Go with the Flow

by Michael Bolton

As Cem Kaner pointed out in a recent talk, when Glenford Myers wrote his seminal book *The Art of Software Testing*, computing was much simpler. Programs were smaller and slower, used character-based interfaces, had only thousands of lines of code, and contained far fewer variables. Today, our programs have millions of lines of code, thousands of variables, and extensive interaction with a multitasking, graphical operating system (see the StickyNotes for more on this topic). Most importantly, today's programs are object-oriented, event-driven, and interactive, rather than batch-driven. The flow of a program can be interrupted at any time, not only by the operating system but also by the user who might invoke any number of features, functions, or services. Paths and sequences of actions are less predictable than ever before. Moreover, our programs depend on mountains of application frameworks, third-party libraries, and operating system code. How can testers keep up? Flow tests help us deal with this new reality.

Traditional test design often is based on a *focusing strategy*: start from a known, clean state; prefer simple, *deterministic actions*; trace test steps to a specified model, such as a use case; follow *established, consistent design procedures*; make *specific predictions*, *observations*, and *records*; and make the test *easy to reproduce,* perhaps even automate it. Tests based on use cases emphasize focusing heuristics. These tests target some specific behavior. They identify an actor—a person or system that interacts with the product—and include pre-conditions and post-conditions to tell us what we need to start with and how things should look when we're done. This approach works for relatively simple testing tasks, where tests require high integrity, where we can anticipate local symptoms of local problems, or when we're in a *confirmatory* process, demonstrating that the system can work.

For more complex tests, a focusing strategy becomes more expensive and potentially more risky, and might blind us to problems outside of our immediate field of view. If we've been using focusing heuristics, it might be important to diversify and use *defocusing heuristics* in test procedures: *don't* start from a known, clean state; prefer complex tasks and actions that *aren't* predefined; *don't* constrain the test to a specific model; question, vary, and even violate standard procedures; *look at the big picture*, testing without a specific observational mode or predicted outcome in mind; and make the test *harder to pass* while relaxing the emphasis on reproducibility. For these purposes, flow testing often fills the bill.

In the Heuristic Test Strategy Model (see the StickyNotes for more information), "flow testing" means "one thing after another after another…" Tests might follow a specific path through the program's functions, tracking a piece of data from cradle to grave, or long sequences of actions without resetting the system. This approach tends to be *investigative*, seeking circumstances in which the system might fail, motivated by a search for nonlocal effects with respect to any given cause. Flow tests may use both scripted and exploratory aspects.

In the agile world, builds of the product are intentionally frequent to give developers quick feedback on the effectiveness and impact of a change, so unit tests often (and appropriately) are designed to emphasize speed over breadth and depth. After we've integrated elements of the product, we do flow tests to expose problems at the interfaces between them or to identify data corruption that occurs over time. Longer sequence tests—complex scenarios or a long series of short tests—model real-world conditions in a way that shorter unit tests and integration tests may not.

Programs, functions, and objects typically start with variables in initial states that eventually change. At least two kinds of errors can come from this—the initial state may be wrong, or some desired change may not occur. However, these errors may not become apparent until some later time, when the program tries to read or display the variable. Shared or global variables can be more vulnerable than other kinds of variables because they can be (mis)accessed by any function at any time. If there is a problem that happens at random or unpredictable intervals, then, generally, the longer the program has been running, the greater the odds are of the accident happening. Some might argue "our developers would never use global variables," but we can neither

"Good end-to-end testing depends on breadth, depth, variability of actions and data, and rich scenarios, both plausible and implausible."

know nor control if someone else does—and our code inevitably interacts with third-party libraries, device drivers, and the operating system where vulnerabilities aren't visible to us. We can't assume that other people are living (or coding) up to our standards.

Long-sequence, randomized, high-volume, automated tests can be particularly good for shaking out reliability problems, timing issues, resource contention, and performance defects. All tests require oracles—trustworthy (albeit sometimes fallible) means of recognizing problems. High-volume automated tests in particular need high-speed oracles, ideally provided by comparable but not identical algorithms. In long-running tests, finding the cause of a problem can be a challenge, so these tests also need traceability, typically provided through detailed logs that not only can be read and understood easily by humans but that also can be stored, accessed, and parsed easily by tools and regular expressions. Log files and scriptable interfaces to the product make test automation much easier. Rapid testers ask early and often for testability.

When I'm testing an application interactively, I might test flow by performing many small, atomic tests without resetting or closing the application. I also might try to perform longer tasks, incorporating some kind of unusual, erratic, or unpredictable behavior. I model the user by changing my mind, backtracking, trying to do things that are outside of some notion of "normal" sequence. My emphasis isn't on the volume of data that I can drive through the system; I let automation handle that. Instead, I'm trying to make new observations, learn more about the program, and improve my test design. This means defocusing, surveying the screen, welcoming productive distraction from rote behavior, looking for new tasks, and inventing new ways to do routine tasks. All of these things require a fundamentally exploratory approach.

I try to make intentional mistakes and try to pay attention to my unintentional ones. A particularly strong focus for human-performed flow testing is watching for error-message hangover—failure to clean up after some exceptional condition. "Error" and "exception" mean "something that isn't supposed to happen (very often)." Consequently, it's easy for requirements analysts, designers, and developers to give error conditions a psychological brush-off and to pay minimal attention to handling the problem.

As in all tests, steps in end-to-end system tests can be performed by machines, humans, or some combination of the two. The object is to follow the flow of individual transactions and their elements from soup to nuts, adding power to the test by combining increasingly complex stories that interact with all of the system's components. An end-to-end test for a retail organization might include creating shopping carts that generate sales orders that produce inventory movements that trigger purchase orders that debit and credit several accounts, while accounting for warranty status, updating appropriate databases, receiving items from wholesalers at one regional warehouse and moving items to another, delivering purchased items to customers' homes, having some customer reject delivery because goods were damaged in transit, delivering replacement items from existing inventory, and matching the customer's online warranty registration to the shipped product (the replacement, not the damaged one).

Automation focus tends to be on volume and performance; the human focus is directed toward developing the scenario and careful observation of the interfaces, the results and the system state, using tools and log files to observe things that might otherwise be invisible. Good end-to-end testing depends on breadth, depth, variability of actions and data, and rich scenarios, both plausible and implausible. It also depends on good models that capture the essentials and exceptions in the business process.

Simplicity in testing is a worthy goal. Flow tests help to address the fact that our applications run continuously in a messy, complex, and human world. {end}

Michael Bolton lives in Toronto and teaches heuristics and exploratory testing in Canada, the United States, and other countries. He is a regular contributor to Better Software *magazine. Contact Michael at mb@developsense.com.*

STAR WEST SPEAKER

Sticky Notes

For more on the following topics go to www.StickyMinds.com/bettersoftware.

- Cem Kaner presentation
- Heuristic Test Strategy Model

Sometimes we see problems where all the pieces are right, but somehow they don't fit when we put them together. Got a story about that?

Follow the link on the StickyMinds.com homepage to join the conversation.

Want to learn more about a great idea or technique in this issue of *Better Software* magazine? The **StickyMinds.com** Discussion Boards provide a great forum for sharing ideas and conversing with other StickyMinded people worldwide! Ask questions. Give pointers. Exchange messages with your colleagues in the software testing and quality engineering world. Or sit back and read what your peers are saying about a variety of subjects important to software development at **www.stickyminds.com/forum.asp**.