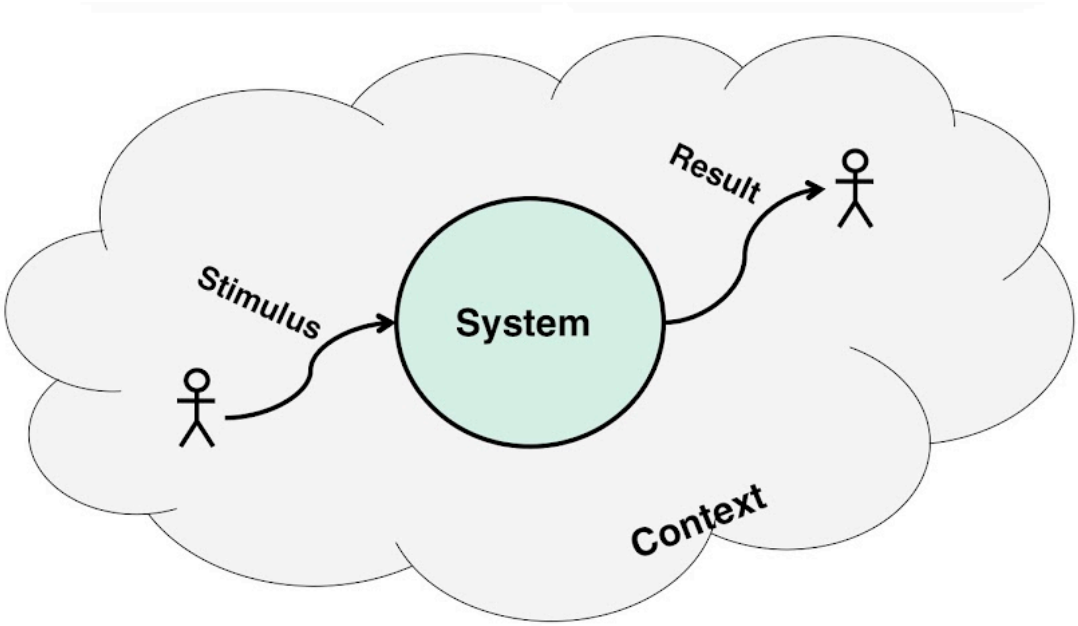
The Anatomy of a Responsibility

Dale Emery[3]

Because the concept of **system responsibility**[37] is so foundational to how I develop and test software, I want to expand on it. Recall that I defined a system responsibility as a system's obligation to respond to each notification of a specified kind of event under specified circumstances by producing a specified set of planned results.

A system responsibility includes three parts:

- A stimulus that triggers the system to respond to an event.
- A context in which the system is required to respond to the stimulus.
- A set of results that the system is obligated to realize in response to that stimulus in that context.



Responsibility

Stimulus

A stimulus is a message, sent by someone or something outside the boundary of the system, that informs the system of an event to which it is obligated to respond. The stimulus has a name, which may identify either the event that it represents or the planned response that the system must carry out. The stimulus may include additional information about the event.

Stimuli are delivered to a system through its interfaces. An interface defines a set of messages to which a system responds, and the mechanisms by which those messages are delivered. For GUI systems, the interface includes a suite of windows, forms, buttons, text fields, and other

mechanisms that translate user gestures (mouse clicks, key presses) into messages. Web-based systems receive stimuli through HTTP requests and other interfaces. Smaller scale systems, such as objects inside a software application, expose Application Programming Interfaces (APIs) that define the set of methods to which internal objects and subsystems respond.

Result

A result is an effect that the system realizes in response to a specified stimulus in a specified context. A result may be either a message delivered to someone or something outside the boundary of the system or a change in the system's internal state.

GUI systems deliver messages through forms, windows, screens, audio devices, and other output devices. Web-based systems deliver messages through HTTP responses and requests. An application's internal objects and subsystems deliver messages through method calls and method return values.

In addition to delivering messages to external entities, systems also respond to events by recording information internally, and by making changes to that internal information. The information may be stored inside the running application, in a database, in files on the computer's file system, or other storage mechanisms. The information that a system stores in order to guide its responses to future events makes up the system state.

Context

Sometimes a system's planned response depends not just on information delivered through the stimulus, but other information as well. *The context for a given responsibility is all of the information other than that delivered in the stimulus that influences the results that the system is obligated to realize in response to an event.* The context may include information about the state of the system itself — that is, information that the system previously recorded in its internal memory about prior events. The context may also include information that the system can observe across its boundary — information that the system must request from external entities in order to fulfill the responsibility.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

The Unbearable Lightness of Faking

Dale Emery[3]

If you want to test class in isolation, but the class works with a collaborator, you may need to provide a fake collaborator for the class to work with. A fake collaborator provides useful isolation in two directions:

- It isolates the test from the quirks of the real collaborators. This makes failures more informative: If the test fails, the fault is likely in the test subject, and not in the collaborator.
- It isolates the real collaborators from the test. This is important if the real collaborator is, say, the corporate accounts receivable database. You don't want your tests messing with that.

Fake collaborators often provide other benefits over real collaborators. One benefit is that **fake collaborators increase testability by increasing your control over the test subject's environment.** It's usually easier to set up a fake collaborator to feed your test subject a particular data value than to set up the real collaborator to do the same thing. And if the real collaborator takes a long time to do its work, you can gain control over the speed of the test by writing a fake collaborator that takes essentially no time at all.

Fake collaborators also increase testability in another way: They give you greater visibility into the results produced by the test subject. Sometimes it's difficult or time consuming to observe what data the test subject delivered to a real collaborator. If you write a fake collaborator, it's easy to instruct it to remember the data that the test subject delivered. And it's easy to gain access to that information so that you can compare it to your expectations.

I've identified a number of jobs that I often want fake collaborators to do for me when I'm writing tests. Each of these jobs helps me to gain control over the test environment or visibility into the test results.

1. **Fill in an argument to a method call.** Suppose the test subject requires me to pass an argument to it — either through the constructor or through the method I'm testing — but the argument is never used during the test. In this case, all I need the "collaborator" to do is to fill in a value in the method call. If that's all I need, I can pass `null`.
2. **Accept calls from the test subject.** If the test subject calls the collaborator's methods, but test doesn't care what the collaborator does, I can write a fake collaborator with dummy methods. If the interface specifies that a method doesn't need to return anything, I can simply write a dummy method with an empty body. If the method must return a value, I can write the dummy method to return a simple default value, such as `0`, `null`, or `false`. Objects like this, and similar objects with very simple default behavior, are often called *Null Objects.*

Copyright © 2003-2013 Dale Emery and George Dinwiddie

3. **Provide inputs to the test subject.** Sometimes the test subject requires a value other than `0`, `null`, or `false` in order to run. And sometimes I'm writing a test to determine whether the test subject responds appropriately when it receives specific interesting values from its collaborators. In either case, I enhance the fake collaborator to store an appropriate value and deliver it to the test subject when called.
4. **Record outputs from the test subject.** Sometimes I want to know whether the test subject sends the right information to the collaborator. I can write the fake collaborator's methods to store the inputs it receives from the test subject. And I can write accessor methods in the fake collaborator, if necessary, so that the test method can retrieve them.
5. **Verify outputs from the test subject.** Sometimes it's useful to have the collaborator do the verification itself, rather than having the test retrieve values from the collaborator and verify them. When I want this, I can create a *mock object,* an object that has expectations and can verify them. I can either write my own mock objects, including the verification methods, or I can use one of the numerous mock object libraries that make mocking easier.
6. **Verify what methods the test subject calls.** Sometimes I want to verify not only whether the collaborator received the right values, but also whether the test subject called all of the right methods. And sometimes I want to make sure the test subject does *not* call certain methods. Mock object libraries typically provide ways to verify function calls.
7. **Verify the sequence in which the test subject calls method.** Every now and then I want to verify that the test subject not only called the right methods on the collaborator, but also called them in a specific order. This can be useful for testing protocols. Some mock libraries provide a way to verify the order of method calls. Sometimes need this feature I write a *logging* collaborator that simply writes each expected method call to a string and each actual call to another string. To verify whether the actual calls matched expectations, my test can direct the logging collaborator to compare the two strings.
8. **Collaborate fully.** If the test somehow requires the full behavior of a real collaborator, I can use a real collaborator. So far, I haven't found a need for this when I'm trying to test classes in isolation. I do use real collaborators when my intention is to test the collaboration, and not just one class or another.

I've numbered these features in order of *lightness*. The lighter features are easier to create; the heavier features take more work. `null` is the lightest collaborator of all, and the real collaborator is the heaviest.

My preference when writing tests is to **use the lighest fake collaborator that gives me the visibility and control that I need for the purposes of my test.** This keeps my tests as light and flexible as they can be.

Often I start by passing the lightest collaborator of all — `null` — to the test subject, and then wait for the test tell me when I need to add more behavior to the collaborator. If the test subject needs something other than null, I'll find out when I try to run the test and get a null reference

pointer exception. Then I'll move to a Null Object. If the default values returned from the Null Object don't satisfy the test subject, the test usually signals that with an exception or failure of some kind, and I'll move to a heavier collaborator.

I call this approach ***The Unbearable Lightness of Faking:*** **start with the lightest possible collaborator, and use it until the lightness becomes unbearable and I absolutely must switch to something heavier.**

Copyright © 2003-2013 Dale Emery and George Dinwiddie

If you don’t automate acceptance tests?

George Dinwiddie[31], iDIA Computing, LLC[32]

If you and your team don’t use automated acceptance tests, please let me know how you handle regression tests as the application grows larger.

OK, I know that “acceptance tests” are somewhat a misnomer. While they may provide a go/no go indicator for the functionality of a user story, we all know that it’s possible that the application passes the test and still isn’t what the Product Owner or Customer wants. You still need to show it to the Product Owner/Customer to get their acceptance. Bear with me, though, and let’s use this common term.

So, a Product Owner, a Developer, and a Tester ~~walk into a bar~~ sit down to talk about something that the system under development should do. The Product Owner describes the user story. The Developer and Tester ask questions (and make suggestions) until they think they can answer the basic question, “How will I know that this story has been accomplished?”

No matter how or when it’s done, these three amigos must agree on this basic criteria or things will go wrong. Turning this agreement into an automated acceptance test (or three) gives it a precision that often tests the agreement and uncovers fuzziness or conflicting definitions in the words we use. **Automated acceptance tests help us express our expectations.**

If you don’t use automated acceptance tests, how do you clearly communicate desires (“requirements”) between the business, the developers, and the testers?

If your testing is manually executed, or is automated using record-and-playback, then you’ll have the problem where the testers have to wait until the developers think they’re done before they can start verifying the functionality. This puts the testers behind from the very beginning. It also delays the feedback to the developers when the functionality doesn’t behave as expected and results in bug-fix cycles on code thought to be complete. These things combine to slow down the pace of development.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

It's more valuable to automate those tests while the code is still written. As development proceeds, you can see those tests start to pass, providing a clear indication of the progress. If a developer writes code expected to make a particular test scenario work, but the test fails, then you can delve into the issue right away. Is there a mistake in the code, in the test, or just a lingering disagreement about what we intended to do? **Automated acceptance tests that pass express the growth of functionality in our application.**

If you don't use automated acceptance tests, how do you monitor the progress of development?

Once the functionality works, we want it to continue working, unless we expressly decide it should work a different way. If we want to know that it continues to work, we need to verify that. That means that we need to continue to check a growing amount of functionality. If that checking requires significant human effort, we'll soon be overwhelmed by it and our progress will get slower and slower.

Computers are great for doing our repetitive grunt work. Yes, it's a continually increasing job for them, too, but they're usually faster than people, they can work longer hours, and they can easily scale to handle more work by adding hardware. If computers are checking that all of our tests are still working on a daily or more frequent basis, then we rapid notification when we've accidentally broken an old feature. **Automated acceptance tests express our confidence that the system continues to work.**

If you don't use automated acceptance tests, how do you maintain confidence that the system still works?

If you don't use automated acceptance test, please let me know the answers to these questions–especially the last one.

Copyright © 2003-2013 Dale Emery and George Dinwiddie

Footnotes

1. Yes, we are aware that chimpanzees are not monkeys. But “the any chimpanzee question” just doesn’t have the same rhythm as “the any monkey question.” ↩

2. “Refactoring with Ben Orenstein.” https://peepcode.com/products/play-by-play-benorenstein ↩

3. Dale Emery. http://dhemery.com ↩

4. I learned this idea from Elisabeth Hendrickson, an extraordinary tester. ↩

5. The examples presented here run within Robot Framework, an increasingly popular test automation tool that allows you to write tests in a variety of formats. As you will see, Robot Framework offers techniques to write clear, maintainable tests. Robot Framework is free and open source. See the Robot Framework web site for further information. ↩

6. My first concern: What’s `fred` doing there? That’s a user name. I’ve given a user name because the *Create Account* command (however it’s implemented) requires a user name. Still, the user name has no bearing on password validation, so it’s extraneous for this test. My second concern is that it isn’t immediately obvious what’s significant about that specific password. ↩

7. I originally counted only seven occurrences, missing the one in the name of the test. That’s another challenge with duplication. When you have to change all of the occurrences, it’s easy to miss some. ↩

8. Or count the number of creates in the original test in Listing 1. Notice that by extracting incidental details from the test into a keyword, we’ve also reduced the number of changes we’d have to make if we switched from *create* to *register*. Bonus! ↩

9. Notice that these new keywords uo not depend on any implementation details of the system. They’re built entirely on our lower-level keywords. If the implementation details change, these keywords will continue to be valid, and will not require change. Also, I’ve changed the user name from `fred` to `arbitraryUserName` to help readers understand that, for the purpose of this keyword, there’s nothing special about this user name. ↩

10. In the real world, accepting a password means more than simply reporting that the account was created. In addition, the system must of course actually create the account. A complete test would verify those essential results, and not simply take the system’s word that it created the account. Systems lie! I’ve omitted those details to keep the example small enough to talk about. ↩

Copyright © 2003-2013 Dale Emery and George Dinwiddie

11. There is still duplication here. We could reduce it further, perhaps by creating a `Rejects Passwords` a keyword that takes a *list* of invalid passwords and checks whether the system rejects each one. Would that make the test clearer or more maintainable? My guess is no, but it's worth considering. Try it for yourself and see. ↩

12. Yes, using variables does require us to include distracting dollar signs and braces in our test. Does the clarity of the names outweigh the distraction of the syntax junk? ↩

13. These steps use another tool, Selenium, to drive a web browser and to interact with the web app. To start up Selenium and a browser to run the tests, and to shut down Selenium and the browser after the tests, we have to add another few geeky lines to our tests. In the interest of staying focused, I won't include those lines here, but it's a grand total of eleven lines of test code. ↩

14. Though these tests use Robot Framework's particular test format, many other open source test tools – Fit, FitNesse, and Cucumber being among the more popular – offer similar ways to express the essence of your tests and to hide implementation details. ↩

15. Robot Framework: http://robotframework.org/ ↩

16. "Writing Maintainable Automated Acceptance Tests". In this handout. Also available at Dale Emery's website at http://dhemery.com/pdf/writing_maintainable_automated_acceptance_tests.pdf ↩

17. Bob Martin's video demonstration of the principles of writing maintainable tests: http://vimeo.com/8041760 ↩

18. FitNesse: http://fitnesse.org/ ↩

19. FEST-Swing: http://fest.easytesting.org/ ↩

20. Robot Framework Selenium Library: http://code.google.com/p/robotframework-seleniumlibrary/ ↩

21. Cucumber: http://cukes.info/ ↩

22. Twist: http://www.thoughtworks-studios.com/agile-test-automation ↩

23. Simple Logging Facade for Java (SLF4J). http://slf4j.org ↩

24. Logback. http://logback.qos.ch/ ↩

25. See "Testing the Tests" in this handout. ↩

26. The word *Hamcrest* is an anagram of the word *matchers.* You can find Hamcrest matcher libraries for a variety of programming languages at http://hamcrest.org/. ↩

Copyright © 2003-2013 Dale Emery and George Dinwiddie

27. For this and other Java code that I find commonly useful when I automate tests, see my *Hartley* library. http://github.com/dhemery/hartley ↩

28. Note that I *did* have to do some extra work in order to convince the failure message to describe the subject and the feature. I had to implement the `toString()` method in both the `Item` class and in my feature object class. For that small amount of work, I can now use those classes in many assertions, and I gain the diagnostic expressiveness at no additional cost. ↩

29. Brian Button http://www.agileprogrammer.com/oneagilecoder/ ↩

30. "TDD Defeats Programmer's Block—Film at 11" http://agileprogrammer.com/2005/03/27/tdd-defeats-programmers-block-film-at–11/ ↩

31. George Dinwiddie's blog, *Effective Software Development*, http://blog.gdinwiddie.com ↩

32. iDIA Computing, LLC, http://idiacomputing.com ↩

33. Mike "GeePaw" Hill, "They're Called Microtests" http://anarchycreek.com/2009/05/20/theyre-called-microtests/ ↩

34. Bruce Schneier, "Security in the Cloud" http://www.schneier.com/blog/archives/2006/02/security_in_the.html ↩

35. "How to find a balance of tasks for testers and devs in different phases?" http://groups.yahoo.com/group/scrumdevelopment/message/45191 ↩

36. *Essential Systems Analysis*. http://www.amazon.com/exec/obidos/ASIN/0917072308/dalehemery–20. ↩

37. "Planned Response Systems". In this handout. Originally published in *Conversations with Dale* at http://cwd.dhemery.com/2009/02/planned-response-systems ↩

Copyright © 2003-2013 Dale Emery and George Dinwiddie