Chapter VI
Agile Quality Assurance Techniques for GUI-Based Applications

Atif Memon
University of Maryland, USA

Qing Xie
Accenture Technology Labs, USA

ABSTRACT

This chapter motivates the need for new agile model-based testing mechanisms that can keep pace with agile software development/evolution. A new concentric loop-based technique, which effectively utilizes resources during iterative development, is presented. The tightest loop is called crash testing, which operates on each code check-in of the software. The second loop is called smoke testing, which operates on each day’s build. The third and outermost loop is called the “comprehensive testing” loop, which is executed after a major version of the software is available. Because rapid testing of software with a graphical-user interface (GUI) front-end is extremely complex, and GUI development lends itself well to agile processes, the GUI domain is used throughout the chapter as an example. The GUI model used to realize the concentric-loop technique is described in detail.

INTRODUCTION

Agile software development has had a significant impact on the development of software applications that contain a graphical user interface (GUI). GUIs are by far the most popular means used to interact with today’s software. A GUI uses one or more metaphors for objects familiar in real life such as buttons, menus, a desktop, the view through a window, a trash can, and the physical layout in a room. Objects of a GUI include elements such as windows, pull-down menus, buttons, scroll bars, iconic images, and wizards. A software user performs events to interact with the GUI, manipulating GUI objects as one would real objects. For example, dragging an item, discarding an object by dropping it in a trash can, and selecting items from a menu are all familiar events...
available in today’s GUI. These events may cause changes to the state of the software that may be reflected by a change in the appearance of one or more GUI objects.

Recognizing the importance of GUIs, software developers are dedicating an increasingly large portion of software code to implementing GUIs—up to 60% of the total software code (Mahajan & Shneiderman, 1996; Myers, 1993a, 1995a, 1995b; Myers & Olsen, 1994). The widespread use of GUIs is leading to the construction of increasingly complex GUIs. Their use in safety-critical systems is also growing (Wick, Shehad, & Hajare, 1993).

GUI-based applications lend themselves to the core practices of agile development, namely simple planning, short iteration, and frequent customer feedback. GUI developers work closely with customers iteratively enhancing the GUI via feedback. Although agile processes apply perfectly to GUI software, integration testing of the GUI for overall functional correctness remains complex, resource intensive, and ad hoc. Consequently, GUI software remains largely untested during the iterative development cycle. Adequately testing a GUI is required to help ensure the safety, robustness, and usability of an entire software system (Myers, Hollan, & Cruz, 1996). Testing is, in general, labor and resource intensive, accounting for 50-60% of the total cost of software development (Gray, 2003; Perry, 1995).

GUI testing is especially difficult today because GUIs have characteristics different from those of traditional software, and thus, techniques typically applied to software testing are not adequate. Testing the correctness of a GUI is difficult for a number of reasons. First of all, the space of possible interactions with a GUI is enormous, in that each sequence of GUI events can result in a different state, and each GUI event may need to be evaluated in all of these states (Memon, Pollack, & Soffa, 1999, 2000b). The large number of possible states results in a large number of input permutations (White, 1996) requiring extensive testing. A related problem is to determine the coverage of a set of test cases (Memon, Soffa, & Pollack, 2001c). For conventional software, coverage is measured using the amount and type of underlying code exercised. These measures do not work well for GUI testing because what matters is not only how much of the code is tested, but in how many different possible states of the software each piece of code is tested. An important aspect of GUI testing is verification of its state at each step of test case execution (Memon, Pollack, & Soffa, 2000a). An incorrect GUI state can lead to an unexpected screen, making further execution of the test case useless since events in the test case may not match the corresponding GUI elements on the screen. Thus, the execution of the test case must be terminated as soon as an error is detected. Also, if verification checks are not inserted at each step, it may become difficult to identify the actual cause of the error. Finally, regression testing presents special challenges for GUIs because the input-output mapping does not remain constant across successive versions of the software (Memon & Soffa, 2003e; Myers, Olsen, & Bonar, 1993b). Regression testing is especially important for GUIs since GUI development typically uses an agile model (Kaddah, 1993; Kaster, 1991; Mulligan, Altom, & Simkin, 1991; Nielsen, 1993).

The most common way to test a GUI is to wait until the iterative development has ended and the GUI has “stabilized.” Testers then use capture/replay tools (Hicinbothom & Zachary, 1993) such as WinRunner (http://mercuryinteractive.com) (Memon, 2003a) to test the new major GUI version release. A tester uses these tools in two phases: a capture and then a replay phase. During the capture phase, a tester manually interacts with the GUI being tested, performing events. The tool records the interactions; the tester also visually “asserts” that a part of the GUI’s response/state be stored with the test case as “expected output” (Memon & Xie, 2004c; Memon, Banerjee, & Nagarajan, 2003d). The recorded test cases are