New Tricks with Old Tools

By Kathy Iberle and Bret Pettichord

Abstract

This paper describes how a long-standing technique for analyzing business processes can be adapted to develop effective testing strategies for complex systems. We use Structured Systems Analysis to develop a model, not of a business process, but rather of the key structural elements of a software system under development. This model then serves as a basis for designing testing strategies that can accommodate the staggered delivery of components. This approach can also be used to design test strategies for systems that contain failure-prone modules and systems assembled from off-the-shelf components.

Biography: Kathy Iberle

Kathy Iberle is a senior software test engineer at Hewlett-Packard. Over the past twenty years, she has been involved in software development and testing for products ranging from medical test result management systems to inkjet printer drivers and Internet applications. Currently, Kathy researches and develops appropriate development and test methodologies and processes for different situations. You can contact her at www.kiberle.com.

Biography: Bret Pettichord

Bret Pettichord specializes in test automation and agile testing methods. He is a lead contributor to the Watir and Selenium open-source testing tools and a co-author of *Lessons Learned in Software Testing*. He is Director of Testing Practice at ThoughtWorks, a multinational consultancy providing IT services to Global 1000 companies. Visit his website at www.pettichord.com.

First published in the Pacific Northwest Software Quality Conference Proceedings, 2005

New Tricks with Old Tools

Introduction

Have you ever planned the testing of a fully assembled system, only to find that some components haven't been delivered yet and you can't get at the parts that are finished? This paper presents a method for working around the missing components. We use a diagramming technique borrowed from Structured Systems Analysis to reverse-engineer the system design, and then use the diagram to identify ways to test the parts of the system that are finished.

What Problem Are We Trying to Solve?

The Classic Approach to Testing Complex Systems

The classic approach to testing systems consisting of multiple components built by different teams is to test smaller pieces at lower levels first, and then integrate those pieces together, test again, and so forth until one reaches the system test level. The testing is usually divided into "test levels" or "test phases" such as unit testing, component testing, integration testing, and system testing. The number of levels and the types of testing to be covered in each level depend on both the architecture of the software being tested and the development or integration plan in use. Craig and Jaskiel, in *Systematic Software Testing*, [CRAI02] briefly discuss some of the variations seen in different projects and companies.

An alternative approach to integration and testing is the "big-bang" approach – slam everything together and test the assembled system. Repeat until satisfied (or until the ship date is reached). Essentially there's only a system test level, since the only testable item is the entire system.

A third approach employs a piece-wise integration like the first approach, but integrates features rather than pieces of software. Staged Delivery and the entire family of iterative lifecycles generally follow this approach. Early versions of components are integrated together to create a system with a small feature set, and those features are tested. Later versions with more features are integrated, and the system is tested, and so testing continues throughout the project.

All these approaches share one key assumption – an assumption that is not necessarily valid on many of today's development projects. **These approaches assume that all the development teams are following a jointly developed plan for integrating and testing the system.**

- In the first approach, there is a systematic, hierarchical integration plan. All the participating development teams have agreed to test and fix their parts before integrating them, and they have agreed on an orderly integration plan. The overall test plan is designed around this integration plan, to test the results of each planned integration in a manner appropriate to a piece of that size.
- In the big-bang approach, all the teams have agreed to integrate their pieces at one pre-determined point in the schedule, at which time testing will start. There usually is some agreement on the level of maturity or finished-ness expected at this point.
- The third approach requires a detailed implementation and integration plan that is carefully orchestrated across teams to ensure that all the bits of code required for an individual feature will appear at the designated build for that feature. The test plan is designed around the feature-release plan.

In all three approaches, the development can be viewed as one large project or as a set of coordinated sub-projects.

Reality Strikes

In many situations, one of these three approaches is used. The multiple-team hierarchical approach is common in fields requiring very high-reliability software, such as telecommunications, aerospace, and medical products. The other two approaches are used in many fields, especially on smaller projects.

However, in a number of fields today, it is increasingly common to assemble large software systems out of disparate parts that come from different organizations on loosely coordinated schedules. The various suppliers may be using different lifecycles and have quite different ideas about the test coverage expected of them prior to delivery. A feature may be implemented by the sum of several software modules or components which are not available at the same time or at the same level of maturity. The integration plan is not organized into a simple hierarchy.

Response to Reality

Testers faced with non-hierarchical or unplanned integrations can't use the classic test levels, because this approach assumes that the component suppliers have agreed on the test levels applicable to each supplier and on the coverage expected at each level. Likewise, the iterative approach can easily fail when the integrations aren't coordinated thoroughly enough to make packets of user-visible features reliably "go live" at any particular point in time. The testers generally fall back on acting as if the big-bang approach has been used, even if there are multiple integrations spread over time.

In a big-bang project, the testers generally rely on testing against what the end-user can see - typically user requirements, functional requirements, and non-functional requirements.[1] Testing is usually organized around features. There is little or no testing that is targeted specifically at the software design, and often the testers don't have much visibility into the design.

Unfortunately, individual development teams often assume that the test team is intentionally testing their component, when in fact the test team is testing the feature most closely associated with that component. If the architecture is well modularized and lightly coupled, testing the feature may test the component pretty thoroughly, but more commonly, there are multiple other components involved in implementing the feature. When those components are delivered piecemeal in varying states of completion, we very often find that the incompleteness of a less-visible component is blocking testing of a feature, and thereby preventing testing of the other components which are involved in that feature.

Consider an inkjet printer. The user is notified when the ink cartridges run low on ink, when the paper jams, and numerous other conditions.

This functionality is implemented by several different components:

- Hardware – senses ink level, senses a paper jam
- Firmware – interprets the hardware signals and sends them to the computer
- Operating System – receives signals and hands them to a status monitor
- Status Monitor – reads the signals and pops up dialogs to inform the user of the printer status

From a software perspective, the status messages are a feature of the Status Monitor. However, the feature doesn't work until the hardware *and* the firmware *and* the status monitor have all implemented

---

[1] In this paper, we use Karl Wiegers' terminology for requirements: *User requirements* represent what the user wants the system to do. *Functional requirements* specify what the developers intend the system to do, once implemented. *Non-functional requirements* or *quality attributes* describe other important characteristics such as usability, portability, reliability, and so forth. [WEIG02]

status messaging support. Using a black-box approach, the software that pops up the status message for paper jams can't be tested until the sensing for paper jam is working properly. This means the testing of the status monitoring software basically can't make any progress until the hardware and firmware are nearly complete. The status monitor development team may not find this situation acceptable.

A Solution

A test team faced with this sort of problem needs to figure out how to test individual components on their own rather than relying solely on testing the entire assembled system. Yet it's not possible to tell a program that all the teams involved have to rethink their entire development approach and undertake a completely hierarchical collaborative integration plan.   It would be much more practical to do just enough lower-level testing to work around the components that are likely to be delivered late.   However, this requires some basic understanding of how the system works.

In short, a high-level design is needed. Often, the high-level design isn't available on paper – either there is very little design in writing, or there are reams of detailed information on each component but no overall high-level diagram.   The tester needs to reverse-engineer the high-level design. We have found that a technique from Structured Systems Analysis can be very useful in reconstructing the high-level system design.

What is Structured Systems Analysis?

Structured Systems Analysis is a modeling technique that was originally created to analyze and accurately describe business processes in order to automate them. The "system" in Structured Systems Analysis is any "connected set of procedures (automated procedures, manual procedures, or both)." [DEMA79], generally some sort of business process. Structured Systems Analysis is a process for producing diagrams that are concrete, readily understandable specifications of the process under study at several levels of detail.[2]

Structured Systems Analysis was popular in the 1980s. The description of structured analysis used in this paper is taken from *Structured Analysis and System Specification* by Tom DeMarco [DEMA79].

Why Use Structured Systems Analysis?

Structured Systems Analysis works for reverse-engineering a high-level design because a computer system and a business process are quite similar at this level - they both consist of loosely coupled, dissimilar activities which provide information to each other. When we're reverse-engineering a system design, we need to elicit a description of a complicated system from the people most familiar with that system, which is exactly what Structured Systems Analysis does. An added advantage is that the Structured Systems Analysis notation is designed to be easily understood without special training, so the people most familiar with the system can easily review and correct the description whether or not they know Structured Systems Analysis.

[2] Using today's modeling languages, the process might be drawn as a swim lane diagram, and the user requirements captured with a combination of use cases and business rules.   The Structured Systems Analysis approach bears a strong resemblance to a process map.

How to Use Structured Systems Analysis

The Structured Systems Analysis process is pretty straightforward:

- Use structured analysis questioning techniques to elicit information from developers.
- Record the findings in a readily understandable form – the simple graphic notation known as a "data flow diagram".
- Review the system design diagram with developers and correct as needed.
- Use the design diagram to identify ways to work around missing parts of the system – identify what you can test and what is needed to enable that testing on an unfinished system.

We'll work through an example using these steps.

Sue and the Acme Automated Grocery Checkout Stand

Sue is the lead test engineer for the software project Gecko, which is part of the Reptile program. Sue's team has been newly assigned to this program. She sets out to find out what this "checkout stand" is all about.

First, Sue locates the Project Data Sheet. Like most Project Data Sheets, this one bubbles over with enthusiasm and optimistic predictions.

Project Reptile

Product Summary: Reptile 3.0 is the newest, most advanced automated grocery checkout stand. Reptile lets shoppers scan and bag their items themselves, totals up the cost, and allows the user to pay for it via cash or credit card. The average grocery store will save millions of dollars in labor costs per year by making a small investment in this state-of-the-art easy-to-use and very reliable system.

New features: Reptile 3.0 will be able to process produce and other items that are sold by weight.

Teams:
Lizard: scale for weighing produce
Caiman: barcode scanner
Vincent: credit card reader and cash acceptor
Gecko: software

Sue starts out by establishing what a checkout stand is, and is not, using Structured System Analysis techniques.

The Level 0 Data Flow Diagram

The usual starting point for systems analysis is to consider the entire system or product as "inside" and everything else as "outside". Structured Analysis calls this viewpoint the Level 0 Data Flow Diagram (DFD) or Context Diagram.

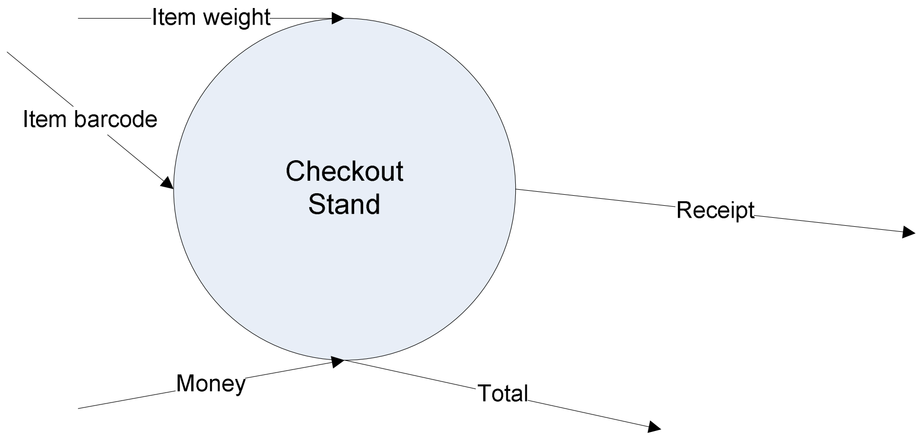
The rules for drawing a Level 0 DFD are very simple.

- Draw one bubble for the process
- Draw all the data that comes into it or out of it. Don't forget the user as a source or destination for data.
- Label all the arrows with the name of the data.

Sue starts by sketching out the Level 0 DFD for using the checkout system, based on the Project Data Sheet.

She identifies input: barcodes and item weights. Oh yes, and money.

She identifies output: a total (otherwise the shopper wouldn't know how much money to give the machine) and, though the data sheet doesn't mention it, there must be a receipt.



The Level 0 DFD establishes the inputs and outputs. This is the same point of view taken by a pure black-box test. The system has inputs, and outputs, and whatever goes on inside is a mystery.   So far, Structured Systems Analysis hasn't told Sue anything that she didn't already know.

Sue could stop at this point and create the test plan without understanding anything more about the internals of the checkout stand. A simple black-box test approach calls for applying input and checking the output – this would translate to scanning, bagging, and paying for various collections of items. However, Sue is suspicious that this straight-forward approach is going to run afoul of the multiple development teams. She's already hearing that the team working on the scale is running behind. Sue decides that it would be prudent to know a bit more about how the system works. She sets out to establish the basic system architecture.

Drawing the Level 1 DFD

In structured analysis, the analyst discovers the workings of an existing process by asking questions of the users, drawing a Level 1 DFD, and then asking the users to confirm the accuracy of the diagram. Sue uses the same process to discover the workings of the checkout stand by questioning the developers.

The notation:

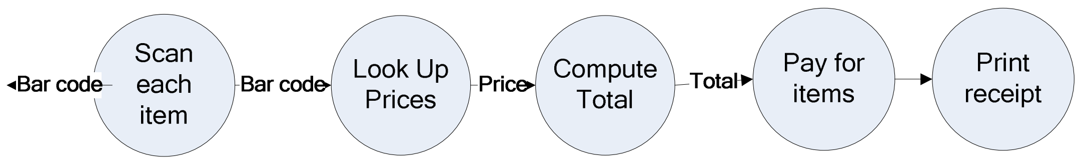
- The bubble: a person or system doing something. Labeled with a **verb**.
- The labeled arrow: data traveling from one bubble to another. Labeled with a **noun**.
- The straight line or cylinder: a file, database, or other permanent data store.

The rules:

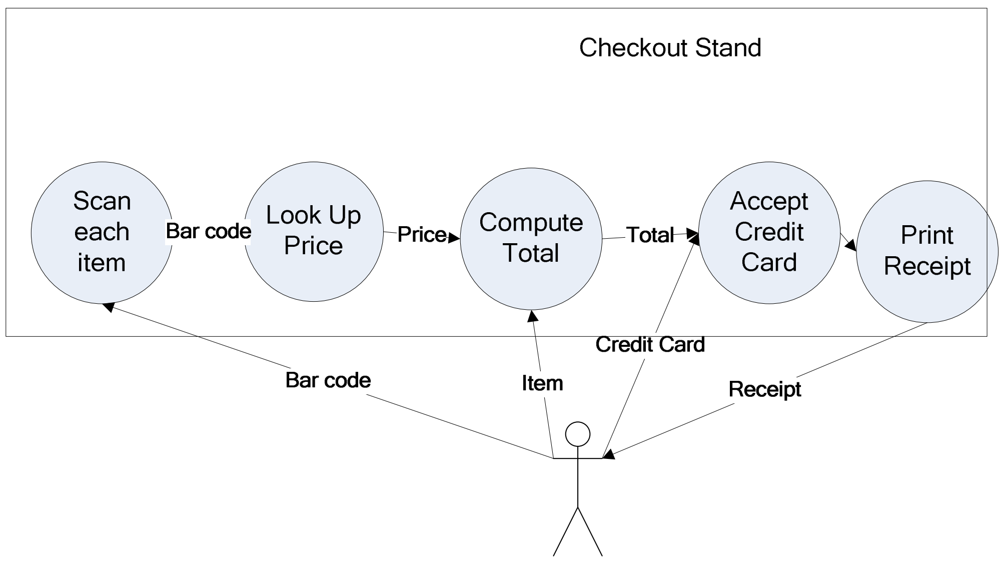
- The data flow diagram always shows flow of data, not of control.
- Decision points are not shown.

Sue interviews Kent, the system architect. Her first sketch is just to establish what the system does.

Sue: "What happens first?"
Kent: "The shopper scans each item."
Sue (draws a bubble for scanning items): "Then what?"
Kent: "The checkout stand looks up the price."
Sue (draws a bubble for looking up the price): "Then what happens?"
Kent: "Well, just keep scanning things until you're done, and then the system computes the total."
Sue (draws a bubble for computing the total): "And then what?"
Kent: "Then the shopper pays for the items and gets a receipt."
Sue busily draws the last few bubbles.
Sue: (showing this picture) is this what happens?



Kent: "Uh, the system doesn't pay for items. The shopper pays".
Sue: "Okay, so what goes into the system when the shopper pays?"
Kent: "You gotta put some money in, or a credit card, or a debit card."
Sue: (erase and redraw) "How about this? Now the checkout stand is doing all the bubbles, and there's a stick person for the shopper".



Sue's first drawing didn't distinguish between the user and the computer system. In her second drawing, the user appears as a stick figure[3] and there's a box is drawn around the system as a whole. Notice that the information going into the box and coming out of the box is the same as the information going into and out of the bubble in the Level 0 diagram. The Level 0 diagram can usually be skipped, but if there are questions about the input and output, it's sometimes worthwhile to draw the Level 0 diagram because the data in and out is more obvious in the Level 0 form than in the Level 1 form.

Anything displayed to a user – numbers, error messages, pretty pictures – is output because it goes out of the system to a human being, who is not part of the system. Likewise, anything that a user communicates to the system – mouse clicks, typed inputs – is input. Generally it's more useful in the diagram to depict mouse clicks as the logical input, such as a menu pick, rather than just showing "mouse click".

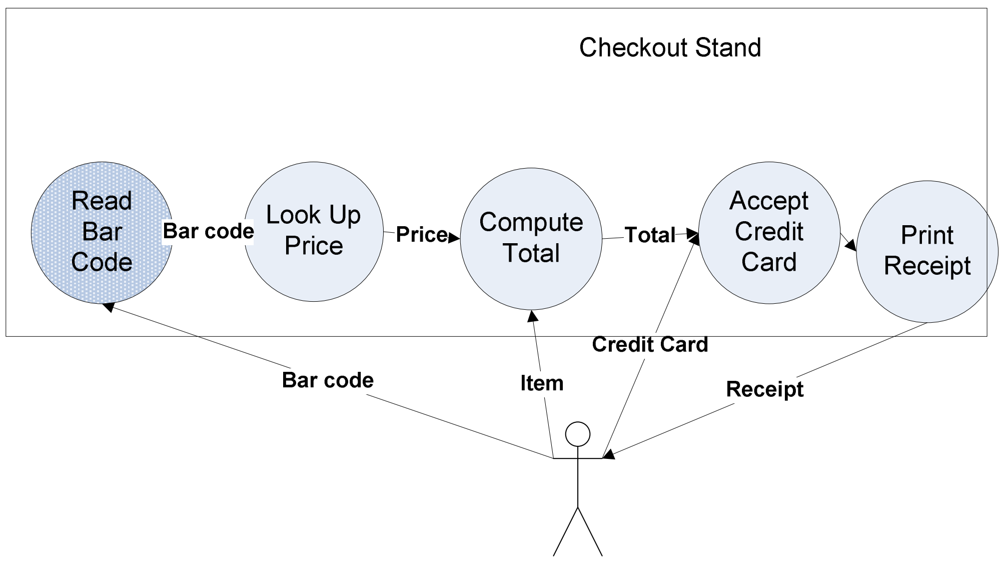
Note that Sue drew the use case where the shopper pays with a credit card. She didn't try to draw a use case where the shopper pays with cash, because this would involve more output (the shopper's change). When you're trying to understand the basic structure of the system, it's often easier to draw the system needed for the simplest use case first.

Establishing the System Structure

Sue considers her diagram. It's pretty obvious that the software inside the stand isn't divining the bar code via ESP. There must be a physical sensor, and some firmware that runs the sensor and passes numerical data to the software proper. Sue redraws the diagram, showing the hardware/firmware tasks in a different color. She goes back to talk to Kent again to confirm her speculations. She also invites Yumiko, the technical lead for the team that's writing the main processing software.

---

[3] The stick figure isn't actually part of the original Structured Systems Analysis notation. In the 1980s, the diagrams were drawn by hand, and so the symbols were kept to a minimum.



Kent and Yumiko look over the diagram.

Yumiko: "Why aren't you using UML[4] for this? This isn't a UML diagram."

Sue: "UML doesn't really have a diagram equivalent to this. An activity diagram is similar, but that's got timing and control flow shown which makes it harder to draw and harder to understand. So I use this notation because it's easier."[5]

Yumiko: "Well, OK, I guess we can do it this way."

Kent and Yumiko mull over the diagram for a bit.

Yumiko: "That Look-Up-Price bubble isn't right. Really there's a database, and the data-retrieval dll calls the dbinf dll, then that calls up the database and gets the price."

Kent: "And the Compute-Total calls the database too, to get the sales tax rules."

Sue: "Those dlls are running on the processor inside the Reptile checkout stand, right? And the database is outside the checkout stand somewhere?" (Sue draws the database outside the box)

Yumiko: "Yes. The database isn't part of the Reptile project, really."

Kent: "Where's the produce handling? I don't see where you can enter the SKU. And we've got the scale for weighing the produce – that's not there either."

Sue: "Oh dear, you're right." (busily adding bubbles to the diagram) "There's not any hardware keypad, so the software must be providing the keypad for entering the SKU, is that right?"

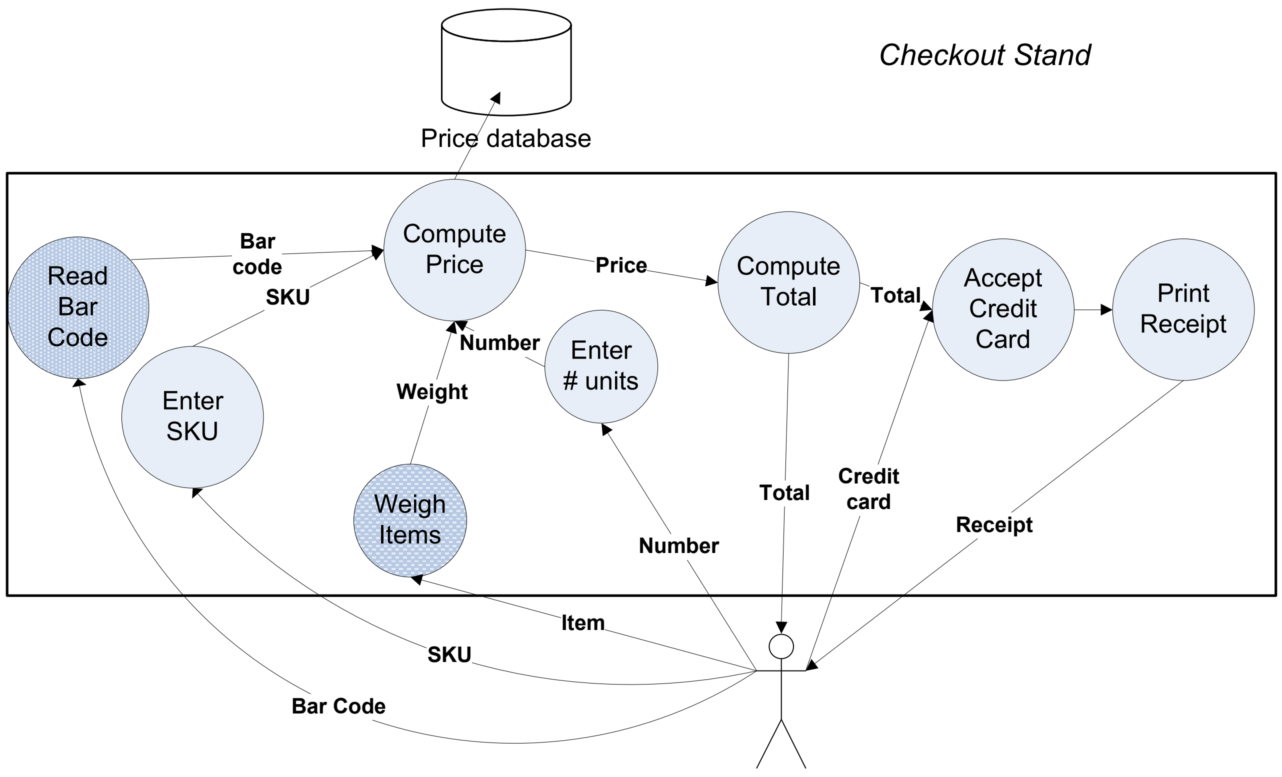
Yumiko: "Yes."

Sue: "And is there a way to buy produce like three for a dollar?"

Yumiko: "Yes, there's a screen for entering the number of items." (pulls a piece of paper out of a pile on her desk) "Here's a workflow diagram – it shows the screens and how they flow into each other."

Sue revises her diagram again.

[4] Unified Modeling Language
[5] Also, understanding data-flow diagrams doesn't require being familiar with UML.



A few things to note:

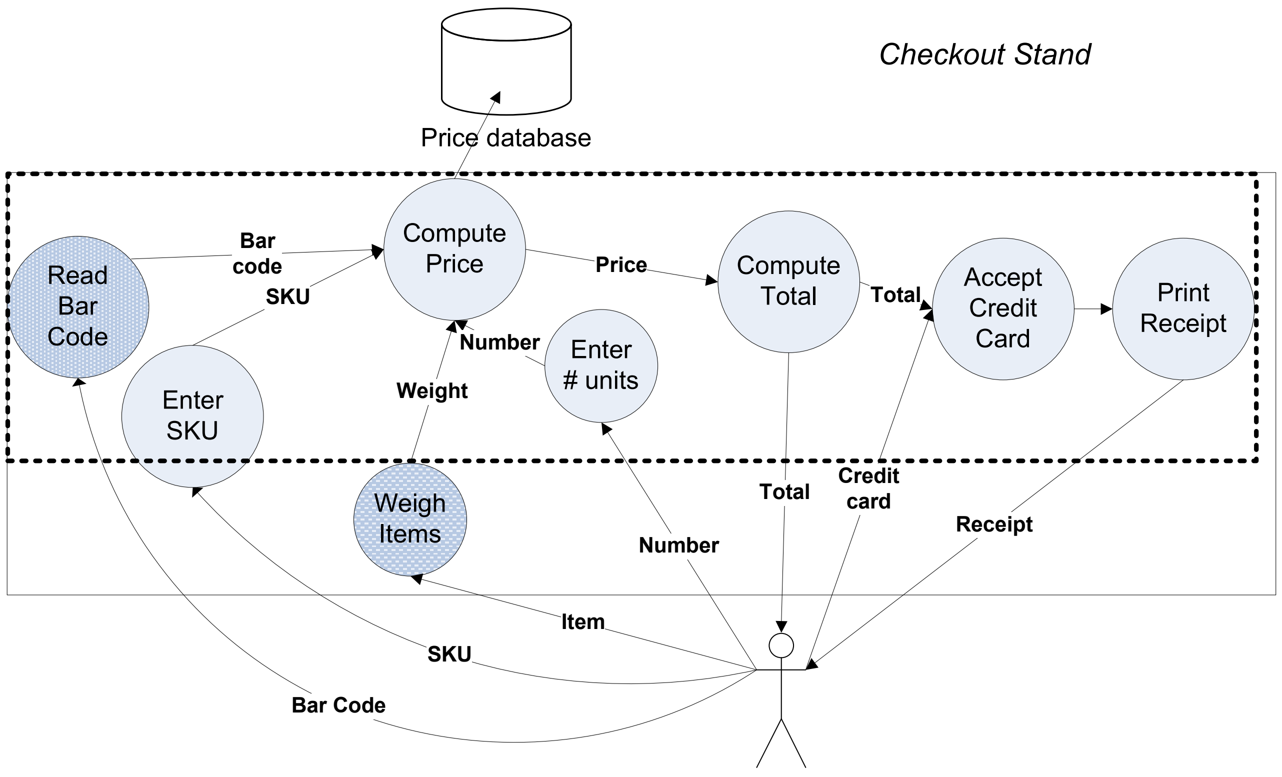
- The database isn't considered to be part of the project, so it's drawn outside the automated checkout stand
- The developers mentioned some dlls by name, but Sue didn't draw them. The data-flow diagram is intentionally kept at a high level of abstraction, often higher than the actual software architecture. The actual software design is often too complicated to help in planning testing at this level.
- The workflow diagram showed that some of the bubbles contained several different screens. In DFD notation, the activity inside a bubble is shown on a separate sheet as a Level 2 DFD. Sue doesn't go to Level 2 at this point because she doesn't see a need for it, although she keeps the workflow diagram to use later in writing test cases.
- The DFD could be simplified by merging some of the bubbles, but Sue didn't do that. For instance, Read-Bar-Code and Enter-SKU could logically be represented as a single bubble Enter-Identifier. Sue hasn't merged those two bubbles because those parts of the system are delivered by different teams, and that might affect the test plan.

The Test Plan

At this point, Sue has enough information to begin laying out a test plan for her team's software. She sets two people in her team to writing test cases based on the functional specifications for the screens. Another person arranges for builds to be received by her team and assembles a test system.

Shortly, one of her staff shows up at Sue's desk. The team delivering the scale for weighing the produce has fallen behind, and as a result the test system can't weigh any produce. The produce handling is a new feature, and therefore one of the features they'd planned to test first. What should they do?

Sue sits down with her diagram. She draws a line isolating the offending scale from the rest of the system. She considers first testing only the items inside the box, then adding in the scale and completing the testing. The early testing would provide information to the software development teams much earlier and thus help the project proceed.



How can they run the system without the hardware and firmware represented by Weigh-Item?

Divide and Conquer: Test Scaffolding

If Sue can find a way to feed weights to the system without the scale, then the routines for pricing produce by weight can be tested without waiting for the bar code reader. Her team needs some *test scaffolding*.

There are several ways to build test scaffolding. Some of them require help from the development team, and others can be carried out by the test team.

USE OLDER SOFTWARE

On projects that are building on pre-existing code, it's often possible to replace an unfinished piece of the system with the older, known good version of that piece and continue testing the rest of the system. There are several drawbacks:

- Whatever bugs are still in the older version can show up during testing and obscure the results of testing the new code
- If the interface definitions have changed, using older software often won't work at all.
- The older software may not be able to supply all possible inputs to the rest of the system – the

new version may have an expanded range.

- People outside the test group often have trouble understanding that it's ok to use an old version of A if you're concentrating on testing B. This is particularly true in groups where all the testing up to this point has been at the system level, so they see no distinction between testing all the code in module B and testing the features in which module B plays a key part...

In this case, there is no older Weigh-Item software, so this isn't going to help.

WRITE DRIVERS AND STUBS

Another way of getting scaffolding is to write it. There are two basic types of test scaffolding, both of which are often referred to as *test harnesses*.

- A *test driver* or *test harness* calls the software under test and passes it the desired data. Typically a test driver will have some kind of simple interface to allow the tester to specify data, either on the command line or by supplying a text file.
- A *stub* is called by the software under test, to permit it to run in the absence of the stubbed-out software. Often stubs do nothing except return success.

Custom scaffolding avoids the disadvantages of using the older software as scaffolding. The main disadvantage is that the scaffolding has to be written. However, often developers have written some scaffolding for their own use during development and are willing to share that with the test team.

On a project with multiple development teams, the test department may find that they need to tell the developers what scaffolding is needed, because the test department is first to know which parts of the system are missing and need to be worked around. The scaffolding can be identified by looking at the DFD. Wherever a data flow crosses the line between the offending component and the rest of the system, there is data flowing either into or out of that component to another part of the system or to the user. These data flows need to be replaced by drivers or stubs. Sometimes both are combined in a single program, and other times there are several bits of code needed.

Sometimes the developers can't quite see how to create a test harness for a tricky bit of code. There is a good introduction to writing test harnesses in Steve McConnell's *Code Complete* [MCCO04]. Michael Feathers presents a variety of design patterns for getting test scaffolding around classes and methods in *Working Effectively with Legacy Code* [FEA05]. Feathers' book is particularly useful in our situation because it assumes that the developer is trying to put scaffolding in place in a few targeted spots, rather than assuming that all the code will have automated unit tests.

HIJACK THE INPUT

Input that is coming from any static data store – a file, registry table entries, a database - can easily be hijacked by changing the file or the registry table.

There is the obvious method of simply substituting one input value for another in the data store. For instance, when a program reads initialization values from an "ini" file, it's easy to change that file.

There's a sneakier method, as well, of hijacking the data midstream to force the system to respond as if a different initial input value had been supplied. For instance, we'd like to test the system with items that are priced \$100.00 or more. But all the bar code stickers we have on hand are for items that are much less expensive. We can go into the Price database and change the price of a can of soup to \$105.21. Then scan the can of soup, and the system is off and running to exercise the entire data path involving prices greater than \$100.00.

The data-flow diagram can be used to determine whether there is any opportunity to hijack data by tracing the data-flow from the initial input to the item-under-test. If the data passes through any static data stores, then additional inputs to the item-under-test can be created by hijacking the data in the data store.

You can get pretty creative if there's a lot of information in static data stores. Sometimes it's possible to change processing rules as well as data to send the system off on the path of your choice. Usually a tester with command-line access to the system and perhaps a few key passwords can make these changes without any development help.

Incidentally, it's pretty common to find systems that include a database server which is not being changed at all and therefore is not considered to be under test. Sometimes the testers are expected to use the "production" database in the testing, and the clients are configured to call that production server. This eliminates the ability to set up interesting test data and to hijack data-flows. A "test" database server which can be freely changed by the test team will speed up testing the rest of the system. (This assumes you have a tester with the skills to change data in a database management system.) Whenever possible, get the developers to set up the client such that the test team can easily redirect the client to a different server.

Remember that any changes you make to static data stores are part of your tests. If you think there'll be a need to re-run the same test or a similar test, then you'll want to save the modified files, or scripts used to make the modifications, or instructions for making the modifications by hand.

Back to Sue and her Team

Sue has to decide which of the above patterns to try. She takes a careful look at the data-flow diagram to understand what part the missing piece plays in the system.

The "Use Older Software" pattern is the easiest to implement, but it's obviously not going to work because there is no older version of Weigh-Item.

The "Hijack the Input" pattern is usually the next easiest to implement, so Sue takes a look at her data-flow diagram. The data-flow diagram doesn't show the data from Weigh-Item going through any static data stores before entering Compute-Price, so apparently there's no way to hijack the input.

Sue thinks about writing drivers and stubs. The people who wrote Compute-Price can't have had the scale and associated firmware while they were writing Compute-Price, since it hasn't been built yet. But they presumably at least tried out the data path for computing the cost of items sold by weight. How did they accomplish this?

Sue goes over to talk to the developers.

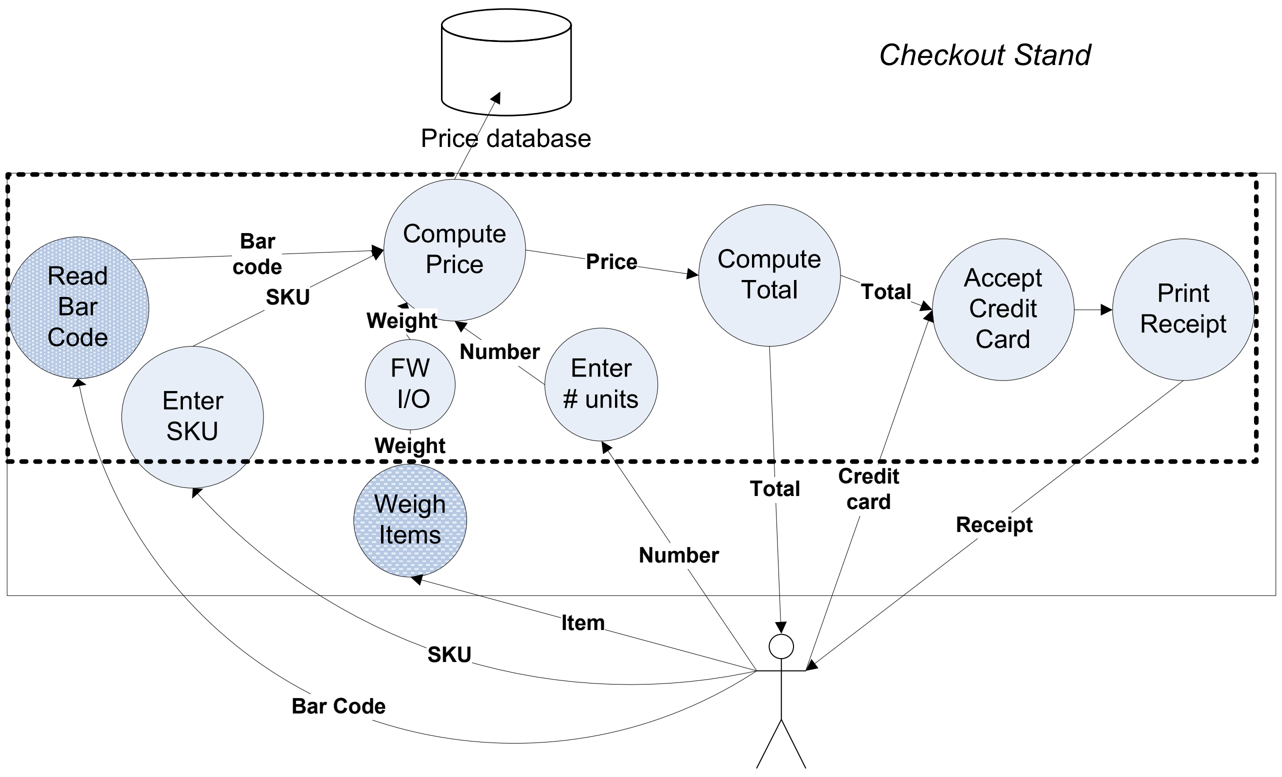
Kent: "Well, Compute-Price isn't talking directly to the firmware, of course. There's a class that interfaces with the firmware."

Yumiko: "I wrote a quick substitute class which pops up a dialog instead of calling the firmware, so I can type in the weight."

Sue: "Can we get that substitute class into the build until the firmware for the scale is ready?"

Yumiko: "I guess so."

Sue has to redraw her diagram again, but now it's clear that there is a way to get input into the system without the scale.



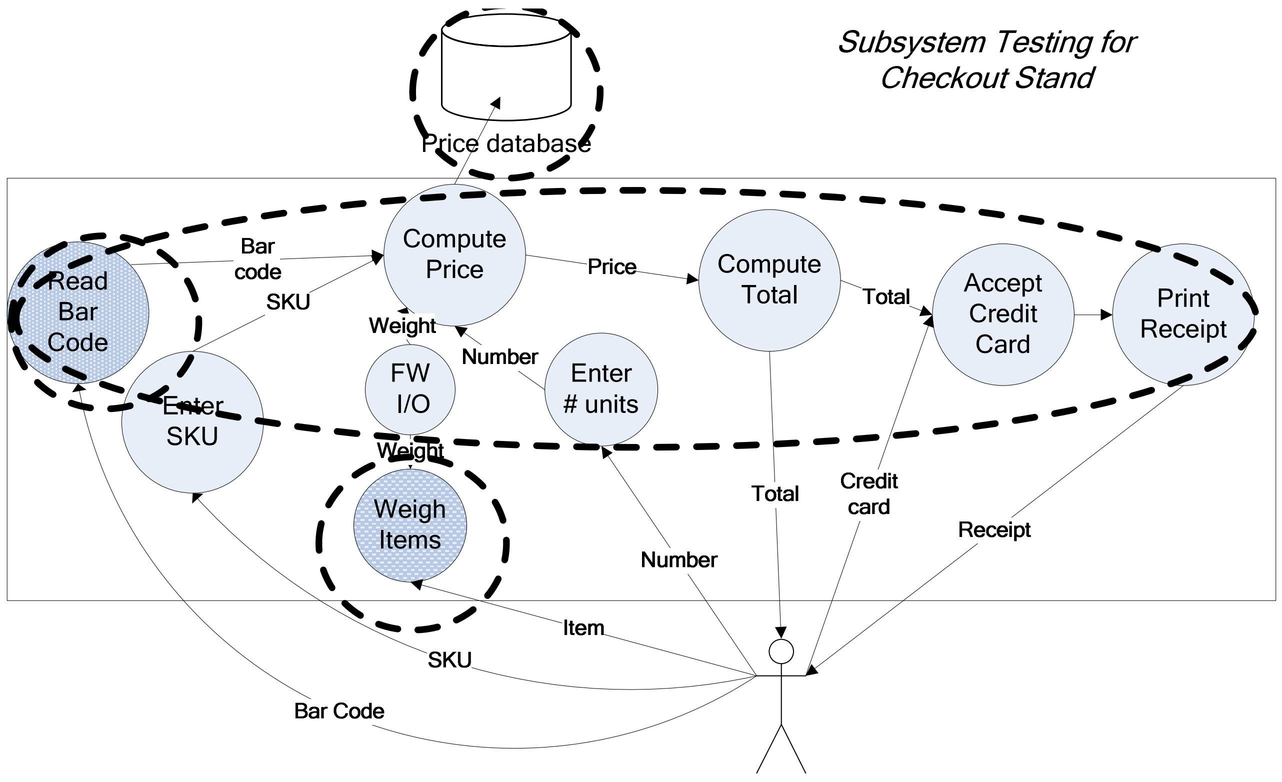
Defining the Test Phases

Sue returns to her test planning. She splits the testing into two parts – testing that will be run on the complete assembled system, and testing that will be run on the software minus the Weigh-Items firmware/hardware. Sue has just defined two test levels – a system test level, and a subsystem test level.

In a system with many parts, it's not unusual to find that there are multiple test efforts run at various levels of integration. Sometimes testing within one team is split into levels or phases, as Sue has just done. Often there is also testing done by some of the supplier teams. In the case of the automated checkout stand, the firmware teams are running their own testing using scaffolding, and the team supplying the Price database has done extensive independent testing.

Sue shows this partitioning in her master test plan by drawing circles around the items tested by each test set. There are four test sets at the subsystem level:

- Price database testing
- Bar code reader testing (firmware)
- Scale testing (firmware)
- Testing of system minus Weigh-Items



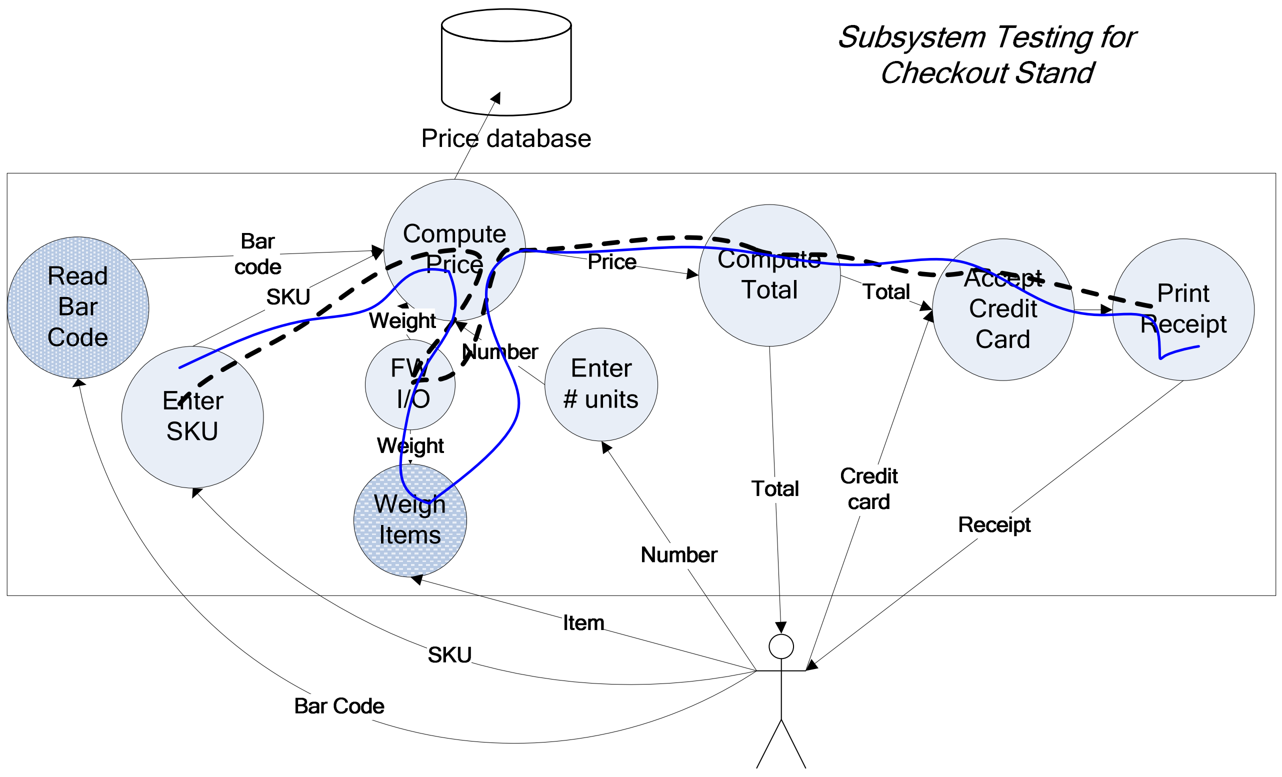
Some people would argue that the first three are a sub-subsystem or component level test, and the fourth is a subsystem level test. This is often a pointless argument, because many systems don't sort out neatly into a hierarchy, and the items-under-test often end up overlapping with each other. In such cases, a drawing of the partitioning usually communicates far more than classifying tests as "subsystem" or "component".

Sometimes the integration is relatively hierarchical, so testing can be split into the traditional system-subsystem-component test levels. Even in this case, if the parts are very different from each other, the test purpose and techniques at a given level may be considerably different for different components. For instance, the database testing and the firmware testing for the checkout stand probably don't resemble each other very much. This means that the content or purpose of "component" testing can't be defined much more precisely than "testing of an item that's bigger than a breadbox and smaller than the whole system". You may end up defining "component testing" as "testing of an item that is the output of a single team". Again the drawing can be much more communicative than a written description of the test levels.

Subsystem vs. System Testing

Sue and her team are able to test most of the functionality of the system with Weigh-Items missing. Once Weigh-Items does arrive, what still has to be tested?

One way to look at this is to consider end-to-end paths or execution threads. You can use the data-flow diagram to keep track of the paths by drawing them on the diagram in different colors or weights and labeling them.



- o Black dotted line: test cases run during the subsystem test
- o Blue solid line: test cases run during system test

There will be a variety of test cases that follow the black dotted line, using different inputs or making different choices along the way. Rather than simply repeating all these test cases on the fully assembled system, it's smarter to focus on the newly added piece and consider what could be broken. Heuristics for integration testing are usually useful. For instance, can the newly added item send input values to the rest of the system which were not used in the subsystem test?

The key interface here is the interface between the Weigh-Item firmware and the Firmware I/O software (which communicates with the firmware). It would be smart to test a range of weights, check the highest and lowest possible weights, and take a look at the expected precision. Somewhere in the scale or the firmware an event in the physical world is converted into a digital signal, which is then sent to the software. The software may well receive an unexpected input because something that wasn't planned for happened in the physical world. What happens if the harried shopper sets his toddler on the edge of the scale? What if you drop a heavy can on the scale from six inches up?

Other Uses for Data-Flow Diagrams

Now that Sue has solved the problem of the missing scale, and has her test plan in place, she doesn't have any more immediate need for the data-flow diagram. However, there are some other situations in which the data-flow diagram may come in handy.

Data-flow diagrams will sometimes make it obvious that a number of test cases are probably equivalent to each other.

Data-flow diagrams can be used to find defects by walking through the system design. I've more than once found significant design problems during the drawing of the data-flow diagram. I'll ask, "So how does this talk to that?", the answer will be rather vague, and I'll ask again. The developer will suddenly stop being ever so patient with my ignorance, stare abstractedly into space for a few minutes, and then announce that the system isn't going to work. This usually means I have to come back the next day to finish the data-flow diagram, as the developers are distracted by figuring out how to solve the design problem.

Another type of diagram similar to the data-flow diagram is the Deployment Diagram (also know as architectural diagrams, block diagrams, and parts maps). These generally are applicable to web applications but not to other types of systems. The Deployment Diagram is showing physical pieces of the system rather than tasks getting done, but it resembles a data-flow diagram anyways, because the system generally is implemented such that each physical piece does a different job. The operation of the system isn't entirely evident from the Deployment Diagram because the Deployment Diagram typically doesn't say what those jobs are, nor what data is being transferred from piece to piece.

Elisabeth Hendrickson [HEND02] shows how to derive a variety of test cases from a Deployment Diagram, many of them error recovery or stress tests. The thinking process described by Hendrickson should work just as well with a data-flow diagram as with a Deployment Diagram.

Conclusion

Data-flow diagramming from Structured Systems Analysis, despite being an old and hoary technique, is a handy tool to have in one's testing toolbox. This is especially true for systems that are constructed out of disparate parts in a decentralized (or even uncoordinated) project or program, because data-flow diagramming helps the test planner figure out how to test partially finished systems whether or not the integration plan is a neat traditional hierarchal plan.

References

[CRAI02]: Craig, Rick D. and Jaskiel, Stefan P; *Systematic Software Testing*; 2002
[DEMA79]: DeMarco, Tom; *Structured Analysis and System Specification*; 1979
[FEA05]: Feathers, Michael C; *Working Effectively with Legacy Code*; 2005
[FOWL04]: Fowler, Martin; *UML Distilled, 3rd edition: A Brief Guide to the Standard Object Modeling Language*; 2004
[HEND02]: Hendrickson, Elisabeth; "A Picture's Worth a Thousand Words"; Software Testing and Quality Engineering Magazine; Sep/Oct 2002' pp. 26-32
[MCCO04]: McConnell, Steve; *Code Complete, 2nd ed*.; section 22.5 Test Support Tools
[WEIG03]: Wiegers, Karl; *Software Requirements, 2nd edition*; 2003;