Testing Smalltalk Applications

Report by Barbara Yates, Bytesmiths

Many of the problems encountered in testing Smalltalk applications are similar to the problems encountered in testing any software system: how to encourage (managerially) an adequate testing process, how to define test plans and test cases, how to test the tests, etc. One major difference in testing a Smalltalk application, however, is the role the development environment plays in the testing process. On one hand, the development environment fosters early and continuous testing, and eases the development of tools that support testing. On the other hand, developers tend to write code and add features more quickly (sometimes even changing the compiler/run-time environment) which complicates the introduction of a testing process that adequately balances the cost of testing with the cost of releasing a product.

Are all Smalltalk testing problems, then, managerial problems that could be solved by the introduction of suitable processes? The answer seems to be no. A greater understanding of the types of problems arising at different stages of the development process that can be used to develop tools and code inspection checklists are needed. More understanding of the role of coverage tools and templates for defining interfaces/APIs (and hence test plans) are also required. A need for more general tools such as GUI testing tools, problem reporting systems, etc., was also identified.

Workshop Organizers

Barbara Yates, Bytesmiths
<barbara.bytesmiths@acm.org>
Jan Steinman, Bytesmiths <jan.bytesmiths@acm.org>
Gail Murphy, Univ. of Washington
<gmurphy@cs.washington.edu>
Mark Murphy, Consultant
<71202.2241@compuserve.com>
Gray Huggins, Texas Instruments <huggins@works.ti.com>

Motivators

Many organizations are jumping on the Smalltalk bandwagon, hoping to reap the many benefits ascribed to OOT and Smalltalk in particular. Those who have already worked hard to bring a product written in Smalltalk to successful release have been forced to come up with their own solutions for how to assure the quality of their product.

The scarcity of literature (particularly how-to information) and tools to support testing is disturbing, especially to organizations beginning a Smalltalk project. Our goals for organizing the workshop were to: (1) Learn about the current state of the art in testing Smalltalk applications. (2) Share what we've done on our projects (process, tools, classes).

Process we used

Participants submitted position papers on the themes of tools, testing strategies, management buy-in, GUI testing, multi-platform testing, how you handle third-party or reused classes/frameworks with respect to testing, fulfilling testing requirements of internal and external standards bodies and integrating those requirements into the OO development process. In advance of the conference, each participant was responsible for reading the papers posted on the workshop's web page. Timing is everything. We timed every part of the agenda, and when we slipped and ran a bit late we compensated later and were able to accomplish all of our goals for the day. To fit in as much discussion and idea-generation as possible, participants were asked to prepare slides showing their main points, and were limited to five minutes each to present their slides. Papers were grouped by subject, and each group of papers was presented like a panel, with all questions reserved until the full panel had presented. Each "panel" was followed by a session of what we called "list work". One of the organizers

acted as moderator while two scribes captured what the participants felt were strengths or "good things" and weaknesses or "bad things" about what the presenters had just discussed. These lists were hung on the walls of the workshop room until we ultimately had papered the room and the doors. In the afternoon we distilled the ideas we had written on the lists in the morning and came up with four discussion groups. We spent some time in the afternoon discussing costly bugs we had found on our projects and what we could learn from that experience. Three of the participants gave live demos of tools that they had developed which aid in the testing process. These demos were of the Bytesmiths' Task Management (defect tracking) tools, the MCG Software's OTF (An Object Testing Framework), and the internal tools used by IBM to test VisualAge and IBM Smalltalk.

Highlights from the panels

The position papers were distilled into a few slides each and presented in panel groups as follows: Process, Experience Reports; What Smalltalk Testing Means; Theory, Techniques, Patterns; and Frameworks and Tools. These are some of the points raised during the panel presentations.

- Role of "bugfix checker": reviews all code changes made to fix a bug, and 20-30% of the time they find a possible problem with the changes. The checker is not the submitter.
- No third-party tool is adequate for automated GUI testing on all platforms on which the product runs. Instead of automated tests, test scripts are followed.
- Relative amount of developer time spent on testing is 40-50% during phase in which test cases are eing written; testing drops to 20% of the time when the product is in maintenance.
- "You can't test without specs" — many presenters mentioned lack of written specifications as a major problem.
- There was general agreement that interfaces need to be tested, with emphasis the "the contract's the thing!"
- Use of test coverage analyzers is recommended.
- Use cases can form the basis for test cases.
- Use a tool to perform static analysis of the code to find complexity, then target testing at the complex parts.
- Use of assertions was recommended by one presenter, but it is necessary that the assertions be easy to remove.
- Some Smalltalk dialects do not include compile-time or batch modes of determining messages sent but not implemented, so some participants have developed checkers to do such tests. Another sort of check that is performed is type-inference for messages sent to instance variables.
- Smalltalk developers do a tremendous amount of informal testing. There seems to be a need to capture the informal tests that developers write. Often "formalized" testing takes place too late. Test suites, like source code, need to be managed, versioned, and controlled.
- The single largest factor in testing is not testing, but management. (This sentiment was echoed by all participants.)
- Performing cluster and class tests cuts the number of bugs that are found later.
- It is worthwhile to classify the patterns used in testing. There will be a book published soon on this subject.
- The Smalltalk archives at UIUC will soon contain a "Style Checker" tool that detects classic Smalltalk bugs and points out possible errors according to some rules about error-prone coding styles.
- For internal use, the ParcPlace Portland office developed a TestCoverage Analyzer and TestWorks toolset. Unfortunately, there are no plans to productize these tools.

- The IBM folks have developed in-house tools to aid in testing VisualAge, IBM Smalltalk, and third-party developed components. These are prototype tools, with no announced plans to productize them.
- Model vs. GUI Testing — participants disagreed about their relative importance. Bottom line is that you must test both, the problem is determining what types and coverage of testing will reduce risks of software failure to an acceptable level.

Conclusions from costly bugs discussion

All participants were asked to be prepared to describe a costly bug from their own Smalltalk projects. About half of the participants talked about their costly bugs. These are some of the lessons we took from their descriptions:

1. Be very careful when tinkering with VMs and third-party code.
2. Integrate frequently and use incremental development.
3. Don't be blinded by technology; keep tabs on the process.
4. Performance tune only when needed.
5. Don't ship prototypes.
6. Have other people inspect problems.
7. Specification bugs are still bugs (use detailed specs/use cases).

Group discussions

After lunch we chose four "hot topics" from the many items that were captured in our morning sessions of list work and separated into groups to discuss them. A summary of those discussions follows.

Management issues

Experience has been that when schedule pressures mount, often it is the testing that is short-changed. Kevin Haaland told us that at OTI, any developer can say "we can't ship this." They strive for consensus. One participant observed that it is often marketing who decides when the product is ready to ship. They are eager for the immediate cash-flow, and will pay what they must for maintenance later. One concern developers have is that their perception of the priorities of bugs do not match their customers' priorities. A participant said that although there is a lot of lip service paid to testing, he finds that he has to "sneak quality in." Developers versus external QA groups were discussed. At OTI the developers do QA. New hires run test cases to help them become familiar with the code. They also act as naive users and find a lot of GUI bugs. An estimate is that OTI gets 60% coverage of their GUI by following test scripts. When OTI gets a customer-reported bug, they immediately write a use case for it and an automated test. One developer writes the use cases and test cases, another implements the feature. OTI prefers to cut features to allow sufficient testing time. The group discussed what should be in the image under test, both during end-of-cycle incremental development and at system test time. It is important to test the runtime, productized image. The image should contain only the product plus the test cases.

Conclusions

Who should decide when the product is ready for release, marketing or engineering? We lean toward a consensus, with clear criteria for judging release-readiness. Each incremental development cycle has a testing phase which should use as close to the delivery environment as possible. Test plan must be written early, and it must be based upon written specifications. There is a need for both developers to test and for a separate QA group for the system testing. The whole team follows the product to the end.

Black box, white box, gray box?

As the discussion progressed we agreed upon definitions of the terms. Black box testing (BBT) is interface based; it assumes no knowledge of nor access to the implementation. Gray box testing (GBT) is also interface based but it uses knowledge of the implementation (no access to implementation's private state). White box testing (WBT) can access private state and makes use of implementation knowledge.

You need stable interfaces in order to build tests. BBT is easier to maintain because it tests stable interfaces. Results of coverage analysis can be fed back into the

test development process. While we sort of called these GBT, this is just a way of thinking about feeding implementation information into the process of developing tests — they are still BBT. New releases of the system under test, that support the same API, will not invalidate BBT — however they may invalidate coverage assumptions — coverage needs to be rechecked.

The question was raised as to the adequacy of BBT. They can be inadequate because certain important behaviors are often not captured in the API specifications. As these behaviors are discovered, this information should be fed back into the API specification.

WBT are to be considered a power tool — you get a lot of leverage per amount of code, as compared to BBT, but over time you may find WBT are too expensive to maintain. In fact, because of the pressures of release cycles, they may not get maintained at all, making them obsolete. BBT/GBT are more likely to still be useful, even if they are not maintained.

Conclusions

Start out with a full suite of BBT based on interfaces, analyze coverage, tweak/add more BBT to get good coverage, and use WBT where it is impractical to fully test with a BBT interface approach. As general goals, try to have most of the test cases be BBT because the maintenance costs are lower.

Developing a taxonomy of bugs

We aren't satisfied with the current state of the art in testing Smalltalk user interfaces. We are also aware that we need to improve our stress-testing of Smalltalk applications. There is a definite need for style guides and good test procedures.

This group drafted a list of commonly detected Smalltalk bugs. It was agreed that having such a list could improve the quality of Smalltalk code in several ways: code reviewers could look for these common bugs when doing their code-reading, and automated checks performed by tools such as the Style Checker developed by John Brant could look for some of the common bugs. The list started with the bugs that the Style Checker already checks, and then the group members added to it, dividing the bugs into broad categories of model and user interface defects. Here are the lists they developed.

Model

1. Typographical errors
2. Syntax errors
3. Failure to initialize
4. Type errors
5. Inverse-Bug. Don't fix the bug, fix around the bug
6. Misuse of exceptions
7. Failure to separate model from GUI code
8. Misuse / confusion of #super and #self
9. Double initialization
10. Duplicate execution
11. Forgetting to return value
12. Code depending on result of an #add:
13. Modifying a collection while iterating it
14. Forgetting to remove <self halt>
15. Accidental overriding of special methods (e.g., #class)
16. When allowing VisualWorks to declare temporary variables, mistakenly causing an instance variable to be used when it should be a method temporary variable.
17. Get ifTrue:[ ] wrong way around: when ifFalse:[] should have been used instead
18. Improper scoping of block temporaries
19. Creation of objects in a loop
20. Global references to objects: causes memory leaks
21. State based errors (e.g., should keep code sequence free where ever possible)
22. Failure to kill caches
23. Copy replication issues
24. Usage of data structures (e.g., arrays) instead of objects
25. Modification of literal arrays
26. Assignment of a literal to a class (e.g., Array := 1)
27. Omission of #yourself in cascading
28. Failure to call <super initialize> in an #initialize method
29. Inconsistent method behavior
30. Incorrect control structures. Examples:

[1 to: aCollection size] do: [....].
(x < y) whileTrue: [....]
[x isBig] ifTrue: [....].

31. Failure to implement a #hash method when an equality operator is overridden
32. Assuming that an accessor method answers the original version of a collection: it may be a copy.
33. Usage of an explicit class name, rather than <self class> in a method
34. Collection bound errors with fixed size collections
35. One off errors

User interface

1. Obtaining extra windows when opening a dialog too quickly
2. Tabbing: incorrect order, absence of, changing of sequence
3. No formatting of fields
4. Failure to specify "hot" keys
5. Not adequately managing screen resolution, which can lead to windows which are too large, or misplacing of fields and labels
6. Incorrect spelling of fields
7. Incorrect casing of fields
8. Inconsistent usage of labels on top or before fields
9. Incorrect usage of fonts
10. Inconsistent usage of colors
11. Too many widgets on window
12. Aesthetic issues
13. Failure to correctly grey out or disable fields, buttons, or menus
14. Damage painting or refreshing
15. "Cheese": Dropping pixels on the screen
16. VisualWorks scrolling problems
17. Incorrect updating of panes
18. Refresh problems: too many, too few
19. Failure to call superclass #close method

What is the role of coverage testing?

This discussion began by centering on coverage analysis tools and their benefits, and evolved to include the testing process in general. Coverage analysis is a tool for measuring code coverage by tests and identifying deficiencies. The results can help with risk reduction. It does not verify test completeness. It should be used as an adjunct to other testing techniques. It does not verify pre/post conditions.

The group then moved onto the quality of the test themselves. How can we best verify the tests? Some testing is done by executing tests written in Smalltalk. Other testing involves manually following the steps in a test script. Inspections and reviews of tests must take place regardless of the medium. But one can't tell if tests are good without a specification.

Conclusions

Regardless of whether tests are written in Smalltalk, are scripts, or something else, the process of validating the test must take place. Maintenance, training of the testers, and other issues have a bearing on the choice of Smalltalk code or test scripts.

Results of list work

At the end of the day we synthesized the many items on our lists into two short lists. The areas we all agreed need the most improvement in Smalltalk testing are:

- Lack of process: The newness of OO and the changes to the development process appear to negatively impact testing. Testing is not being given sufficient attention.
- Lack of discipline: Managers do not appear to be enforcing their quality assurance process (when they have one). There also must be specifications; and test plans must be written and reviewed.
- There is no standard way to specify interfaces (API). Without knowing the "contract", good tests of the interfaces (especially black box tests) cannot be written.
- Measurement: There is insufficient measurement of quality by projects. Lack of data makes it very difficult or impossible to develop quality metrics.
- Vendors don't perceive that the testing tool market is viable (they develop test tools and then keep them in-house).

The areas that we agreed are the strongest and biggest

assets for Smalltalk testing are:

- The development environment fosters early testing.
- The open development enivronment enables us to build tools to support testing.
- Refactoring improves our implementations. This makes black box testing and regression testing more important. At least some of the black box tests should still be valid after refactoring. Regression testing is needed to check the refactoring.
- Some organizations have found that assigning novices to test classes has provided the novices with a good introduction to the organization's code.

What next?

The participants would like to see this workshop repeated. We gave ourselves some tasks with that in mind:

- Collect data on our projects regarding the kinds of bugs we find. Also count the bugs per class, noting the size of the class (in lines of code and number of methods) and the number of message sends in the defective code.
- Try to create a WWW site to collect Smalltalkers' requirements for testing frameworks and tools. Also collect requirements for defect tracking tools.

At the next workshop on Smalltalk testing, we think the focus should include these topics:

- Recommendations for a set of tools and the requirements for them
- Quality metrics
- Good examples of testing process

Acknowledgements

We wish to thank Gail Murphy, Phil Haynes, and Dave Thomson for their invaluable assistance in writing this report. Also, we especially appreciate Gail Murphy's efforts in creating and maintaining the workshop's web page.

Participants

John Brant, University of Illinois
Donald G. Firesmith, Knowledge Systems Corp.
Mark Foulkrod, IBM
Steve Goodman, USF&G
Kenneth R. Greene, Siemens Power Corp.
Kevin Haaland, Object Technology International (OTI)
Philip Haynes, Object Oriented Pty Ltd.
Leo Hsu, Management Strategies
Jeff McKenna, MCG Software, Inc.
Regina Obe, Management Strategies
Roxie Rochat, UniSQL
Michael Silverstein, IBM
Dave Thomson, Object Technology International (OTI)
Jay P. VanSant, GemStone Systems
Charles Weir, Object Designers Ltd.

Web

The workshop position papers can be read at the following web location:
http://www.bytesmiths.com/pubs/95StTestingWorkshopPapers