CHAPTER 4
Exploratory Testing in the Large

"A good traveler has no fixed plans and is not intent on arriving."

—Lao Tzu

Exploring Software

The techniques presented in the preceding chapter help software testers make the numerous small decisions on-the-fly while they are running test cases. Those techniques are good for choosing among atomic inputs and arranging atomic inputs in combination or in sequence. This chapter is about the larger decisions testers must make concerning feature interaction, data flows, and choosing paths through the UI that result in making the application do real work. Instead of decisions regarding atomic inputs that serve some immediate purpose on a single input panel, we'll reason about inputs to guide us toward some larger goal. An exploratory tester will establish such goals in advance of actual testing and let the goal guide her during testing sessions. We accomplish this with a tourism metaphor that treats exploration of software using the tools of a conventional tourist: organized tours, guidebooks, maps, and local information. This helps us set goals that can then guide our decision making during testing.

In *How to Break Software* and its series companions,[1] I used a military metaphor to describe software testing. Given that those books were aimed exclusively at breaking software, it was a metaphor that helped get the attack-minded test strategies of those books into the heads of readers. I've received overwhelmingly positive feedback from my readers that they found the metaphor helpful (and fun!), and so I am encouraged to use the same approach, with a different metaphor, for this book whose purpose is much broader.

---

[1] Whittaker, *How to Break Software* (Addison-Wesley, 2003); Whittaker and Thompson, *How to Break Software Security* (Addison-Wesley, 2004); Andrews and Whittaker, *How to Break Web Software* (Addison-Wesley, 2006).

Metaphors can be a powerful guide for software testers.[2] This is the exact purpose we want to achieve: a metaphor that will act as a guide to help testers choose the right input, data, states, code paths, and environment settings to get the most out of their testing time and budget.

Testers must make many decisions while they are testing. There are big decisions, like how to obtain realistic data to simulate customer databases. There are small decisions, like choosing what string of characters to enter in a text box. Without the proper mindset and guidance, testers can end up wandering aimlessly around an application's interface looking for bugs that may or may not actually be there and gaining only poor coverage of the application in the process.

I often tell testers I work with that if it feels like you are on a ghost hunt during a testing session, you probably are.[3] If you feel like you are wandering around looking for shadows, stop testing and try to establish some better guiding goals for your effort.

This is where the metaphor can help by providing a strategy and a set of specific goals for software testers to follow, or at least keep in the back of their mind, as they are exploring an application. For an explorer, having a goal to pursue is better than just wandering around. A guiding metaphor should give testers a goal and then help them understand how to perform testing that fulfills that goal. If a metaphor yields goals that help testers make both big and small decisions, then a testers aren't just wandering aimlessly. The metaphor has helped them organize their testing to approach software's complexity and its broad range of features in a more methodical way. This chapter is about using a metaphor, specifically a tourism metaphor, to make large test decisions that will guide overall exploratory strategy and feature usage.

In finding a metaphor that works for exploratory testing, it is important to understand the spirit and intent of exploratory testing so that we ensure the metaphor is helpful. The goals of exploratory testing are as follows:

---

[2] However, the wrong metaphor can be distracting. It's an interesting piece of testing history that provides a case in point. The technique of estimating the number of fish present in a lake was used as a metaphor leading to the concept of *fault seeding* in the late 1980s and early 1990s. The idea was that to estimate the number of fish in a lake, one could stock a fixed number of a specific type of fish. After a bit of fishing (testing) where some number of real fish and some number of seeded fish are caught, one can estimate the number of real fish using the ratio of seeded fish caught to the total number of seeded fish. The fact that this technique has been relegated to the dusty libraries of testing past is enough proof that the metaphor was less than useful.

[3] Ghost-hunting television shows are popular in the U.S. Teams of paranormal experts (is such a thing possible?) search old buildings and graveyards for what they deem as evidence of the supernatural. My kids like the shows, and I often watch them in their company. The ghost stories are great, but the experts never confirm paranormal presence. They never rule it out either. This may work for entertainment, but it is not much use for testing. If you can neither confirm nor deny the presence of a bug in your software (or whether it is working as specified), you aren't doing a very good job testing. Perhaps, instead, you should start a television show about the process instead.

- **To gain an understanding of how an application works, what its interface looks like, and what functionality it implements:** Such a goal is often adopted by testers new to a project or those who want to identify test entry points, identify specific testing challenges, and write test plans. This is also the goal used by experienced testers as they explore an application to understand the depth of its testing needs and to find new unexplored functionality.
- **To force the software to exhibit its capabilities:** The idea is to make the software work hard and to ask it hard questions that put it through its paces. This may or may not find bugs, but it will certainly provide evidence that the software performs the function for which it was designed and that it satisfies its requirements.
- **To find bugs:** Exploring the edges of the application and hitting potential soft spots is a specialty of exploratory testing. The goal is purposeful, rather than aimless, exploration to identify untested and historically buggy functionality. Exploratory testers should not simply stumble across bugs, they should zero in on them with purpose and intent.

Even as real explorers seldom go about their tasks without some planning and a lot of strategy, exploratory testers are smart to do the same to maximize the potential that they look in places where functionality is complex, users are likely to tread, and bugs are more likely to exist. This is a broader mission that subsumes just breaking software, and it deserves a new metaphor. The one I think works best is that of a tourist trying to explore a new destination. I like to call this the "touring test" in honor of Alan Turing, who proposed the original Turing test.

The Tourist Metaphor

Suppose you are visiting a large city like London, England, for the very first time. It's a big, busy, confusing place for new tourists, with lots of things to see and do. Indeed, even the richest, most time-unconstrained tourist would have a hard time seeing everything a city like London has to offer. The same can be said of well-equipped testers trying to explore complex software; all the funding in the world won't guarantee completeness.

How does a savvy tourist decide whether car, underground, bus, or walking is the right method of getting around London? How can you see as much of the city as possible in the time allocated? How can you cover the most activities with the shortest commute between them? How can you make sure you see all the best landmarks and attractions? What if something goes wrong, who do you go to for help? Should you hire a guide or figure things out for yourself?

Such tours require some strategy and a lot of goal setting. Goals will affect how tourists plans their time and will determine what parts of the city they will see. A flight crew on an overnight layover will approach their

tour much differently than the organizer of a troupe of visiting students. The purpose and goals of the tourist will weigh heavily in the actual touring strategy selected.

On my first trip to London, I was alone on a business trip and chose to simply walk the streets as my exploration strategy. I didn't bother with guidebooks, tours, or any other guidance beyond a vague notion of trying to find cool things. As it turns out, cool things are hard to avoid in London. But despite walking all day, I missed many major landmarks completely. Because I wasn't sure where anything was, I often saw things without appreciating them for what they were. That "awesome church" was actually Saint Paul's Cathedral, and its significance and history went unrecognized and unappreciated. When I tired of walking and resorted to the underground, I lost track of distances and haphazardly surfaced from the subway without really understanding where I was, where I had been, and how little ground I had actually covered. I felt like I had seen a lot but in reality barely scratched the surface. From a testing point of view, having such a false sense of coverage is dangerous.

My London tourist experience is a fairly common description of a lot of manual and automated testing[4] I see on a regular basis. I was a freestyle tourist, and had I not been lucky enough to return to London on many future occasions and explore it more methodically, I would have really missed out. As testers, we don't often get a chance to return at a later date. Our first "visit" is likely to be our only chance to really dig in and explore our application. We can't afford to wander around aimlessly and take the chance that we miss important functionality and major bugs. We have to make our visit count!

My second trip to London was with my wife. She likes structure, so she bought a guidebook, filled her (actually, my) pockets with tourist brochures, booked the Big Red Bus Tour, and paid for various walking tours guided by local experts. In between these guided tours, we used my method of wandering aimlessly. There is no question that the tours took us to more interesting places in much less time than my wandering. However, there was synergy between the two methods. Her tours often uncovered interesting side streets and alleys that needed additional investigation that my freestyle methods were perfectly suited as follow up, whereas my wandering often found cool places that we then identified guided tours to explore more thoroughly. The structure of the guided tour blended seamlessly with the freestyle wandering technique.

Tourism benefits from a mix of structure and freedom, and so does exploratory testing. There are many touring metaphors that will help us

---

[4] There is no fundamental difference in designing automated tests and designing manual tests. Both require similar design principles with the primary difference being how they are executed. Poor test design principles will ruin both manual and automated tests with automated tests simply doing nothing *faster*. Indeed, I maintain that all good automated tests began their lives as manual tests.

add structure to our exploration and get us through our applications faster and more thoroughly than freestyle testing alone. In this chapter, we discuss these tours. In later chapters, we actually use the tours as part of a larger exploratory testing strategy. Many of these tours fit into a larger testing strategy and can even be combined with traditional scenario-based testing that will determine exactly how the tour is organized. But for now, all the tours are described in this chapter to "get them inside your head" and as a reference for later when we employ them more strategically.

"Touring" Tests

Any discussion of test planning needs to begin with decomposition of the software into smaller pieces that are more manageable. This is where concepts such as feature testing come into play, where testing effort is distributed among the features that make up the application under test. This simplifies tracking testing progress and assigning test resources, but it also introduces a great deal of risk.

Features are rarely independent from each other. They often share application resources, process common inputs, and operate on the same internal data structures. Therefore, testing features independently may preclude finding bugs that surface only when features interact with each other.

Fortunately, the tourist metaphor insists on no such decomposition. Instead, it suggests a decomposition based on *intent* rather than on any inherent structure of the application under test. Like a tourist who approaches her vacation with the intent to see as much as possible in as short a period of time as possible, so the tester will also organize her tours. An actual tourist will select a mix of landmarks to see and sites to visit, and a tester will also choose to mix and match features of the software with the *intent to do something specific.* This intent often requires any number of application features and functions to be combined in ways that they would not be if we operated under a strict feature testing model.

Tourist guidebooks will often segment a destination into districts. There's the business district, the entertainment district, the theater district, the red light district, and so forth. For actual tourists, such segments often represent physical boundaries. For the software tester, they are strictly logical separations of an application's features, because distance is no real issue for the software tester. Instead, software testers should explore paths of the application that run through many features in various orders. Thus, we present a different take on the tourist guidebook.

We separate software functionality into overlapping "districts" for convenience and organization. These are the business district, the historical district, the tourist district, the entertainment district, the hotel district, and the seedy district. Each district and their associated tours are summarized here,

and then tours through those districts are described in the sections that follow:

- **Business district:** In a city, the business district is bounded by the morning rush hour and evening commute and contains the productive business hours and after-work socials. In the business district, there are the banks, office buildings, cafes, and shops. For software, the business district is also "where the business gets done" and is bounded by startup and shutdown code and contains the features and functions for which customers use the software. These are the "back of the box" features that would appear in a marketing commercial or sales demo and the code that supports those features.
- **Historical district:** Many cities have historic places or were the setting for historic events. Tourists love the mystery and legacy of the past and that makes historical districts very popular. For software, history is determined by its legacy code and history of buggy functions and features. Like real history, legacy code is often poorly understood, and many assumptions are made when legacy code is included, modified, or used. Tours through this district are aimed at testing legacy code.
- **Tourist district:** Many cities have districts where only tourists go. Locals and those who live in the city avoid these congested thoroughfares. Software is similar in that novice users will be attracted to features and functions that an experienced user has no more use for.
- **Entertainment district:** When all the sites and historical places have been exhausted by a tourist (or have exhausted the tourist!), some mindless relaxation and entertainment is often needed to fill out the corners of a good vacation. Software, too, has such supportive features, and tours to test these features are associated with this district. These tours complement the tours through other districts and fill out the corners of a good test plan.
- **Hotel district:** Having a place for tourists to rest at night, recover from their busy day, or wait out lousy weather is a must for any destination city. As we shall see, software is actually quite busy when it is "at rest," and these tours seek to test those features.
- **Seedy district:** Seedy districts are those unsavory places that few guidebooks or visitors bureaus will document. They are full of people doing bad and illegal things and are better off left alone. Yet they attract a certain class of tourist anyway. Seedy tours are a must for testers because they find places of vulnerability that may be very unsavory to users should they remain in the product.

Tours of the Business District

Business districts for cities bustle during the morning and afternoon rush and over the lunch hour. They are also where work gets done. They contain banks, office buildings, and usually aren't interesting places for tourists to hang out.

This is not the case for the software tourist. The parts of the application that "get the business done" are the reasons that people buy and use software. They are the features that appear in marketing literature, and if you polled customers about why they use your software, these are the features they would quote.

Business District tours focus on these important features and guide testers through the paths of the software where these features are used.

The Guidebook Tour

Guidebooks for tourists often boil down the sights to see to a select (and manageable) few. They identify the best hotels, the best bargains, and the top attractions, without going into too much detail or overwhelming a tourist with too many options. The attractions in the guidebook have been visited by experts who tell the tourist exactly how to enjoy them and get the most out of a visit.

It is sheer speculation on my part, but I would imagine many tourists never wander beyond the confines created by the authors of these guidebooks. Cities must ensure that such attraction areas are clean, safe, and welcoming so that tourists will spend their money and return often. From a testing standpoint, hitting such hotspots is just as important, and that makes this tour a crucial part of every testing strategy. Like cities, we want users to enjoy their experience, and so the major features need to be usable, reliable, and work as advertised.

The analogous artifact for exploratory testing is the user manual, whether it is printed or implemented as online help (in which case, I often call this the *F1 tour* to denote the shortcut to most help systems). For this tour, we will follow the user manual's advice just like the wary traveler, by never deviating from its lead.

The guidebook tour is performed by reading the user manual and following its advice to the letter. When the manual describes features and how they are used, the tester heeds those directives. The goal is to try and execute each scenario described in the user manual as faithfully as possible. Many help systems describe features as opposed to scenarios, but almost all of them give very specific advice about which inputs to apply and how to navigate the user interface to execute the feature. Thus this tour tests not only the software's ability to deliver the functionality as described but also the accuracy of the user manual.

Variations of this tour would be the *Blogger's tour,* in which you follow third-party advice, and the *Pundit's tour,* where you create test cases that describe the complaints of unhappy reviewers. You'll find these sources of

information in online forums, beta communities, user group newsletters, or if your application is big and widely distributed like Microsoft Office, on bookstore shelves. Another useful variant is the *Competitor's tour,* where you follow any of the above artifacts' advice for competing systems.[5]

The guidebook and its variants test the software's ability to deliver its advertised functionality. It's a straightforward test, and you should be alert for deviations from the manual and report those as bugs. It may end up that the fix is to update the manual, but in any event you have done a service for your users. The guidebook tour forces you to string together features of the software much the same way as a user would string them together and forces those features to interact. Any bugs found during this tour tend to be important ones.

In Chapter 6, "Exploratory Testing in Practice," several examples of the Guidebook tour are given.

The Money Tour

Every location that covets tourists must have some good reasons for them to come. For Las Vegas, it's the casinos and the strip, for Amsterdam it's the coffee shops and red light district, for Egypt it's the pyramids. Take these landmarks away and the place is no longer an attraction, and tourists will take their money elsewhere.

Software is much the same: There has to be some reason for users to buy it. If you identify the features that draw users, that's where the money is. For exploratory testers finding the money features means following the money, literally. And money usually leads directly to the sales force.

Sales folk spend a great deal of time giving demos of applications. One might imagine that because they get paid based on fulfilling their sales quota, they would be very good at it and would include any interesting nuances of usage that make the product look its very best. They also excel at shortcuts to smooth out the demo and often come up with scenarios that sell the product but weren't part of any specific requirements or user story. The long and short of it is that salespeople are a fantastic source of information for the *Money tour.*

Testers performing the Money tour should sit in on sales demos, watch sales videos, and accompany salespeople on trips to talk to customers. To execute the tour, simply run through the demos yourself and look for problems. As the product code is modified for bug fixes and new features, it may be that the demo breaks and you've not only found a great bug, but you've saved your sales force from some pretty serious embarrassment

---

[5] Testing your own application using the "guidebook" for a competing system is a novel approach to this tour. It works out very well in situations where the competing product is a market leader and you are trying to supplant it with your own. In these cases, the users who migrate to your application may well be used to working in the manner described in those sources, and therefore, you'll explore your application much the same way as (hopefully) lots of transitioning users. Better that such a tour happen with you as the tourist than to let your users discover whether your software meets their needs all on their own.

(perhaps even salvaging a sale). I have found enough bugs this way to privately wonder whether there is a case to be made for testers sharing in sales commissions!

A powerful variation of this tour is the *Skeptical Customer tour,* in which you execute the Money tour but pretend there is a customer constantly stopping the demo and asking "what if?" "What if I wanted to do *this?"* they may ask, or, "How would I do *that?"* requiring you to go off script and include a new feature into the demo. This happens a lot in customer demos, especially the serious ones where a purchase is imminent and the customer is kicking the tires one last time. It's a powerful way to create test cases that will matter to end users.

Once again, sitting in on customer demos by the sales force and having a good relationship with individual salespeople will give you a distinct advantage when you use this tour and will allow you to maximize the effect of this variation.

Clearly, any bugs you find on this tour are very important ones as they are likely to be seen by real customers.

In Chapter 6, several examples of the Money tour are given.

The Landmark Tour

As a boy growing up in the fields, meadows, and woods of Kentucky, I learned to use a compass by watching my older brother, who seemed to spend more time in the woods than he did anywhere else. He taught me how to orient myself by using the compass to pinpoint landmarks that were in the general direction we wanted to go. The process was simple. Use the compass to locate a landmark (a tree, rock, cliff face, and so forth) in the direction you want to go, make your way to that landmark, and then locate the next landmark, and so on and so forth. As long as the landmarks were all in the same direction, you could get yourself through a patch of dense Kentucky woods.[6]

The *Landmark tour* for exploratory testers is similar in that we will choose landmarks and perform the same landmark hopping through the software that we would through a forest. At Microsoft, we chose our landmarks in advance by selecting key features identified during the Guidebook tour and the Money tour. Choose a set of landmarks, decide on an ordering for them, and then explore the application going from landmark to landmark until you've visited all of them in your list. Keep track of which landmarks you've used and create a landmark coverage map to track your progress.

Testers can create a great deal of variation in this tour by choosing first a few landmarks and executing the tour, and then increasing the number of landmarks and varying the order in which you visit them.

---

[6] We once found a moonshine still this way, but that's one of the hazards of exploring rural Kentucky. Bugs will surface at a much faster rate than stills!

In Visual Studio, the first group at Microsoft to use the tours in production, this is the most popular and useful tour, followed closely by the Intellectual tour, which is described next.

The Intellectual Tour

I was once on a walking tour of London in which the guide was a gentleman in his fifties who claimed at the outset to have lived in London all his life. A fellow tourist happened to be a scholar who was knowledgeable in English history and was constantly asking hard questions of the guide. He didn't mean to be a jerk, but he was curious, and that combined with his knowledge ended up being a dangerous combination…at least to the guide.

Whenever the guide would talk about some specific location on the tour, whether it was Oscar Wilde's former apartment in Chelsea, details of the great fire, or what life was like when horses were the primary mode of transportation, the scholar would second guess him or ask him some hard question that the guide struggled to answer. The poor guide had never worked so hard on any past tour. Every time he opened his mouth, he knew he was going to be challenged, and he knew he had to be on his toes. He wasn't up to the task and finally admitted that he had only actually lived in London for five years, and he had memorized the script of the tour. Until he met the intellectual, his ruse had worked.

What a fantastic bug! The scholar actually managed to *break* the guide! I was so impressed I bought both the guide and the intellectual a pint when the tour ended at a pub (a place, incidentally, where the hapless guide was infinitely more knowledgeable than the scholar).

When applied to exploratory testing, this tour takes on the approach of *asking the software hard questions.* How do we make the software work as hard as possible? Which features will stretch it to its limits? What inputs and data will cause it to perform the most processing? Which inputs might fool its error-checking routines? Which inputs and internal data will stress its capability to produce any specific output?

Obviously, such questions will vary widely depending on the application under test. For folks who test word processors, this tour would direct them to create the most complicated documents possible, ones full of graphics, tables, multiple columns, footnotes, and so forth. For folks testing online purchasing systems, try to invent the hardest order possible. Can we order 200 items? Can we place multiple items on backorder? Can we keep changing our mind about the credit card we want to use? Can we make mistakes on *every* field in the data entry forms? This tour is going to be different for every application, but the idea is the same: Ask your software hard questions. Just as the intellectual did with the London guide, you are likely to find gaps in its logic and capabilities in the exact same manner.

A variation of this tour is the *Arrogant American tour* that celebrates a stereotype of my countrymen when we travel abroad. Instead of asking hard questions, we ask silly questions otherwise intended simply to annoy

or impede the tour and draw attention to ourselves. We intentionally present obstacles to see how the software reacts. Instead of the most complicated word processing document, we make the most colorful, invert every other page, print only prime number pages, or place something in a location that makes little sense. On a shopping site, we'll seek out the most expensive items only to return them immediately. It doesn't have to make sense…we do it because we *can.* It isn't unheard of for your users to do the same.

This tour and its variants will find any number of types of bugs from high priority to simply stupid. It's up to the exploratory testers to rein themselves in. Try to separate the truly outrageous hard questions (like asking a London guide whether he founded the city on the north side of the river Thames or the south side; it's a hard question, but it doesn't have much purpose) from questions that really make the software work. Try to create *realistically* complicated documents, orders, or other data so that it is easier to argue that bugs you find really matter to the user and should be fixed.

The Intellectual tour is used in Chapter 6 by Bola Agbonile to test Windows Media Player.

The FedEx Tour

FedEx is an icon in the package-delivery world. They pick up packages, move them around their various distribution centers, and send them to their final destination. For this tour[7], instead of packages moving around the planet through the FedEx system, think of data moving through the software. The data starts its life as input and gets stored internally in variables and data structures where it is often manipulated, modified, and used in computation. Finally, much of this data is finally "delivered" as output to some user or destination.

During this tour, a tester must concentrate on this data. Try to identify inputs that are stored and "follow" them around the software. For example, when an address is entered into a shopping site, where does it gets displayed? What features consume it? If it is used as a billing address, make sure you exercise that feature. If it is used as a shipping address, make sure you use that feature. If can be updated, update it. Does it ever get printed or purged or processed? Try to find every feature that touches the data so that, just as FedEx handles their packages, you are involved in every stage of the data's life cycle.

David Gorena Elizondo applies the *FedEx tour* to Visual Studio in Chapter 6.

The After-Hours Tour

Even business has to stop at some point for workers to commute to their homes or head for the after-hours gathering spots. This is a crowded time

[7] This tour was first proposed by Tracy Monteith of Microsoft.

for cities, and many tourists choose to stay away from the business districts after hours.

But not the software tester! After hours, when the money features are no longer needed, many software applications remain at work. They perform maintenance tasks, archive data, and back up files. Sometimes applications do these things automatically, and sometimes a tester can force them. This is the tour that reminds us to do so.

A variation of this tour is the *Morning-Commute tour*, whose purpose it is to test startup procedures and scripts. Had this tour been applied on the Zune, we may well have avoided the infinite loop that bricked first-generation Zune devices on December 31, 2008.

The After-Hours Zune Bug

December 31 was the 366th day in the leap year of 2008. On that day, Microsoft's first-generation Zune froze and never recovered. The bug was an off-by-one error in a loop that only handled a year with 365 days. The result is that the loop never ended and the Zune froze, which is exactly what an infinite loop will cause software to do. The code looks like this:

```
year = ORIGINYEAR; /* = 1980 */
while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}
```

This code takes the clock information and computes the year and counts down from either 365 or 366 until it can determine the month and day. The problem is that 366 is too large a number to ever break out of the `while` loop (meaning that the loop never ends and the Zune goes off into never-never land). Because this script is in the startup code, once you turn the Zune on, it is caput. The solution requires that you need a new clock value on January 1 in order to reset the device: Therefore, the fix is to wait until the new year and pull the battery!

The Garbage Collector's Tour

Those who collect curbside garbage often know neighborhoods better than even residents and police because they go street by street, house by house, and become familiar with every bump in the road. They crisscross neighborhoods in a methodical manner, stopping at each house for a few moments before moving on. However, because they are in a hurry, they don't stay in one place very long.

For software, this is like a methodical spot check. We can decide to spot check the interface where we go screen by screen, dialog by dialog (favoring, like the garbage collector, the shortest route), and not stopping to test in detail, but checking the obvious things (perhaps like the Supermodel tour). We could also use this tour to go feature by feature, module by module, or any other landmark that makes sense for our specific application.

The *Garbage Collector's tour* is performed by choosing a goal (for example, all menu items, all error messages, all dialog boxes), and then visiting each one in the list by the shortest path possible. In Chapter 6, Bola Agbonile applies this tour to Windows Media Player, and Geoff Staneff applies it to Visual Studio.

Tours Through the Historical District

Historical districts within cities are areas that contain old buildings and places of historical note. In cities like Boston, they are distributed around the city and connected by marked walking trails. In Cologne, Germany, a small contiguous section of the city is called the "old town" to mark where the city stood before its modern expansion.

In software, historical district can be as loosely connected as in Boston or as contained as they are in Cologne. The historical district represents areas of legacy code, features that debuted in earlier versions, and bug fixes. The latter is particularly important because when it comes to bugs, history does indeed repeat itself, and it is important to retest previously buggy sections of code. Tours through the historical district are designed to test legacy functionality and verify bug fixes.

The Bad-Neighborhood Tour

Every city worth visiting has bad neighborhoods and areas that a tourist is well advised to avoid. Software also has bad neighborhoods—those sections of the code populated by bugs. The difference between real tourists and exploratory testers, however, is that the former try to avoid bad neighborhoods, and the latter are well advised to spend as much time in them as possible.

Clearly, we do not know in advance which features are likely to represent bad neighborhoods. But as bugs are found and reported, we can connect certain features with bug counts and can track where bugs are

occurring on our product. Because bugs tend to congregate,[8] revisiting buggy sections of the product is a tour worth taking. Indeed, once a buggy section of code is identified, it is recommended to take a Garbage Collector's tour through nearby features to verify that the fixes didn't introduce any new bugs.

The Museum Tour

Museums that display antiquities are a favorite of tourists. The Smithsonian and various museums of natural history draw many thousands of visitors on a daily basis. Antiquities within a code base deserve the same kind of attention from testers. In this case, software's antiquities are legacy code.

Untouched legacy code is easy to identify by a quick perusal of the date/time stamps in the code repository or on project binary and assembly files. Many source repositories also maintain a modification record, so testers can do a little research to see what older code may contain some recent modifications.

Older code files that undergo revision or that are put into a new environment tend to be failure prone. With the original developers long gone and documentation often poor, legacy code is hard to modify, hard to review, and evades the unit testing net of developers (who usually write such tests only for new code). During this tour, testers should identify older code and executable artifacts and ensure they receive a fair share of testing attention.

The Prior Version Tour

Whenever a product is constructed as an update from a prior version, it is good to run all the scenarios and test cases that applied to the prior version. This will validate that functionality that users are used to is still supported in a useful and usable way in the new product. In the event that the newer version reimplements or removes some functionality, the tester should choose the inputs that represent the new way of doing things as defined by the latest version. Any tours that are no longer possible in the new version should be scrutinized to ensure that no necessary functionality was lost.

Tours Through the Entertainment District

On every vacation, tourists will need a break from their busy schedule of fighting crowds and seeing the sites. Visiting the entertainment district, taking in a show, or having a long quiet dinner out is a common way of doing this. Entertainment districts aren't about seeing sites, they fill in the gaps of a vacation and give a local flavor to relaxation.

---

[8] Bugs congregate in features for any number of reasons. Because developers tend to be assigned to a project based on feature ownership, a single feature will have a proportion of bugs based on the skill of the individual developer. Bugs also congregate around complexity; so features that are harder to code may end up with more bugs. The idea here is that after a feature is shown to be buggy, there are very likely more bugs to be found in that feature if only you keep looking for them.

Most software has features that serve these purposes. For example, the business district for a word processor is the set of features to construct the document, write text, and insert graphics, tables, and artwork. The entertainment district, on the other hand, is the filler features for laying out pages, formatting text, and modifying backgrounds and templates. In other words, the work is in making the document and the "fun" part is making it look nice and represents an intellectual break from the actual work.

Tours through the entertainment district will visit supporting rather than mainline features and ensure that the two are intertwined in useful and meaningful ways.

The Supporting Actor Tour

I'm glad I started this chapter using London as my analogy because London is full of interesting sites and really cool buildings. On one of the many guided walking tours I've taken through nearly all parts of the city, I couldn't help but to be more attracted to the buildings that the guide was *not* pointing out than the ones he was telling us about. As he described a famous church that had historical significance, I found myself drawn to a short row house of buildings with rounded doors barely more than 5 feet high. It was like the hobbit portion of the city. On another stop, he was telling the story of pelicans that had been granted tenure in one of the city's parks. I found the pelican uninteresting, but a small island in the pond had a willow tree with a root structure that looked like dragon's teeth. I was enthralled.

Whenever salespeople demo a product or marketing touts some feature of our application, users are liable to be tempted by features nearby those in the spotlight. The *Supporting Actor tour* encourages testers to focus on those features that share the screen with the features we expect most users to exercise. Simply their proximity to the main event increases their visibility, and we must not make the mistake of giving those features less attention than they deserve.

Examples include that little link for similar products that most people skip in favor of clicking on the product they searched for. If a menu of items is presented and the second item is the most popular, choose the third. If the purchasing scenarios are the bread and butter, then select the product review feature. Wherever the other testers are looking, turn your attention a few degrees left or right and make sure that the supporting actor gets the attention it deserves.

In Chapter 6, Nicole Haugen shows how she used the Supporting Actor tour on the Dynamics AX client software.

The Back Alley Tour

In many peoples' eye, a good tour is one in which you visit popular places. The opposite of these tours would be one in which you visited places no one else was likely to go. A tour of public toilets comes to mind, or a swing through the industrial section of town. There are also so-called "behind the

scenes" tours at places like Disney World or film studios where one can see how things work and go where tourists ordinarily don't tread. In exploratory testing terms, these are the least likely features to be used and the ones that are the least attractive to users.[9]

If your organization tracks feature usage, this tour will direct you to test the ones at the bottom of the list. If your organization tracks code coverage, this tour implores you to find ways to test the code yet to be covered.

An interesting variation on this theme is the *Mixed-Destination tour.* Try visiting a combination of the most popular features with the least popular. You can think of this as the Landmark tour with both large and small landmarks intermixed. It may just be that you find features that interact in ways you didn't expect because developers didn't anticipate them being mixed together in a single scenario.

Feature Interaction

It's a frustrating fact of testing life that you can test a feature to death and not find bugs only to then see it fail when it interacts with another feature. In reality, one would have to test every feature of an application with every other feature in pairs, triplets, and so on to determine whether they interact in ways that will make the software fail. Clearly, such an exhaustive strategy is impossible, and for the most part it is not necessary. Instead, there are ways to determine if two features need to be tested together.

I like to frame this problem as a series of questions. Simply select two candidate features and ask yourself the following:

- **The input question:** Is there an input that gets processed by both features in question?
- **The output question:** Do the features operate on the same portion of the visible UI? Do they generate or update the same output?
- **The data question:** Do the features operate on shared internal data? Do they use or modify the same internally stored information?

If the answer to any of these questions is "yes," then the features interact and need to be tested together.

In Chapter 6, Nicole Haugen, David Gorena Elizondo, and Geoff Staneff all use the Back Alley tour for a variety of testing tasks.

---

[9] It would be fair to ask whether we should test these features at all, but I feel that it is important that we do so. If the feature has made it into the product, it is important to someone somewhere. At companies like Microsoft and Google, the user base is so large that even the less-popular features can be used millions of times in a single day. In such cases, there really is no such thing as an unimportant feature. However, it is wise to proportion a testing budget in accordance with usage frequency as much as it is possible to do so.

The All-Nighter Tour

Also known as the *Clubbing tour,* this one is for those folks who stay out late and hit the nightspots. The key here is *all night.* The tour must never stop; there is always one more club and one last drink. Such tours, some believe, are tests of the constitution. Can you last? Can you survive the all-nighter?

For exploratory testers, the question is the same: Can the app last? How long can it run and process data before it just collapses? This is a real challenge for software. Because of the buildup of data in memory and the constant reading and writing (and rereading and rewriting) of variable values, bad things can happen if given time to do so. Memory leaks, data corruption, race conditions…there are many reasons to give time to your testing. And because closing and opening an application resets the clock and clears out memory, the logic behind this tour is to *never close the app.* This also extends to using features continuously and keeping files open continuously.

Exploratory testers on the *All-Nighter tour* will keep their application running without closing it. They will open files and not close them. Often, they don't even bother saving them so as to avoid any potential resetting effect that might occur at save time. They connect to remote resources and never disconnect. And while all these resources are in constant use, they may even run tests using other tours to keep the software working and moving data around. If they do this long enough, they may find bugs that other testers will not find because the software is denied that clean reset that occurs when it is restarted.

Many groups use dedicated machines that never get turned off and run automation in a loop. It is even more important to do this for mobile devices that often do stay on for days at a time as a normal course of usage. Of course, if there are different stages of reset, such as a sleep mode or hibernation mode, these can be used at varying rates as long as the software itself retains its state information.

Tours Through the Tourist District

Every city that focuses on tourism has a section of town where tourists congregate. It's full of souvenir shops, restaurants, and other things to maximize spending and ensure the profits of the local merchants. There are collectibles for sale, services to be bought, and pampering to be had.

Tours through the tourist district take several flavors. There are short trips to buy souvenirs, which are analogous to brief, special-purpose test cases. There are longer trips to visit a checklist of destinations. These tours are not about making the software work, they are about visiting its functionality quickly…just to say you've been there.

The Collector's Tour

My parents possess a map of the United States with every state shaded a different color. Those states all started out white, but as they visited each state on vacation, that state was then colored in on the map. It was their goal to visit all 50 states, and they went out of their way to add to their collection. One might say they were collecting states.

Sometimes a tour involves freebies that can be collected. Maybe it's a wine tasting or a fair with booths for children to work on some craft. Whatever it may be, there is always someone who wants to do everything, collect everything. Perhaps it's the guy who has to take a picture of every single statue in the museum, a lady who wants her kid to meet every overstuffed character at Disney World, or the guy at the supermarket who must consume every free sample on offer. Well, this is just the kind of greed that will come in handy to the exploratory tester.

For exploratory testers also collect things and strive for completeness. The *Collector's tour* suggests collecting *outputs* from the software; and the more one collects, the better. The idea behind this tour is to go everywhere you can and document (as my parents did on their state map) all the outputs you see. Make sure you see the software generate every output it can generate. For a word processor, you would make sure it can print, spell check, format text, and so on. You may create a document with every possible structure, table, chart, or graphic. For an online shopping site, you need to see a purchase from every department, credit card transactions that succeed and ones that fail, and so on and so forth. Every possible outcome needs to be pursued until you can claim that you have been everywhere, seen everything, and completed your collection.

This is such a large tour that it is often good to take it as a group activity. Divvy up the features among members of the group or assign certain outputs to specific individuals for collection. And when a new version of the application is ready for testing, one needs to throw away all the outputs for the features that have changed and restart the collection.

The Collector's tour is demonstrated by Nicole Haugen in Chapter 6.

The Lonely Businessman Tour

I have a friend (whom I won't name given the rather derogatory title of this tour) who travels a great deal on business. He has visited many of the world's great cities, but mostly sees the airport, the hotel, and the office. To remedy this situation, he has adopted the strategy of booking a hotel as far away from the office he's visiting as possible. He then walks, bikes, or takes a taxi to the office, forcing him to see some of the sites and get a flavor of the city.

Exploratory testers can perform a variety of this tour that can be very effective. The idea is to visit (and, of course, test) the feature that is furthest away from the application's starting point as possible. Which feature takes the most clicks to get to? Select that one, click your way to it, and test it. Which feature requires the most screens to be navigated before it does anything useful? Select it and test it. The idea is to travel as far through the application as possible before reaching your destination. Choose long paths over short paths. Choose the page that is the buried deepest within the application as your target.

You may even decide to execute the Garbage Collector's tour both on the way to such a destination and once you get where you are going.

The Supermodel Tour

For this tour, I want you to think superficially. Whatever you do, don't go beyond skin deep. This tour is not about function or substance; it's about looks and first impressions. Think of this as the cool tour that all the beautiful people take; not because the tour is meaningful or a great learning experience. On the contrary, you take this tour just to be *seen.*

Get the idea? During the *Supermodel tour,* the focus is not on functionality or real interaction. It's only on the interface. Take the tour and watch the interface elements. Do they look good? Do they render properly, and is the performance good? As you make changes, does the GUI refresh properly? Does it do so correctly or are there unsightly artifacts left on the screen? If the software is using color in a way to convey some meaning, is this done consistently? Are the GUI panels internally consistent with buttons and controls where you would expect them to be? Does the interface violate any conventions or standards?

Software that passes this test may still be buggy in many other ways, but just like the supermodel…it is going to look really good standing at your side.

The Supermodel tour is used extensively in Chapter 6. Along with the Landmark tour and the Intellectual's tour, it was used on every pilot project at Microsoft.

The TOGOF Tour

This tour is a play on the acronym for Buy One Get One Free—BOGOF—that is popular among shoppers. The term is more common in the United Kingdom than the United States, but in either case it is not just for groceries and cheap shoes anymore. The idea isn't for the exploratory tester to buy anything, but instead to *Test One Get One Free.*

The *TOGOF tour* is a simple tour designed only to test for multiple copies of the same application running simultaneously. Start the tour by running your application, then starting another copy, and then another. Now put them through their paces by using features that cause each application to do something in memory and something on the disk. Try using all the different copies to open the same file or have them all transmit data over the network simultaneously. Perhaps they will stumble over each other in some way or do something incorrect when they all try to read from and write to the same file.

Why is it a TOGOF? Well, if you find a bug in one copy, you've found a bug in all of them! David Gorena Elizondo demonstrates how he applied this tour to Visual Studio in Chapter 6.

The Scottish Pub Tour

My friend Adam Shostack (the author of *The New School of Information Security*) was visiting Amsterdam when he had a chance meeting with a group of Scottish tourists. (The kilts betrayed their nationality as much as their accents.) They were members of a pub-crawling troupe with international tastes. He joined them on a pub tour of the city that he readily

concedes consisted of venues he would never have found without their guidance. The pubs ranged from small, seedy joints to community gathering places buried in neighborhoods more than a little off the beaten path.

How many such places exist in my own town, I wonder? There are many places that you can find only by word of mouth and meeting the right guide.

This tour applies specifically to large and complicated applications. Microsoft Office products fit this category. So do sites such as eBay, Amazon, and MSDN. There are places in those applications that you have to know about to find.

This isn't to say they receive hardly any usage, they are just hard to find. Adam tells stories of many of the pubs on his Scottish tour fairly heaving with people. The trick is in finding them.

But testers can't depend on chance meetings at hotels with kilt-wearing guides. We have to meet the guides where they are. This means finding and talking to user groups, reading industry blogs, and spending a great deal of time touring the depths of your application.

Tours Through the Hotel District

The hotel is a place of sanctuary for the tourist. It is a place to get away from the hustle and bustle of the vacation hotspots for a little rest and relaxation. It is also a place for the software tester to get away from the primary functionality and popular features and test some of the secondary and supporting functions that are often ignored or under-represented in a test plan.

The Rained-Out Tour

Once again, my selection of London as my tourist base pays off because sometimes even the best tours get rained out. If you've taken a pub tour in London between the autumn and spring months, it can be a wet, rainy affair, and you may well find yourself tempted to cut the tour short at the second stop. For the tourist, I do not recommend this tactic. You are already wet, and that's just going to ensure that the next pub feels even better than the last once you manage to actually get there. But for the exploratory tester, I highly recommend use of that cancel button.

The idea behind the *Rained-Out tour* is to start operations and stop them. We may enter information to search for flights on a travel website only to cancel them after the search begins. We may print a document only to cancel it before the document is complete. We will do the same thing for any feature that provides a cancel option or that takes longer than a few seconds to complete.

Exploratory testers must seek out the time-consuming operations that their application possesses to use this attack to its fullest. Search capabilities are the obvious example, and using terms that make for a longer search is a tactic that will make this tour a little easier. Also, every time a cancel button appears, click it. If there is no cancel button, try the Esc key or even the back

button for apps that run in a browser. There's always Shift-F4 or closing the X button to close the application completely. And try this, too: Start an operation, and then start it again without stopping the first.

The failures you will see on this tour are mostly related to the inability of the application to clean up after itself. There are often files left open, data left clogging up internal variables, and a state of the system that is no longer tenable for the software to do much else. So after you hit cancel (etc.), take some time to poke around the application to make sure it is still working properly. At the very least, you want to make sure that whatever action you canceled should be able to be exercised again and complete successfully. After all, one would expect a user to occasionally cancel something and then try again.

The Rained-Out tour is used extensively in Chapter 6.

The Couch Potato Tour

There's always one person on a group tour who just doesn't participate. He stands in the back with his arms folded. He's bored, unenergetic, and makes one wonder exactly why he bothered paying for the tour in the first place. However, on such tours, the guide is often prompted to work harder to try to draw the couch potato in and help him enjoy the tour.

From the tourist's perspective, it sounds like, and probably is, a waste of time. But it's exactly the opposite for software testers. Couch potatoes can make very effective testers! The reason is simple, even if it is not intuitive: Just because the tester isn't doing much does not mean that the software follows suit. Like the diligent tour guide, it's often the case that nonactivity forces software to work very hard because it is busy executing the "else" clauses in IF-THEN-ELSE conditions and figuring out what to do when the user leaves data fields blank. A great deal of "default logic" executes when a user declines to take the initiative.

A *Coach Potato tour* means doing as little actual work as possible. This means accepting all default values (values prepopulated by the application), leaving input fields blank, filling in as little form data as possible, never clicking on an advertisement, paging through screens without clicking any buttons or entering any data, and so forth. If there is any choice to go one way in the application or another, the coach potato always takes the path of least resistance.

As lazy as this sounds, and granted the tester does little real interaction, this does not mean the software is not working. Software must process default values, and it must run the code that handles blank input. As my father used to say (mostly during basketball games…I grew up in Kentucky, where the sport of hoops ruled), "That spot on the couch doesn't keep itself warm." And the same can be said of those default values and error-checking code: It doesn't execute itself, and missing default cases are far too common and very embarrassing in released products.

Tours Through the Seedy District

Much of the material presented in Chapter 3, "Exploratory Testing in the Small," would fit into this district if you could blend it into a tour. Inputs meant to break the software and do general harm are seedy in nature and fit the general purpose of these tours.

The Saboteur

This is the *Saboteur tour,* and during it we will attempt to undermine the application at every possible opportunity. We will ask the application to read from the disk (by opening a file or using some disk resource), but then sabotage its attempt to do so by rigging the file operations to fail (perhaps by corrupting the file in question). We will also ask it to do some memory-intensive operation when the application is either on a machine with too little memory or when other applications are operating in the background and consuming most of the memory resources.

This tour is simple to conceptualize:

- Force the software to take some action.
- Understand the resources it requires to successfully complete that action.
- Remove or restrict those resources in varying degrees.

During this tour, a tester will find that there are many ways to rig environments by adding or deleting files, changing file permissions, unplugging network cables, running other applications in the background, deploying the application under test on a machine that has known problems, and so forth. We might also employ the concept of *fault injection*[10] to artificially create errant environmental conditions.

The saboteur is very popular at Microsoft, using fault-injection tools and simpler mechanisms that are illustrated in Chapter 6. In particular, Shawn Brown has used this tour extensively on Windows Mobile.

The Antisocial Tour

Pub tours are one of my personal passions, and I enjoy them on my own or as part of a guided walk. I recall a specific tour in which a husband clearly had coerced his wife to accompany him. She wanted no part of the tour. When we went into a pub, she stayed outside. When it was time to leave the pub, she would walk in and order a drink. When we admired scenery or some landmark, she suddenly found a common squirrel fascinating. Everything the tour took in, she made it a point to do the opposite. She was so successful that at the end of the tour another tourist handed her husband his business card; that other tourist was a divorce attorney.

---

[10] The concept of runtime fault injection is covered in detail in *How to Break Software* from page 81 to 120, and again in Appendixes A and B.

An attorney with a sense of humor notwithstanding, I found her antisocial behavior absolutely inspiring from a test point of view. Exploratory testers are often trying specifically to break things; and being nice, kind, and following the crowd is seldom the best way to accomplish that goal. As a tester, it pays to be antisocial. So if a developer hands you the business card of a divorce attorney, you may consider it the highest of compliments.

The *Antisocial tour* requires entering either the least likely inputs and/or known bad inputs. If a real user would do *a,* then a tester on the Antisocial tour should never do *a* and instead find a much less meaningful input.

There are three specific ways to accomplish such antisocial behavior, which I organize here into subtours:

- The *Opposite tour* is performed by entering the least likely input every chance you get. Testers taking this tour select inputs that are out of context, just plain stupid, or totally nonsensical. How many of that item do you want in your shopping cart? 14,963. How many pages to print? –12. The idea is to apply the input that is the least likely input you can come up with for a specific input field. By doing so, you are testing the application's error-handling capability. If it helps, think of it as testing the application's patience!
- Illegal inputs are handled during the *Crime Spree tour,* and the idea here is to apply inputs that should *not* occur. You wouldn't expect a tourist to steal a pint on a pub tour, but on a crime spree that's the analogous behavior. You'll do the things that are not just antisocial, but downright illegal.

  Breaking the law as a tourist will land you in trouble or in jail; breaking the law as a tester will result in lots of error messages. Expect the Crime Spree tour inputs to invoke error messages, and if they do not, you may very well have a bug on your hands. Enter inputs that are the wrong type, the wrong format, too long, too short, and so forth. Think in terms of "what constraints are associated with this input," and then break those constraints. If the application wants a positive number, give it a negative number. If it wants an integer, give it a character. Keeping a tally of the error messages will be important for the next few chapters, where these tours will actually be used.
- Another aspect of antisocial behavior is embodied in the *Wrong Turn Tour*, which directs the tester to do things in the wrong order. Take a bunch of legal actions and mix them around so that the sequence is illegal. Try checking out before putting anything in your shopping cart. Try returning an item you didn't purchase. Try to change the delivery options before you complete your purchase.

The Obsessive-Compulsive Tour

I'm not quite sure such a tour in real life would be all that popular, and only its name made me put it in the seedy district. I can't actually imagine that a walking tour in which you can't step on any sidewalk cracks would gain

many customers beyond the kindergarten demographic. Nor would a bus that drives only on a single street—just because the driver is desperate not to miss anything—find many riders. But being obsessive in testing can pay off.

OCD testers will enter the same input over and over. They will perform the same action over and over. They will repeat, redo, copy, paste, borrow, and then do all that some more. Mostly, the name of the game is repetition. Order an item on a shopping site and then order it again to check if a multiple purchase discount applies. Enter some data on a screen, then return immediately to enter it again. These are actions developers often don't program error cases for. They can wreak significant havoc.

Developers are often thinking about a user doing things in a specific order and using the software with purpose. But users make mistakes and have to backtrack, and they often don't understand what specific path the developer had in mind for them, and they take their own. This can cause a usage scheme carefully laid by developers to fall by the wayside quickly. It's better to find this out in testing than after release, which makes this an important tour for testers to complete.

Putting the Tours to Use

Tours give a structure to testing and help guide testers to more interesting and relevant scenarios than they would ordinarily come up with using only freestyle testing. By giving a goal to testers, the tours help guide them through interesting usage paths that tend to be more sophisticated than traditional feature-oriented testing where an individual tester will try to test a single feature in isolation.

Features are a common pivot for testers. A test manager may divide an application into features and distribute those features across her testers. Testing features in isolation will miss many important bugs that users, who mostly use features in combination and in sequence, will encounter. The tours are an important tool for testers to discover interesting ways of combining features and functionality in single test cases and in sets of test cases: The more interaction, the more thorough the testing.

Repeatability is another aspect of using the tours that I have noticed in practice. If two testers are told "go test this app," it is very likely that the two of them will test it in completely different ways. If the same two testers are told "go run this tour," they will tend to do very similar things and likely even identify the same bugs. The built-in strategy and goal of the tour makes them more repeatable and transferable among testers. It also helps a great deal in educating testers about what constitutes good test design as the tours guide testers through the question of *what should I test.*

Some tours are very likely to find many more problems than other tours, and if careful records are kept, the tours can be rank-ordered late in

the cycle when every single test case counts. Testers should track which ones find the most bugs, take the least amount of time to execute, cover the most code/UI/features, and so forth. This is a side benefit of using actual strategy to organize testing; the tours provide specific categories of tests that we can rule in or out as better or worse in some situations. That way, over time, we can refine our methods and techniques and improve our testing from project to project. You'll only understand which tours are working if you pay attention and track your progress in finding bugs, discovering usability or performance problems, or simply verifying functionality in a cost- and time-effective manner.

The tours are an excellent way of distributing testing needs among members of a test team. As you gain comfort with the tours, patterns will emerge about which tours find certain classes of bugs and which are compatible with a specific feature. It is important that such knowledge gets documented and becomes part of the testing culture within your organization. Thus the tours not only become a way to test, they are also a way to organize testing and improve the spread and retention of testing knowledge across your team.

In many ways, that is what testing is all about: doing your best this time and making sure that next time you do even better. The tourist metaphor helps us organize according to this purpose.

Conclusion

Tours represent a mechanism to both organize a tester's thinking about how to approach exploring an application and in organizing actual testing. A list of tours can be used as a "did you think about this" checklist and also help a tester match application features to test techniques that will properly exercise them.

Furthermore, the tours help testers make the myriad decisions about which paths to choose, inputs to apply, or parameters to select. Certain decisions are simply more in the spirit of the selected tour than others and thus naturally emerge as "better" choices. This is testing guidance in its purest form.

Finally, at Microsoft, the tours are seen as a mechanism for gathering tribal knowledge, in that some tours will eventually establish a track record of success. In Visual Studio, the Landmark tour and the Intellectual tour have become part of the everyday language of our test community. Testers know what those tours are, how to apply them, and have a general idea of how much coverage and what types of bugs will ensue. It makes discussing testing easier and becomes part of the way we train new testers on our teams.

Exercises

1. Write your own tour! Using the tours discussed in this chapter as a guide, create your own tour. Your tour should have a name, a tie-in with the tourist metaphor, and you should use it on some software system and describe how the tour helps you test.
2. Find at least two tours that give similar testing advice. In other words, the tours might end up discovering the same bug or covering the same features in an application. Give an example testing scenario where the two tours cause a tester to do roughly the same test.
3. Using your favorite web application (eBay, Amazon, MySpace, and so on), write a test case for that application using any five of the tours discussed in this chapter as a guide.

CHAPTER 6
Exploratory Testing in Practice

"Not all who wander are lost."

—J. R. R. Tolkien

The Touring Test

A testing technique is nothing until it leaves home and makes a name for itself in the real world. It is one thing to talk of adventure and travel, and quite another to live it. This chapter was written by the first testers to apply the tourism metaphor purposefully on real projects and under real ship pressure.

The in-the-small and especially in-the-large techniques were born within the Developer Division of Microsoft and saw their first use within our teams in Redmond, Washington, and our India Development Center at the hands of David Gorena Elizondo and Anutthara Bharadwaj, respectively. They saw their first public unveiling at EuroSTAR 2008 in Den Haag, Netherlands,[1] in November 2008. Since then, I am personally aware of some dozens of groups outside Microsoft who have made them work.

Later that same month, two efforts were launched inside Microsoft to take the tours "on tour." A formal effort within Visual Studio was started that is still ongoing at the time of this writing. At the same time, a broader grassroots effort began companywide. I began it by sending an email to the Test Managers and Test Architects in the company, those testers of the highest level and broadest reach within Microsoft, asking to be referred to talented testers, regardless of their experience. I specifically asked for promising manual testing talent.

I was flooded with responses and whittled down the pool with personal interviews, frankly by choosing those who seemed most enthusiastic. I like working with passionate people! Then we began training, and each tester read and edited this text. After that, we began touring.

---

[1] You can find information about EuroSTAR at http://qualtechconferences.arobis.com/content.asp?id=91.

Many product lines were involved, from games to enterprise, mobile to cloud, operating system to web services. I've selected five of the most informative to appear here. The remainder are still ongoing, so don't be surprised if there are follow-ups on Microsoft blogs or elsewhere.

As you will see in this discussion, many of the tours survived being put into practice intact and were applied exactly how they were documented in Chapter 4, "Exploratory Testing in the Large." However, many variations were crafted on-the-fly, and in some cases new tours were created from scratch. Not only is this acceptable, it's desirable. I expect any team to find tours that work for them and tours that don't. That's why this chapter is here: to show how the tours work in practice.

The experience reports that follow are from the following Microsoft testers and appear with their permission:

- Nicole Haugen, Test Lead, Dynamics AX Client Product Team
- David Gorena Elizondo, SDET, Visual Studio Team Test
- Shawn Brown, Senior Test Lead, Windows Mobile
- Bola Agbonile, Software Development Engineer in Test, Windows
- Geoff Staneff, SDET, Visual Studio Team System

Touring the Dynamics AX Client

By Nicole Haugen

My team is responsible for testing Dynamics AX's client. Dynamics AX is an enterprise resource planning, or ERP, solution that was implemented more than 20 years ago in native C++ and was acquired when Microsoft purchased Navision. As the client team, we are considered to be a "foundation" team that is responsible for providing forms, controls, and shell functionality that the rest of the application is built on. Prior to this, my team was primarily testing public APIs, so Dynamics AX was a mind shift to testing through a GUI. When we made this transition, we learned several things:

- Many of the bugs that we found were not being caught by the test cases that we had identified in our test designs.
- Testing through the GUI introduced a seemingly infinite number of scenarios and complex user interactions that were not easily captured using automated tests.
- Whether a test is automated or manual, it is a regression test that must be maintained. My team has thousands of tests, so we must constantly consider the return on investment associated with adding a new test case to our regressions.
- Dynamics AX is a massive application and there was a lot about it we did not know, let alone how it should be tested.

Exploratory testing helped us to address all of the proceding issues. As a result, we have incorporated it into our process in the following ways:

- Before each feature is checked in, a tester performs exploratory testing on the code; this is to find important bugs fast and preferably, before they are ever checked in. We also follow this same practice for check-ins related to fixes of critical or high-risk bugs.
- Exploratory testing is used to help with the development of test cases while writing our test designs. It helps us to discover new scenarios that may have been missed in requirements.
- During our manual test pass, we use the test scripts as a jumping-off point to inject exploratory testing, as described in Chapter 5, "Hybrid Exploratory Testing Techniques." It has been my personal experience that the manual tests as they are written rarely detect new issues; however, with even the slightest detour from the test, many bugs are found.
- During bug bashes, we perform exploratory testing, which has helped lead us to investigate other areas outside of the current feature area to discover related issues.

Useful Tours for Exploration

The concept of tours makes exploratory testing much more concrete, teachable, and repeatable for us. This section describes a few tours that have been particularly useful for finding bugs in Dynamics AX.

Taxicab Tour

When traveling by mass public transportation, there is always risk that travelers may board the wrong route or get off at the wrong stop. Another drawback is that it is often impossible to take a direct route to a desired location. In the rare case that it is, the exact same route is usually taken over and over, with no variety to offer those that have visited the destination more than once. One surefire alternative is to travel by taxicab. While the fare of a taxicab costs more, the old adage "you get what you pay for" definitely applies. In a city such as London, where there are more than 25,000 streets, cab drivers must take rigorous exams to ensure that they know every route possible to get from point A to point B within the city. You can bet your bottom dollar (or should I say pound) that cab drivers know which route has the shortest distance, which offers the shortest amount of travel time, and even which is the most scenic. Furthermore, each and every time, passengers will consistently arrive at their specified location.

This same type of tour is also applicable to testing software applications. To reach a desired screen, dialog, or some other piece of functionality, there are often myriad routes that the user can take. As a result, testers have the same responsibility as taxicab drivers, in that they must educate themselves on every possible route to a specified location. Testers can then leverage this knowledge to verify that each route consistently delivers users to

their target destination. In some situations, the state of the target destination may be expected to vary depending on the route, which should also be verified. Notice that this tour is a derivative of the Obsessive-Compulsive tour, in that the ultimate goal is to repeat a specific action; however, rather than exercising the exact *same* path to the action over and over, the key difference is that this tour concentrates on exercising *different* paths.

Consider an example using Microsoft Office's Print window. To open this window, users have various options:

- They can send the Ctrl+P hotkey.
- They can select the Print menu item through the Office menu button.
- They can click the Print button on the toolbar of the Print Preview window.

No matter which of these routes the user chooses, the end result should be the same: The Print window opens.

Conversely, there is also the concept of the *Blockaded Taxicab tour.* The objective of this tour is to verify that a user is consistently blocked from a destination regardless of the route taken. There are many different reasons why a user may be prevented from accessing functionality within an application, whether it is because the user does not have sufficient permissions or it is to circumvent the application from entering into an invalid state. Regardless, it is important to test each and every route because it is surprising how many times a developer overlooks ones.

Although the preceding example involving the Print window is becoming somewhat contrived at this point, let's continue with it for the sake of simplicity. Suppose that a user should be prohibited from printing hard copies. Because we know that there are many different ways to access the Print window, it is important to verify that no matter how the user tries to access this window that the user is consistently prevented. This means that at a minimum, the Ctrl+P hotkey should result in a no-op and that the Print menu item and toolbar button should become disabled.

Multicultural Tour

One great aspect of London is the sheer diversity of people that it encompasses. Without leaving the city, let alone the country, a tourist can experience cultures from around the world. For example, a tourist might choose to visit London's Chinatown, where the delicious aroma of Chinese cuisine can be smelled and traditional Chinese scripture can be viewed throughout. Similarly, a tourist may opt for an Indian restaurant tucked within one of London's busy streets, where diners enjoy smoking tobacco out of hookahs. Tourists can choose to submerse themselves into many different cultures as an exciting and wonderful way to spend a vacation in London.

The *Multicultural tour* applies to testing because it is important that testers consider the implications of providing software that is localized to different countries around the world. It is essential that language, currency,

formatting of dates, types of calendars, and so forth be adapted appropriately to the end users' region. In addition, it is important that functionality continues to work as expected, regardless of the locale.

Although testing a product's localization can be very complex, here are a few basic ideas to get you started. Notice that you do not necessarily need to be fluent in a different language to perform these types of tests:

- A basic aspect of localization is that no text should be hard-coded (and thereby prohibiting it from being translated to the appropriate language). A great way to test this is simply to change the application's and operating system's language and verify that labels, exception messages, tooltips, menu items, window captions, and so on no longer appear in English. Also, it is likely that there are specific words that should *not* be translated, which should also be verified, such as words that are part of a brand name.
- Try launching the application under test in a right-to-left language, such as Arabic; verify that controls and windows behave correctly. With right-to-left languages, it is interesting to change the size of windows and make sure that they repaint correctly. Also, test controls, especially custom-implemented controls, and make sure that they still function as they did in left-to-right mode.

This list is obviously far from being comprehensive but at least gives an idea of general aspects of an application to verify without necessarily getting language-specific.

The Collector's Tour and Bugs as Souvenirs

This section is dedicated to the bugs that have been collected as souvenirs while traveling throughout Dynamics AX using the Taxicab and Multicultural tours. While these tours have helped to find many bugs, here are a few of my favorites.

A Bug Collected Using the Blockaded Taxicab Tour

Dynamics AX has a known limitation: Only eight instances of the application's workspace can be opened simultaneously.[2] Any more than eight will cause the entire application to crash (which, by the way, is an issue that could have been caught by using the *Test One, Get One Free tour*). Because of the complexity involved with fixing this issue, it was decided that users should simply be prevented from opening more than eight workspaces so that they never get into the situation where the application crashes.

When I learned of this behavior, the Taxicab tour immediately came to mind. Specifically, I began to think of all the possible ways that a user can open up a new workspace. Like any experienced taxicab driver, I came up with several routes:

---

[2] Note that although multiple application workspaces are created, in this scenario they are tied to a single Ax32.exe process.

- Clicking the New Workspace button on the Dynamics AX toolbar
- Sending the Ctrl+W hotkey
- Executing the New Workspace menu item under the Dynamics AX Windows menu
- Clicking the New Workspace button that exists in the Dynamics AX Select Company Accounts form

Once I was equipped with all the possible routes, I then proceeded to open seven application workspaces. With seven workspaces open, my first objective was to verify that an eighth workspace could be opened using each of the routes. As it turned out, each route was successful in doing so.

Next I applied the *Blockaded Taxicab tour.* Now that I had eight workspaces open, my objective was to verify that the user was blocked from opening a ninth workspace. I attempted to travel the first three possible routes, and in each case I was prevented. However, when I got to the fourth route, it was still possible to create a new workspace. As a result, I was able to launch a ninth workspace, which caused the application to crash.

A Bug Collected Using the Taxicab Tour

Like most applications, Dynamics AX provides several common menus to the user, such as View, Windows, and Help. To test Dynamics AX menu behavior, I executed each menu item to ensure that the desired action was performed. This, of course, is a pretty straightforward approach; so to make it more interesting, I decided to apply the *Taxicab tour.* Specifically, I executed menu items by traveling the following routes:

- Clicking the menu and menu item with the mouse
- Sending the hotkey that corresponds to the menu item
- Sending the menu's access key followed by the menu item's accelerator key

Sure enough, I discovered a bug when I used the third route (using the access and accelerator keys to execute the menu item). For example, to execute the Help menu's Help menu item, I attempted to send the Alt+H access key to open the menu, followed by the H accelerator key to execute the menu item. Surprisingly, I did not get that far because the Help menu failed to even open. On the bright side, this was an important accessibility issue that was identified and fixed before the product was shipped.

Bugs Collected Using the Multicultural Tour

Dynamics AX is shipped in many different languages, including right-to-left languages, so it is important to verify that the application supports globalization and localizability. The *Multicultural tour* has helped to discover many bugs in this area.

Example 1

One bug that I discovered while using the *Multicultural tour* involves the tooltips that display menu access key information to the user. Consider the Windows menu tooltip, which in English is displayed as Windows <Alt+W>.

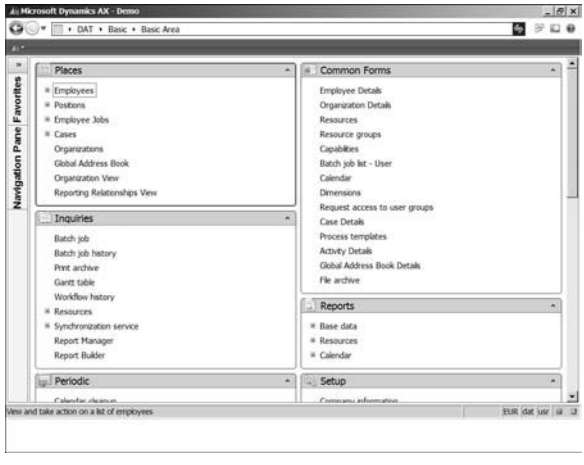
I opened Dynamics AX in various languages, such as Italian, and noticed that the Windows menu tooltip displayed Finestre <Alt+W>. Although the name of the Windows menu had been properly translated, the access key had not. Instead, it should have displayed Finestre <Alt+F>.

Example 2

A majority of Dynamics AX controls are custom, so it's very interesting to run the application in right-to-left languages and verify that the controls behave correctly. In fact, a great example of a bug found by one of my colleagues involves the Dynamics AX Navigation Pane, which can be put in either an expanded (see Figure 6.1) or collapsed state (see Figure 6.2) by the user:

FIGURE 6.1 Expanded Navigation Pane.

FIGURE 6.2 Collapsed Navigation Pane.



Notice that the << and >> buttons at the top of the Navigation Pane are used for changing the pane's state. When my colleague launched Dynamics AX in a right-to-left language, clicking the << arrow button simply failed to collapse the Navigation Pane; however, these buttons worked fine in left-to-right languages.

Both of these bugs would have gone undiscovered if we had not taken the *Multicultural tour* through Dynamics AX.

Tour Tips

While applying the tours described in Chapter 4, I compiled the following list of tips as a "traveling" companion for the savvy tester.

Supermodel Tour

When we are testing a product with a GUI, the *Supermodel tour* is vital to eliminating glaring flaws in the interface. A useful tip for using this tour to find even the most subtle flaws is to combine it with some of the other tours described in Chapter 4.

Combine with Supporting Actor Tour

While perusing the interface, always take care to look beyond the current window or control that you are primarily focused on to see how the rest of the application appears. This technique is comparable to the *Supporting Actor tour,* where you must "turn your attention 10 degrees left or right" to get the full effect. For instance, I once found a bug where I opened a pop-up window on a form and the title bar became grayed-out, thereby making it appear as though the entire form had lost focus. While my main attention was on the pop-up window, by taking a step back to look at the entire form, I caught a very subtle bug with the title bar.

Combine with Back Alley\Mixed Destination Tour

The main premise of the *Back Alley\Mixed Destination tour* is to test how different features interact with one another. With respect to a GUI, it's important to verify how features of the external environment affect the appearance of an application. Here are a few examples:

- Modify the OS display settings, such as to high contrast, and take a *Supermodel tour* through the product to verify that all controls, icons, and text display properly.
- Use Terminal Services to remote into the machine that has the application installed and verify that there is no painting or flickering issues as windows in the application are drawn.
- Run the application with dual monitors and verify that menus and windows display in the correct monitor.

Because most first impressions are based on looks alone, bugs involving the appearance of an application can lead to perceptions of an unprofessional, poorly engineered product. Unfortunately, these types of bugs are often deemed as low priority to fix. A handful of appearance-related bugs in an application may seem harmless, but together they often have a cumulative effect on usability and therefore should *not* be ignored.

Rained-Out Tour

The *Rained-Out tour* concentrates on terminating functionality and verifying that the application continues to behave correctly. The two tips that I provide for this tour are actually already mentioned in Chapter 4; however, I want to emphasize them here because they truly are very helpful in detecting bugs.

First, it is important to change the state of the object under test *before* canceling out of it. Let's use a form as an example: Do not just open a form and immediately close it. Instead, alter either the form's or application's state before closing it. To demonstrate this, here are some actual bugs that I have found in Dynamics AX using this technique:

- I opened a form and then opened a pop-up window that was parented by the form. With the pop-up window still open, I clicked the form's X button to close it. As a result, the application crashed because the form failed to properly close the pop-up window before closing itself.
- After opening the User Setup form, I left the form open and switched to a different module in the application. Next, I clicked the User Setup form's Cancel button, which caused the application to crash.

Second, it is imperative to reattempt the same scenario after canceling out of an operation. I used this very technique recently while performing exploratory testing for a new feature planned for the 6.0 release of Dynamics AX and found yet another client-crashing bug. This new feature ensures that creates, updates, and deletes for inner/outer joined data sources occur within a single transaction. While testing updates, I decided to cancel (discard) these changes by clicking the Restore button on the form's toolbar. As a result of clicking Restore, my changes were discarded and replaced by values in the database table. I then reattempted to update the same record again, and lo and behold Dynamics AX crashed.

Landmark Tour

Some applications are so large, such as ERP solutions, that it's overwhelming to even think where to start with the *Landmark tour,* simply because there are so many features. In some cases, testers may not even be very familiar with other features outside their primary area of responsibility. A tip for combating this issue is to pair up with another person who is an expert in a complementary feature area.

Using Tours to Find Bugs

By David Gorena Elizondo

I started working for Microsoft as soon as I graduated from college. (I had done a summer internship for Microsoft a year prior.) I joined as a Software design engineer in Test, and I was part of the first version of Visual Studio Team System in 2005. My job at the time included testing the tools within Visual Studio for unit testing, code coverage, remote testing, and so forth. I am a tester who tests testing tools!

Throughout my four years at Microsoft, I became involved with the different testing approaches and methodologies used inside and outside the company, from test automation to script-based testing, E2E testing, and exploratory testing. I have experimented with a lot of different test techniques. It is in learning and performing exploratory testing that I found my testing passion. I've now been using exploratory tours for over a year. Organizing my thinking around the tourist metaphor has substantially increased the number of fixable bugs that I find in features that I'm responsible for testing.

Although all the tours have been valuable to me at one time or another, I've seen through experience that certain tours work best under specific circumstances. I will try to share some of my thoughts on when to use them, and the kinds of bugs that I've found while testing the test case management solution that I've been working on for the last year.

Note that all the bugs I describe here have been fixed and will be unable to plague any users of our test case management system!

Testing a Test Case Management Solution

I used the exploratory tours with the test case management system that we've been developing for the past year and a half on my team. Before I describe how I used the tours, I will describe our product because its features and design affect the way I chose and executed the tours.

The test case management client works hand in hand with a server, from which it basically pulls what we call "work items"; things such as test cases, bugs, and so forth to display for a user to manipulate. Without the server, the client can basically do nothing. Just by knowing this piece of information, you can easily tell that tours such as the *Rained-Out tour* and the *Saboteur* yielded a lot of bugs. Imagine canceling a server operation halfway through its completion, or just getting rid of the server itself. Then if you think about it a bit deeper, you realize that a work item (or anything on the server) can be modified at the same time by more than one client. So, the *TOGOF tour* yielded a good amount of bugs, as well. Updates are happening all the time throughout the application, so the *FedEx tour* is also one to target.

Complete books on the tours presented in this book could be written. Until then, here are some of my bug-finding experiences.

The Rained-Out Tour

When you're working with a client application that works closely with a server, you can assume that any unexpected server activity can cause weird things to happen (both on the server and on the client). Interrupted requests to a server is tricky business, as is refreshing client-side data. Think about it: If you open a page that starts loading information from a server, and you immediately click Refresh, one action is canceled, and a new one starts immediately. The software on both sides of such a transaction needs to be on its toes. That's why this tour finds tons of bugs. A few of my favorites are (using the exact titles that were entered into our bug management system) as follows:

- **Bug: If we cancel initial connect to a project, we can no longer connect to it manually.**

  Whenever our test case management system is launched, we remember the previous server (that contains the test case and test data repository) that the user was connected to and initiate an automatic connection to this data store. If users want to connect to a different store, they have to cancel this operation. It turns out that the developers didn't think through the cancellation scenario well and caused an environment variable to be deleted under certain circumstances; so when I loaded a new version of the test data repository, the connection to the server was lost.

- **Bug: When deleting a configuration variable and canceling or confirming, you get prompted again.**

  This would have been a nasty bug to ship with. It happened whenever I tried to delete an existing test repository variable and then canceled the request at a certain UI prompt. Because the *Rained-Out tour* makes a tester conscious of recognizing complex and timing-sensitive actions, it led me to naturally think of this test case.

  The *Rained-Out tour* makes us think beyond explicit actions. There are times when a feature or a product has to implicitly cancel an action that has already started (often for performance reasons). This was exactly the case for this bug.

- **Bug: Plan Contents: Moving between suites does not cancel the loading of tests.**

  It turns out that whenever we chose a test repository, the application would start loading the test cases that were associated with it. The tour led me to think that there had to be an implicit cancel action whenever I quickly chose a different repository, and if not, there would be performance issues. It turned out to be true; we were not canceling the action properly, and performance was really bad.

  The *Rained-Out tour* makes it clear to the tourist: Cancel every action you can, and cancel it many times and under many different circumstances. It is raining, so scream and yell for everyone to cancel their plans! This strategy led me to bugs like the next one.

- **Bug: Time to refresh starts growing exponentially as we try it several times.**

  Knowing that a refresh action will basically cancel any in-progress activity, I decided to click that Refresh button many times (quickly), and this actually caused horrible performance issues in the product.

  The strategy used to find this next bug was exactly the same one mentioned in the previous one.

- **Bug: Stress-refreshing test settings manager crashes Camano.**

  However, the result was much better for a tester's perspective: a crash! Clicking the Refresh button like a maniac actually caused the application to die.

The Saboteur

The *Saboteur* forces you to think though the application's use of resources so that you can vary the amount of resources available, and thus potentially find scenarios that will cause it to fail, as discussed here:

- **Bug: Camano crashes when trying to view a test configuration when there is no TFS connection.**

  TFS is the server used to store test cases and test data. The *Saboteur* caused me to think through many scenarios where availability of TFS was crucial and where good error-handling routines needed to be in place. Making the server unavailable at the right places sometimes found serious crashing bugs, and the resulting fixes have made our product robust with respect to server connection problems.

  In this next bug, the strategy was the same one: looking for a resource that the application uses.

- **Bug: Camano crashes at startup time if Camano.config becomes corrupt. Camano continually fails until the config file is corrected.**

  The resource in this case was a configuration file that the application uses to persist data between sessions. The *Saboteur* requires that all such state-bearing files be tampered with to see whether the application is robust enough to handle it when those persistent resources are corrupt or unavailable. Playing around with these persistent files not only found some high-severity bugs, it was also fun and had me anticipating when and where the crash would occur. Given that no one else on our team had thought to try these scenarios made me happy to have the *Saboteur* in my tool kit.

  This next bug shows the exact same strategy as the previous one but with a new twist.

- **Bug: Camano crashes when config file is really large.**

  Finding this particular bug consisted of playing around with the configuration file and creating variants of it and modifying its properties. The

Saboteur suggests making it read-only, deleting it, changing its type, and so forth. But the property that worked for me was its size. When I created a very large config file, the application was simply not able to cope with it.

The FedEx Tour

Our test case management solution processes a great deal of data that flows freely between the client and the server: bugs, test cases, test plans, test plan content, and so on. All this information has to be refreshed appropriately to remain in sync, and this is not easy for an application where multiple activities work with the same artifacts simultaneously. The *FedEx tour* is tailor-made to help a tester think through such scenarios. The following bugs show some defects that the *FedEx tour* found.

- **Bug: Test plan not refreshing automatically after coming back from a work item.**

  The test plan contents show the actual test cases in the test plan. The strategies that the tour led me to were actually modifying properties on test cases (and the test plan itself) and ensuring that all these were refreshed appropriately. It turns out that if we modified the *name* of a test case, and came back to view the test plan, you had to manually refresh the activity for the test case to show the updated name.

  We found a number of bugs similar to this next one.

- **Bug: When a test plan is selected in the TAC, and we modify one of its artifacts, Camano crashes.**

  In this particular scenario, modifying a property (such as configuration name) of a test plan, while showing that same property on another activity, would crash the application because it would not be able to cope with the property change. If you think about it, this bug was found using the exact same strategy as the previous one: modifying properties and artifacts, and making sure they were refreshed correctly somewhere else.

  This next one was an interesting bug.

- **Bug: Camano will crash forever and ever if we have a test plan that uses a build that has been deleted.**

  Our test plans can be bound to builds, meaning that we can link a test plan to a particular build. However, if we deleted the build that a test plan was bound to, the application would crash every time we opened that particular test plan. Here the *FedEx tour* is helping us identify those types of data dependencies and guiding us through thinking about such associations between data elements in a methodical manner.

The TOGOF Tour

The *TOGOF tour* will find bugs in applications that can be used by multiple simultaneous users. We found the following bug while having multiple users active within the application at the same time.

- **Bug: Test Configuration Manager: Camano crashes when you "Assign to new test plans" if the configuration is not up-to-date.**

Test configurations have a Boolean property called "Assign to new test plans" (which can be toggled on or off). It turns out that if user A and user B were looking at the same exact copy of any given configuration (with say, the Boolean property set to true), and user A changed the property to false (and saved it), whenever user B tried to make any change to the configuration and saved it, his application would just crash. This shows a bug that would have been very difficult to catch if only one user was testing the application. The *TOGOF tour's* strategy is very clear in these kinds of scenarios: Test different instances of the application at the same time to find bugs that will be difficult to find otherwise.

The Practice of Tours in Windows Mobile Devices

By Shawn Brown

In 2000, Microsoft shipped a product that could fit on a device to be carried around and perform many of the same functions that a full-size PC could perform. This device was called the Pocket PC and started a series of releases of Windows Mobile. Throughout the releases of Windows Mobile, more and more functionality was added, and therefore the ecosystem to test became more and more complex: from the first, nonconnected PDA-style device, to the multiconnected GSM/CDMA, Bluetooth, and WiFi devices that could stay connected and provide up-to-date information without the user even having to make a request.

During this evolution, the testing of these devices also had to evolve. Constraints such as memory, battery life, CPU speed, and bandwidth all had to be considered when developing and testing for this platform. In addition, being the first multithreaded handheld device allowed for more functionality and more applications working together to give the user more "smart" behavior, and hence the name Smartphone came into the picture. Now, take that environment and add an additional "unknown" variable into the picture called ISVs (independent software vendors). ISVs may use an SDK (software development kit) to create applications on this platform to expand its capabilities and to a buck or two for themselves through revenue, and may thus cause a new testing challenge. These creative and smart ISVs may not abide by certain development practices that internal-to-Microsoft developers are trained to do, or may want to push the boundaries of what the platform can do, and therefore they may potentially cause unexpected issues when deploying their applications to the devices. Some

global measures can be put into place, but as a tester on a Mobile platform with a fairly extensive SDK, we cannot ignore the potential of one of these add-ons to affect the overall health of the platform with respect to all the categories of test. Given the fluidity and the overall challenges of testing Windows Mobile, this is a great product to hone testing skills on.

During my career with Windows Mobile, I have owned the testing of Connection Manager, certain Office applications in the earlier releases, and one of my favorites, the phone functionality.

I catch myself hunting for areas of the product that go unnoticed for lengths of time and capitalize on their neglect. When I find a flaw in a product, it creates a sense of accomplishment and job satisfaction. As my career in test evolved, I took notice that in addition to being able to discover new and creative ways to break my product, I was also taking a closer look at how to prevent bugs from getting into the product and was paying more attention to the end-to-end solution of the system. Bug prevention starts with testing the initial design. Roughly 10 percent of all of the bugs found over the years are spec issues. Looking back, this 10 percent, if they had made it into the product, could have caused more bugs as well as more time to complete the project. As you can understand, finding these issues early helps the product and the schedule.

My Approach/Philosophy to Testing

My philosophy to testing is quite basic. Find the weaknesses of the product before anyone else does, and ensure the product's strengths are highly polished. This approach requires a continual watch on the product and the environment around it that can affect the performance or functionality. How can you look at the environment you are positioning yourself to test in a way that encompasses everyone who is going to interact with it? Then, how do you prioritize your attack?

Independent of the test problem, I begin by defining the problem statement. What is the desired end goal? I determine the variables. I formulate an approach (keeping it simple). After a simple solution is defined, I determine the reusability and agility of the solution. Where are the solution's trade-offs and weak points? I have used this approach in developing and testing hardware as well as software.

One example of my use of this methodology is when I was developing a shielding technique to enable electronic components to be used in the bore of an MRI. This violent magnetic environment was never meant to have metallic objects inside (because of safety and because of the potential to disrupt the sensitive readings). Knowing the problem statement was "create a method for housing electronics in the bore of an active MRI unit without disrupting the readings more than 5%," I started by gaining a better understanding of the environment: the strength of the magnetic field, the frequency of the gradient shifting when the unit is in active mode, the method of the readings, and the requirements around what the electronics inside the bore were expected to do. This information narrowed the solution to two main options:

1. Create a shield to prevent the magnetic from penetrating into the housing of the electronics.

   or

2. Create a shield that prevented the RF from escaping from the housing.

After researching the known methods for both, one being a ferrous metal that would prevent the magnetic fields from entering the box, and the other, a highly permeable metal to shield the RF from escaping from the box, I realized that traditional shielding methods would not fit this goal. (Because the RF that would be generated by either material upon placing them inside the bore during active mode would most likely degrade the reading beyond the 5 percent margin.) In addition, the ferrous material could cause more of a hazard to anyone in the room, given the highly concentrated and erratic magnetic field being produced could cause any ferrous metal to be hurled through space at a deathly rate. Therefore, one more option was removed from the equation. Given the material for shielding could not be a solid plate (because of the RF induced by the environment), it had to be unconnected loops. After I calculated the penetration of the RF, the end results were to create a <1cm grid of nonconnecting loops of copper, which would prevent the RF from escaping from the box and would mostly prevent the RF from entering or being generated by the material itself. My approach led to the shielding technique being a success, and I continue to use this process when testing software.

Testing is a continually growing domain, much like any other engineering discipline. To keep increasing your product's quality, expose how you test to your developers/designers. The more they know about how you are going to test it, the more they will try to create a design that will account for the methods you will use to exploit the gaps in their design. Testing can be difficult, because even though you want to maintain the edge on being able to break any product, you also want to help your developers write better code. Therefore, you must always be one step ahead of your developers. As they learn how you test and you learn how they approach solving algorithm problems, your ability to break their code becomes more and more difficult, Success! However, it is still your job to find the bugs in the product.

Interesting Bugs Found Using Tours

Using the Rained-Out Tour

While performing exploratory testing on a previous release of Windows Mobile, I caught a race condition with "smart dial" functionality. In this instance, it was known that a side process was being used to do a background search on the search criteria to aid with performance. The larger the data set, the longer it takes this process to complete. This turned out to be an excellent opportunity to use the *Rained-Out tour.* After loading more than

4,000 contacts onto the device under test, I entered a search string that was expected to end in no results. As the side process was churning away in the background, I changed the search string by deleting one character. The expected result was to still have no matches, but the side process was not done with checking the entire data set, and therefore the next filter, which is designed to check the results of the initial side process, was passed the data that was not checked yet. This resulted in an incorrect IF statement, and data that did not match the search clause was displayed incorrectly. If this bug had not been caught early and through exploratory testing, it could have caused more downstream design challenges.

Another example of using the *Rained-Out tour* is in an issue found and then fixed in the Bluetooth bonding wizard in Windows Mobile. After creating bonds with a few headsets and peripherals, I proceeded to use the *Rained-Out tour* during the connection-request phase. When the listed peripherals were all disconnected, there was an option to connect one of them to the phone. I selected one of the BT peripherals and selected the Connect option. I then noticed a time delay between when the connection request was made and when the connection or the failure dialog was actually shown. During this time, I tried all the available menu items. They all functioned properly during this state. It wasn't until I moved focus away from the existing item, then back to the item with the current connection request, that I noticed that the connection option became available again. So as a tester using the *Rained-Out tour,* I selected that option. Doing so keyed up another connection request to the same peripheral. I performed the same steps a few more times in quick succession before the initial connection request completed, and voilà, a sequence of failure dialogs popped up. Ultimately, the code was able to handle this situation, but there was no need to be able to continue to make multiple connection requests to the same peripheral, and therefore the bug was fixed.

Using the Saboteur

A contact list is linked to a number of other functionalities (call history, text messaging, speed dial, and so on). Knowing this, I used a *Saboteur* and created a list of contacts and some speed-dial entries. Now for the unexpected situation to be triggered: A sync error was simulated where the contacts on the device were all removed from one of the linking databases on the device, but the device still thought they were synced (because they still existed on the device). Therefore, syncing was seemingly successful, but the database that housed the links to the speed dial did not exist anymore, and therefore a blank speed dial was shown.

Using the Supermodel Tour

Here is an example of when I used the *Supermodel tour* on a Windows Mobile device and explored UI centering and anchor points using different resolutions. After booting an image with a resolution that I knew was not used a lot, I navigated around the device performing simple user tasks (for

example, creating contacts, checking email, and checking calendar events). During the calendar event navigation, I noticed that when I selected to navigate to a specific date, the calendar week view centered itself onscreen. A number of views could be used, and so I inspected each view. Not until I got to the month view did I see a centering issue. After I changed which month to view via the month picker, the repainting of the month that was selected was centered in the middle of the screen, instead of being top justified like every other view. This incorrect centering occurred because of a missing flag in this one view.

Example of the Saboteur

A good time to use the *Saboteur* is when testing an application that utilizes data connectivity. Data connectivity is a fickle beast and can come and go when you least expect it. Using this tour, I was able to find a problem with a device-side client who requires a connection to provide its functionality. Not only was this application coupled to connectivity, it also had an impact on other locations where the user's entity can be used. I'm speaking of Instant Messenger, of course. I successfully configured this application and signed in. Now, knowing my device is a 2.5G device, which means it can handle only one data pipe at a time, I then called the device from another phone, thus terminating the active connection, which, on Windows Mobile on GSM, puts the device connection into a suspended state. Then I signed in to the same account on my desktop, attempting to fully sever the connection to that service on the device. Voilà, the device-side client never got the notification that it was signed in to another location but it also appeared to be signed out. However, in the menu options, the option to Sign Out was still available (because it never was told it was signed out).

Another way to be a saboteur is to turn the device's flight mode on, which turns off all radios (for when the user goes on an airplane but still wants to use the device). Many times, applications do not listen to radio on/off notifications, and this can cause a limbo state to occur. Sure enough, signing on to IM on the device and then turning off the cellular radio, which in turn removes all GSM connectivity, was not detected by this application, and the application is in a perpetual "connected" state, which is quite confusing to the user.

Example of the Supermodel Tour

An example of a bug found by using the *Supermodel tour* coupled with focus specifically on usability and intuitiveness of designs is as follows. While traversing through a connected mapping application on Windows Mobile, I decided to see how easy it was to get directions from my current location to another location, such as a restaurant. I launched the application, the device found my current location, and then I decided to select the option to get Directions from A to B from the menu. In this UI, there was no intuitive way to select my current location as my starting point. Given that I did not know

the address where I currently was, I just then decided to enter the point B location and hit Go. An error appeared stating it needed to know where I was starting from. Interesting, because upon entering the application, it made it a point to tell me where I was. This lack of completeness can cause much pain when using the application, and a simple addition to prepopulate this field could be the difference between someone using this application all of the time versus it getting slammed in a review on the Web.

The Three-Hour Tour (or Taking the Tours on Tour)

by Shawn Brown

I found a fun way to use the tours to build teamwork and increase morale and find a lot of great bugs in the process. Given that our product is meant to travel, we assembled the team and headed off campus on a testing adventure. We brought our devices, chargers, tools, and tours and drove around looking for bugs. We set a goal of finding 20 bugs in 3 hours and to have a lot of fun. The result, we found 25, and no one was ready to return to the office! Here's the report.

Applications can function perfectly in a controlled test environment, but when put in the hands of an end user, coupled with its interactions with other applications or protocols on a mobile device, is when it gets interesting; and it is critical to be correct. In the ever-increasing mobile environment, technology is being put in awkward situations, which makes end-to-end testing more and more critical. Many years ago, a computer would sit in a dedicated room and not move around, which made "environment" testing not as critical. Yes, you would have to take into consideration changes in bandwidth if connectivity was available, and different users, but even so, the user segment was more adaptable to this new technology. As technology grew, it became more interesting, and the more interesting it got, the more it fell into the hands of users who were not prepared, nor wanted, to understand how it works; they just wanted it to work. As a technology's user segment evolves, our testing strategies also have to evolve to ensure that what is important to the end user is at the forefront of our minds when we create a test plan. This does not mean that we abandon the tried-and-true methods and "test buckets"; although we may have to approach and prioritize how we use these buckets depending on the technology being tested.

With that said, being mobile and staying connected has become a necessity in a lot of regions. This new environment variable needs to be taken into consideration when defining a test strategy or plan. For the sake of brevity, this discussion focuses on Windows Mobile. Windows Mobile is composed of many moving parts, and its customers can be anyone from a teenager in high school to a retired businessperson who wants to keep current with news, email, and so on…wherever he/she goes.

Knowing that our end users use devices in a variety of ways, a small set of test engineers set out to accomplish some predefined tasks that mobile devices can do in the wild. We used multiple tours and uncovered a number of bugs in a short amount of time. We found the *Supermodel tour,* the *Saboteur,* and the *Obsessive-Compulsive tour* most useful, although we could have leveraged just about any tour. The *Supermodel tour* was primarily used in the mobile platform during this "tour of tours" to flush out interoperability and integration between applications and tasks. While taking these tours on tour to accomplish the tasks, each attendee also had their "attention to detail" radar cranked up. In addition, we paid extra attention to certain requirements of the device because they have been proven time and time again to aggravate users when malfunctioning and delight users when "it just works." These requirements related to always-on connections, performance in accomplishing tasks, or navigating (especially via an error message or a dialog to guide users back on track after something went wrong).

As we took the tours on tour, we uncovered new bugs that might not have been found this early in a static testing environment. Using the *Obsessive-Compulsive tour,* we discovered that attempting to use every WiFi hot spot in the area to browse the Web eventually caused IE to unexpectedly stop working. This could be due do an authentication error or a protocol that has not been accounted for. In addition, this tour also uncovered an authentication page that IE could not render and therefore could never be successfully used.

The *Saboteur* happened to be one of the more fun tours during a team outing such as this one. As testers, we want to break things, and this notion of being a saboteur of the device just fits. As we were out in public, there were many discussion of "what if" this happens or that happens, such as, "I wonder what will happen if I'm in a call with you, and I attempt to play a song?" If that functioned properly, the next question would have been this: "How about we pull the storage card that the song resides on and see what happens?" This constructive destruction was like a snowball rolling down a hill picking up speed. In addition to finding a number of new issues in the product in just three hours, it was a team-building outing that enabled brainstorming to teach each other new ways of thinking.

During the entire time, the *Supermodel tour* was at the forefront of our minds. Each attempted task was put under high scrutiny, and within just a few hours, a large number of fit and finish bugs were discovered (for example, error dialogs that really were not helpful, and messages that could not be sent because of either network reliability or as a result of our sabotage [and which were never sent, even after connectivity was reestablished]). In addition, bugs were found where the input panel covered the edit field, which can be quite annoying or confusing to the end user.

The Practice of Tours in Windows Media Player

By Bola Agbonile

I am a graduate of the University of Lagos, Nigeria, where I attained an MBA (1996) and a Bachelor's degree in Electrical Engineering (1990). I presently work on the Windows Experience (WEX) platform with the Windows Media Player team as a Software Development Engineer in Test (SDET), a role that I continue to relish daily.

As an SDET, my role is primarily to work with others on my team to ensure that a high-quality product is what we present to our customers. To achieve this, one of my key roles as an SDET is to validate that the finished product satisfies what our target-market customers require and adheres to the written specifications as outlined by the Program Manager. Another one of my primary roles is to ensure that the product can withstand rigorous tests thrown at it.

Windows Media Player

In the Windows Experience division, on the Windows Media Experience (WMEX) team, I have worked with others at enhancing Windows Media Player (WMP) from previous versions, with the sole aim being to satisfy our target market by providing a solid, robust, functional, media player. For instance, it was with WMP 10 that we first introduced being able to synchronize *to* devices, and with WMP 11 we took it one step further by adding greater support for Media Transfer Protocol (MTP) devices, including the ability to have an automatic sync partnership and allowing for synchronization *from* the device. WMP allows for sync, burn, rip, and playback of numerous file types, including pictures and DVDs. WMP 12 introduces a lightweight player mode for quick, easy, and clutter-free playback with practically no UI chrome.

WMP is an application that is UI-centric. As such, the tours relevant to this kind of application are the ones I use. To be specific, WMP's input sources are via text boxes, check boxes, option buttons, and "shiny discs" (CDs, DVDs, and CD-R[W]s), while its output is as audio, video, and dialogs displayed to the user.

Following are the tours that I use for testing WMP 12, with examples of things that I have encountered along the way that I consider to be interesting.

The Garbage Collector's Tour

Garbage collectors are systematic, house to house, driveway to driveway. Some testers might, as Chapter 4 suggests, be systematic by testing features that are close together in an application. I apply it a bit differently and arrange features in buckets according to their similarity. For WMP, the first categorization bucket could be "All UI Objects." The next bucket will be "Dialogs," followed by "Text Boxes," "Boundaries," and so on. Then, the garbage collector can begin her systematic collection.

WMP buckets look something like this:

1. WMP's Player mode
    a. Transport control
        1. Shuffle
        2. Repeat
        3. Stop
        4. Back
        5. Play
        6. Next
        7. Mute
        8. Volume
    b. Buttons
        1. Switch to Library mode
        2. Switch to Full Screen mode
        3. Close WMP
        4. Minimize WMP
        5. Maximize WMP
    c. Seek bar
    d. Title bar
    e. Right-click context menu
        1. Spelling of labels
        2. Persistence of user's choice
        3. Navigation using keyboard
    f. Hotkey functionality
        1. Alt+Enter
        2. Ctrl+P
        3. Ctrl+H
        4. Ctrl+T
        5. Ctrl+Shift+C
    g. Dialogs
        1. Options
            Tabs
            Options buttons
            Check boxes
            Text boxes
            Command buttons

2. Enhancements
    - Next button
    - Previous button
    - Hyperlinks
    - Option buttons
    - Check boxes
    - Labels
    - Hover tooltips
    - Sliders
    - Mouse pointer
    - Drop lists
3. Default language settings
    - Drop list
    - Arrow-key navigation
    - Command buttons

h. List pane

i. Center pane

1. Play all music shuffled
2. Play again
3. Continue playlist
4. Go to Library
5. Play previous playlist

j. External links

1. Windows Help and Support
2. Download visualizations
3. Info Center view

One of the benefits of this tour is that with the categorization of the application, the tester can systematically walk through and identify features that might otherwise be overlooked. For example, while taking this tour and looking WMP over, I came across a bug that I have now logged. The bug is that in the center pane, "Play previous playlist" does not actually start playback unlike "Play all music shuffled" and "Play again." A user would expect consistent behavior, and if I had not applied the *Garbage Collector's tour,* I would have missed this difference.

The *Garbage Collector's tour* made me take a long hard look at WMP and systematically categorize all the buttons available in the center pane (for instance) and then test each item's functionality for consistency when compared with other items in a similar bucket.

The Supermodel Tour

This tour should be on your list when success is to be measured by the quantity of bugs found. For example, if there is to be a bug bash, and there are no restrictions on which feature gets tested, it is beneficial to be one of the first people on the ground when the whistle blows. Examples of supermodel bugs that I classify as "low-hanging fruit" are typographical errors. Early on in the product cycle is when this category of bugs is more common. To effectively spot text bugs, one should read *each* word and then count to *two* before reading the next word. In my opinion, this is the secret to spotting typos and grammatical errors.

Figure 6.3 is an example of a bug that I found on a WMP dialog. Take a look at the dialog and see whether you can spot the typo in it.

FIGURE 6.3 An easy bug to miss.

Spot the bug? It is in the second paragraph. "Do you want to allow the Web page full *to access your*…." This should be, "Do you want to allow the Web page full access to your…." It is easy to read too fast and correct the sentence in your head while not seeing the error in front of you.

The Intellectual Tour

When running manual tests, it pays to keep asking "What if?" For instance, when looking at the Player versus Library mode of WMP, one could ask this: "What if the Library mode is busy and then WMP is expected to be in the Player mode?"

Having asked this question, one would then investigate possible answers by performing a task that is exclusively for the Library mode (such as ripping an audio CD) and then performing a task that is exclusively for the Player mode (such as DVD playback). The whole purpose of this is to see whether this scenario has been handled or whether one would get the undesirable situation of WMP running in *both* modes (see Figure 6.4).

FIGURE 6.4 One application running in two different modes.

Twenty-Five Examples of Other WMP-Related "What If?" Questions

What if a user decides to…

- Burn via Shell *and* via WMP concurrently?
- Change options from one to another?
- Change view from smaller to larger UI object?
- Delete via WMP a file that is already gone?
- Double-click a UI object?
- Exit from Full Screen mode with ongoing playback?
- Exit WMP with an ongoing process (sync, rip, playback, burn, transcode, and so on)?
- Insert an audio CD with auto-rip enabled while a context menu is displayed?
- Place an ampersand (&) in an editable field?
- Play back content over the network, and then disable network connectivity?
- Play back an expired DRM track after moving back the system clock?
- Play back two DVDs concurrently, one with WMP and one with a third-party app?
- Play back two DVDs using WMP concurrently from different ROM drives?
- Press a keyboard key during an ongoing process?
- Read Web Help?
- Repeat a hotkey sequence?

- Sync content to an already full device?
- Sync/burn/rip content on a PC with no hard drive space?
- Synchronize to two devices concurrently?
- Synchronize/burn/playback, and then hibernate the PC?
- Transcode a file on a laptop using battery power?
- Transcode a file prior to DRM license acquisition?
- Turn *off* an option, and then verify whether the option is in the off state?
- Uncheck *all* the options on a dialog (like Customize Tree View)?
- Use word wheel search, and then drag and drop?

The Intellectual Tour: Boundary Subtour

This is one of my favorite tours because it makes me feel like a real detective. The *Boundary tour* involves conducting tests close to the upper and lower boundaries, all the while looking for a breaking point. It's a specific case of the *Intellectual tour* that suggests that we ask the software hard questions.

Classic examples include the following:

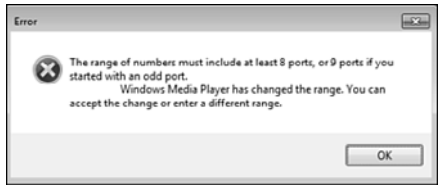
- Filling a text box with its maximum number of characters or null
- Creating a very deep folder hierarchy, and then placing a media file in the last folder and attempting to play back the media file in WMP
- Clicking a button very close to its outer edges to see whether the click command will still be recognized

For example, in WMP, after I attempted to type characters in a text box meant for only numbers, the mitigation taken was to ensure that a user can type in only numbers (see Figure 6.5).

FIGURE 6.5 Mitigation taken to prevent non-numeric inputs.

Conducting more tests on a different tab unearthed a dialog with bad indentation (see Figure 6.6).

FIGURE 6.6 Dialog with bad text indentation.



The bugs found while on the *Boundary subtour* are varied in nature, including buffer overruns, bad handling of data, and UI formatting errors.

The Parking Lot Tour and the Practice of Tours in Visual Studio Team System Test Edition

By Geoff Staneff

In 2004, I came to Microsoft with zero programming experience, but a Ph.D. in Materials Science from the California Institute of Technology. I spent the first nine months taking computer science courses and test training in the mornings while working on the Windows event log in the afternoons. By the end of the first year, I owned 50k lines of native test code, filed 450 bugs, and had an 80 percent fixed rate. Since moving to Visual Studio, my focus has changed tremendously. Our work, both the product we test and the test code we write, contains predominantly managed code, and most of our regular testing is manual or semi-automated. Although the particulars of the day-to-day work are vastly different, the act of testing is the same. I keep a lab book at my desk to track identified and reproduced defects and to note interesting areas of investigation that bear future attention.

Tours in Sprints

Our development and test teams work closely during each sprint. It is standard practice for the test team to build new bits between check-ins from our developers and participate in code reviews for the same. As such, we have some uncommon advantages in knowing where certain kinds of defects are likely to be found. Instead of jumping right into these known risk areas, I take a more methodical approach ingrained by my years of studying experimental science. The first thing I do with a new application (or version) is take a brief *Parking Lot tour* to get the lay of the land. As I learn how the various parts of the application are supposed to work together, I take notes on which areas deserve in-depth attention and follow-on tours. The next wave of tours focuses on individual features, typically with an imposed bias such as accessibility, expose all error dialogs, or force all default values (permutations pulled straight out of *How to Break Software* and summarized in Chapter 3, "Exploratory Testing in the Small," of this book).

Between the *Parking Lot tour* and the *Breaking Software tour,* I generally pick most of the "low-hanging" fruit in terms of obvious bugs.

My final wave of tours is a much more targeted endeavor. After a couple days observing how the product under test responds, it is time to go for more abusive tours that challenge the assumptions of implementation. The ones I end up using the most are the *Back Alley tour* and the *Obsessive-Compulsive tour,* built on previously observed reactions in the product (observations or behavior that were technically correct, but perhaps incomplete or narrow) or conversations with the developer about his implementation.

Over the course of two such tight sprints, I identified about 75 defects in each, of which all but 5 were fixed before the end of the week in which they were identified. We were working in what we call "low-overhead mode," such that if development could address the issue before the end of the week, test would not formally log the bug. This arrangement was great from both sides of the dev-test relationship; product quality was quickly improved, and no one had to deal with bug-reporting and -documenting overhead. Had the turnaround on bugs been days rather than hours, this arrangement would not have been appropriate, because details of the observed bug could have been lost.

Keeping the pace brisk was facilitated by a cut time each day when a new build would be produced and tested. This had a dual impact. First, development would communicate the fixes we should expect prior to the cut time each day, focusing the testing effort to newly implemented or repaired features. Second, the regular testing start time provided regular feedback to development about the state of their progress. Developers were eager to get their fixes into the next day's testing effort, causing them to work with test to ensure they'd make the test deadline each day, which helped keep the features to spec with limited creep. We maintained a very high rate of code churn and a tight loop of checking in and testing, which allowed us to stay on top of defects and find the important ones early. Testing during these sprints lent itself to exploratory tours, as the testing cycle was iterative and short. Returning to a general tour such as the *Money tour* or the *Guidebook tour* every few days helped ensure that we didn't let any new work slip through undetected. I do not recall any new bugs in those features over the last four months, despite regular use by our team and others.

Reviewing the bugs identified over one of these testing efforts shows a breakdown by detecting tour:

- 9% *Taxicab tour* (keyboard, mouse, and so on)
- 9% *Garbage Collector's tour* (checking resources after releasing/removing them)
- 15% *Back Alley tour* (attempting known bad actions, such as closing a dialog twice)
- 18% *Obsessive-Compulsive tour*
- 19% *Landmark tour*
- 30% *Supermodel tour*

While most of the *Supermodel tour* defects were not recall class, nearly all the *Back Alley* and *Garbage Collector's tours* would have forced a recall or patch on a released product. Although the *Supermodel tours* didn't directly reveal the more severe issues, they did identify places where more-focused tours could follow up and deal real damage. One such example was the identification of keyboard acceleration for the Cancel button in a wizard. This observation identified a bug immediately. Wizards do not typically accelerate the Cancel button with a lowercase *L;* instead, they usually reserve the Esc key for that, without any visual indicator. A follow-up *Back Alley tour* exploited this keyboard acceleration to cancel the wizard several times. This exposed an unhandled exception and a UI timing issue where a modal dialog was hidden behind the UI element waiting on it. Of course, canceling the same wizard instance more than once should also have not been possible.

Whenever I revisit a feature we've previously tested, I run another *Parking Lot tour*. Just recently, using this method I observed two defects in functionality that had been verified not two weeks earlier. Even when I wasn't planning to test this particular part of the feature, just by revisiting the bounds of the code under test, a broken link and an unhandled exception presented themselves for observation. Following a strict script and just getting in and getting your work done as efficiently as possible would have missed these opportunities, leading to increased mean time to detection for new defects in the system.

Parking Lot Tour

The *Parking Lot tour* was born of countless family vacations, where distances on the map didn't look nearly as imposing as they did after driving from place to place. Such ambitious plans often saw the family arrive at a venue after closing time, leaving nothing to do but tour the parking lot and try to make the next stop before it too closed. I used a *Parking Lot tour* to find a usability bug and an app crash during my interview for my current team:

- **Primary objective:** Identify the entry points for all features and points of interest within the scope of testing.
- **Secondary objective:** Identify areas where specific tours will be helpful.
- **Targets of opportunity:** Enumerate any show-stopping bugs on the first pass.

In a way, the *Parking Lot tour* is like a mixture of a *Landmark tour* and a *Supermodel tour.* The first pass isn't going to look too deep and is really more interested in how the code under test presents itself than anything the code under test does.

Test Planning and Managing with Tours

By Geoff Staneff

When testers express an interest in exploratory testing, this can often be interpreted by their managers as a rejection of rigor or planning. Through the use of the touring metaphor, managers can gain repeatability and an understanding of what has been tested, while testers can maintain their autonomy and exploratory imitative. This section outlines some of the strategies and techniques that will prove useful when managing a tour-driven test process.

Defining the Landscape

Two concerns frequently arise when discussing exploratory testing from the standpoint of the person who isn't using it right now in his own work. First, there are concerns about what will be tested. How can we know what we've covered and what we've not covered when the testers decide where and how to test while they are sitting with the application under test? Second, there are concerns about the transferability of that knowledge. What happens when the exploratory tester becomes unavailable for further testing efforts on this feature or product? Both of these concerns can be addressed through the utilization of testing tours.

What is a testing tour but a description of *what* and *how* to test? By taking a well-defined tour, for instance the *Supermodel tour,* and pointing your defect observer (the tester) at a particular feature or product, you can know that a test pass has been scheduled to review said feature or product to survey the state of fit and finish class defects. Although the tester may, under his or her own initiative, find other kinds of defects, the entire testing act has been biased to detect those defects that make the product look bad.

Where heavily scripted testing may specify precisely which defects to confirm or deny the existence of in a given product, the tour specifies a class of behavior or defect and leaves it to the tester to determine the best route to confirming or denying the existence. This leaves management free to set the testing strategy, without impairing the tester's ability to choose tactics suitable to the feature or session of exploration.

The second question is perhaps a more difficult question for test as a discipline and not just exploratory testing. As testers gain experience, they improve their understanding of software systems, how they are typically constructed, and how they typically fail. This has the advantage of making experienced testers more efficient at finding typical software defects, or even defects typical of a given piece of software through the various stages of development, but this increased value makes the temporary or permanent loss all the more damaging to the overall test process. Testers will become unavailable throughout the course of development: They can take time off, be transferred to another project or team, change their role within the organization, leave the company entirely, or any number of other things

may arise that conspire to deprive your test effort of its most experienced testers. This is where touring really shines, because it defines *what* and *how,* such that any rational agent should be able to perform a similar tour and detect similar defects. Tours will not help testers detect subtle changes in typesetting, but tours can inform a tester that they ought to be concentrating on that sort of defect as they traverse the features they've been assigned.

A successful tour is one that intentionally reveals a particular class of defect, providing enough detail to bias the detection, but not so much as to narrow the focus too tightly either in feature scope or defect class. As such, it becomes even more important to record side trips and ancillary tours that were spawned by the scheduled tour. It should not be uncommon for your testers to pass up bug-finding opportunities but note them for further study when they can dedicate their focus to this unplanned area of interest. Touring, therefore, instills some discipline on the testing process. No matter if you choose to take the side trip now and resume the tour later, or mark the POI on your map and return after your tour, by making the distinction between on- and off-tour, one can come along later and understand which combination of tours should reveal all the defects observed. It is at this point when a decision can be made to either expand the set of regularly scheduled tours, or to run with the normal set and track whether these side-trip opportunities present themselves to other testers in the future.

Expanding the set may be helpful if you have a wide product and need to split work between new resources. By expanding the set, one is withholding some of the decision-making process from those doing the testing, but this can be critically important to avoid overlap and redundancy in a testing effort.

Tracking the recurrence of side trips in the future makes sense if you will have several opportunities to test different versions of the same application or same kind of application. In the event that a specific side trip does not present itself, it can be assigned as a special tour just to ensure that there really were not any such significant defects present in this testing effort. The subsequent sections walk through the use of tools through various points of an application development life cycle.

Planning with Tours

Before setting foot in a new city, prepared travelers will have acquired some basic information about their destination. It's good to know such things as what language is spoken there, if they will accept your currency, and whether they treat foreigners kindly, before you find yourself in unfamiliar surroundings with no plan in place to mitigate the challenges you'll encounter. This might mean taking one of the survey tours (*Landmark* or *Parking Lot*) yourself to get an overview of the key features you'll be expected to report on throughout the development and maintenance of the

application. At the end of the cycle, you'll want to be able to report any show-stopper defects as well as how wide the happy path for users actually is through your application. This means providing tours aimed at covering well the core scenarios and providing opportunistic tours with great breadth across the application to pick up abnormalities in out-of-the-way corners of the application.

Many tours fit into the start or end of a development cycle naturally. A tour capable of detecting improper implementation of controls, such as a *Taxicab tour,* should come earlier than one focused on fit and finish, such as the *Supermodel tour.* Although the individual testers needn't have any privileged information about the data or class structure of the application under test, they are still capable of revealing defects that are sensitive to these structures. It is therefore important to find systematic errors early, such as misuse of a classes or controls, before the application has had a chance to solidify around this peculiar behavior and fixing the defect becomes too risky, or time-consuming, or the repercussions of the change too poorly understood to undertake a fix.

Early-cycle objectives include the following:

- Find design defects early.
- Find misuse of controls.
- Find misuse of UI/usability.

These objectives lend themselves to *tours of intent:* those explorations that focus on getting something done rather than doing that thing in any particular way. Tours of intent include the *Landmark tour* and the *Taxicab tour.* At the start of the cycle, a testing effort will usually attempt to identify big problems.

Late-cycle objectives include the following:

- Ensure public functions function.
- Ensure user data is secure.
- Ensure fit and finish is up to expectation.
- Characterize the width of the functional feature path.
- Confirm previous defects are not observable.

These objectives lend themselves to *tours of specifics:* those explorations that focus on a particular something in a particular way. Tours of specifics include the *Back Alley tour, Garbage Collector's tour, Supermodel tour,* and *Rained-Out tour.*

Permutations on tours should be intentional and planned ahead of time. Thinking about taking a *Landmark tour,* but only using the mouse to navigate your application? Make that decision up front.

At the end of the day, one must still plan and martial testing resources effectively from the start to secure the opportunity to succeed for one's team, regardless of the testing techniques employed. Often, realizing that a

delay exists between development and test, test will undertake a testing effort over parts of the application in isolation rather than biting off an incomplete end-to-end scenario. Even when the application is nearly complete, when it is handed over to test as a full unit, it makes sense to begin with the early-cycle tours because they help identify opportunities for more in-depth tours and allocation of testing resources.

Letting the Tours Run

While exploration and permutation are important parts of the exploratory testing process, it is important that the tour and the tour guide stay on target for the duration of the tour. Staying on target means sticking within the intent of the tour. For example, if your current tour is scheduled to hit *N* major features, make sure you cover those *N* major features. This advice applies not just to the tester, but also to the test manager: Letting the planned tour run is important to identify the various side trips and follow-up tours that will help guide the next iteration of testing. Tours are meant to be repeated, by different testers, with different permutations in focus.

Because each tour is meant to be short, comprising a session of a few hours, there isn't much to gain from interrupting the tour. There is, however, much to lose from interrupting a tour in the way of knowing what you were looking for during that session. Testers will find bugs in products. This is not a surprise and isn't anything out of the ordinary. Figuring out how and why they found that bug might be something to talk about. With a guided tour, you have a framework under which another tester may be able to find similar defects. This is special and should be leveraged in subsequent tour assignments. When testers go off-tour, for whatever reason, they may find defects, but we already expect them to find defects on the tour they were experiencing. By leaving the tour, they will have lost the focus and observational bias granted by that tour, which makes scheduling a sensible follow-up tour much more difficult. A tester who chases down a promising side trip of "good bugs" might not get through the balance of the tour, leaving an even larger risk area in a completely unknown state. Without the confidence to come back to an area despite knowing bugs are ready to be found right now, you'll put yourself in a state of not knowing what you don't know at the end of the tour.

Analysis of Tour Results

Because tours impart a bias to the observer (for example, the tester is interested in specific kinds of observations with respect to the features comprising the tour), it provides great information about both the toured portion of the software and the need for additional tours through those parts of the

software. Touring testers will report both opportunities for side trips and defects that were strictly "off-tour" but prominent enough to stand out despite the testing focus elsewhere. This provides several opportunities to involve the testers, helping them to take ownership in the entire process. The opportunities for side trips are clear and can lead directly to specific in-depth tours or additional broad tours in this previously missed feature area. Reports of bugs that reside outside the focus of the tour are indicators that a tour similar to the last but with a focus more attuned to the unexpected bug detected should be performed.

Finally, when multiple testers have taken the same tour, some overlap will occur as to the bugs they report. Because the tour is providing a bias to the act of bug detection, the overlap, or lack thereof, should provide some insight into how many of those kinds of bug remain undetected in the product—or if further tours in this area should be scheduled until such time as the reported bugs from different tourists converge. Assigning multiple testers to the same tour is not actually a waste of resources (in contrast to assigning multiple testers to the same scripted test case, which is a waste of resources). Because the tour leaves many of the tactical decisions of test to the individual, variation will exist between testers despite the similarity in the testing they have performed. Although this might run counter to the claim that a tour will improve reproducibility in a test pass across testers and time, so long as the tour reveals a particular kind of defect with great regularity it is not incompatible with the discovery of different bugs (for example, a variety of manifestations will point back to the same root cause of failure).

Making the Call: Milestone/Release

When the time comes to report on product quality, a tour-based testing effort will be able to report on what works in the product and how well it works. A tour-based testing effort will be able to report on what fraction of the planned work actually works, in addition to how wide the path to each working feature actually is. From a high level, test will be able to report how likely additional bugs will exist in given features, remaining to be discovered. Likewise, they will be able to report how unlikely certain types of defects will occur along the tour paths taken during testing. While the number of bugs detected per hour might not differ from other testing methods, the tour-based effort should be able to prioritize which types of bugs to find first, and thus provide early detection of assumed risks.

In Practice

Your circumstances may dictate much of your strategy for breaking down work and running a tour-based testing effort. Some test teams may be involved during the feature design stages and provide input on usability

and testability of features before they are implemented, or your team might have no direct contact with the developers, only interacting with "complete" projects thrown over the wall in your general direction.

No matter what deadline you are under, you will want to start with a wider or overview tour to help identify where opportunities for in-depth exploration reside and specific tours to schedule. The feedback loop between these first explorations and the second wave is the most critical, as this is where you will begin to identify the overall allocation of testing resource to the product you have in front of you. Therefore, it is important to have great discipline early in sticking to the tour and identifying places to return with subsequent investigations. First, map the space for which you are responsible, and then go about focusing on particular points of interest or hot spots of trouble.

Conclusion

The touring concepts applied at Microsoft have helped software testers to better organize their approach to manual testing and to be more consistent, prescriptive, and purposeful. All the testers involved in these and other case studies have found the metaphor useful and an excellent way to document and communicate test techniques. They are now focused less on individual test cases and more on the higher-level concepts of test design and technique.

Exercises

1. Pick any bug described in this chapter and name a tour that would find it. Is there a tour, other than the one cited by the author in this chapter, that would also find the bug?
2. Pick any two bugs in this chapter at random and compare and contrast the bugs. Which one is more important in your mind? Base your argument on how you think the bug would impact a user who stumbles across it.
3. The *Supermodel tour* has been cited as a good tour for UI testing. Can you describe a way in which this tour might be used to test an API or other such software that has little or no visible UI?