Smalltalk SQA: What to test?

Jeff McKenna

As developers use Smalltalk more frequently to develop production systems, the quality of Smalltalk-based applications has become an important issue. This is the first of a series of articles on Smalltalk SQA, the goal of which is to discuss the issues of SQA in the Smalltalk development environment. Each article will focus on a specific aspect of Smalltalk SQA. Note that Software Quality Assurance in Smalltalk is not a solved problem. This series hopes to contribute to the solution.

Many of the points to be made in these articles apply to software developed using any development environment. However, the use of Smalltalk makes the impact of not "taking care of business" more visible because of the high development productivity. We need to develop processes and tools to help us develop robust software without sacrificing high productivity.

We must build the quality in.

THE TOP QUESTION

When considering SQA, the first question we ask is this: *What do we test*? The first answer is easy: *The software*. This answer is insufficient. For any real project there will be insufficient time or resources to *completely* test the software. This means that we must decide how to deploy those resources most effectively.

For applications with a GUI, the answer seems easy: *Test the interface*. The need for such testing is real and the success of GUI testing tools attest to that reality. The GUI testing tool vendors are working to make sure that their tools work with the dialects of Smalltalk. Many projects are using GUI testing as their sole method of testing.

This reminds me of quality control activity in the automotive industry in the 50s and 60s. TV ads from that time show hordes of white-coated, quality technicians poring over a car fresh off the assembly line. The technicians assured us that they would find every defect before it reached us.

Unfortunately, those technicians did not find enough of the defects. The automotive industry went through a very traumatic time over quality. The industry finally came to the same conclusion that other industries have. In short, it is *not* possible to insure sufficient quality with *only* inspections at the end of the manufacturing process. This is exactly what we are doing when we test only by using GUI testing tools.

In our experience, the GUI is the least stable part of an application. Typically, it is the last part finished. If we are to have the SQA work proceed with speed that we have come to expect in development, we must do more work in parallel. Basing all testing on the GUI means that little testing can occur until near the end of a development cycle. Since the GUI may change as well, GUI based tests will tend to be unstable.

GUI-based testing is necessary but it is not sufficient. If it was, software would be of much higher quality than we experience. In addition, when the software is "headless," GUI-based testing is not even possible. I am not saying here that GUI-based testing should not be done. If my resource constraints are so severe to limit my testing technique to a single one then GUI testing would be it.

The automotive industry solved its quality problems by utilizing a number of techniques. One common thread of these techniques is that they maintain quality at each stage of production. The automotive worker can stop the assembly line if cars of "bad" quality are being produced. Component vendors ship components of top quality, that is, no defects. We hear the result of these changes in the language of the advertising: "Quality is Job #1" or "Quality is built in."

"What to test?" is now a question that has a clearer answer. We must test the final product and the components that make up that product. We must build the quality in.

COMPONENTS

Before we can discuss how to build quality in, we must first establish our definition of a component. In automotive manufacturing, example components are the screw, the bolt, the starter and the transmission. In transmission manufacturing, example components are the gear, the switch, the screw and the bolt. Higher "level" components contain lower level components. Different higher level components often share the same lower level components. Both the starter and the transmission may use 1/4 6-32 screws and bolts. We see in the automotive industry two of the component constructs we find in our industry: Encapsulation and Aggregation. A bill of materials illustrates the use of these constructs.

Initially most of us felt that the class was the lowest level component in Smalltalk, the lowest level of reuse. We now know that the class is not typically the lowest level component. The lowest component is a combination of classes and methods with a clear purpose and intent. An example of multiclass capsulation into a component is the Graphics Component of many textbooks:

Graphic
  Ellipse
    Circle
  Rectangle
    Square
  Line

Taken together, these classes form a logical grouping. If the class Graphic implements a method Graphic>>isGraphic, we should also define Object>>isGraphic. If we do so, the Object method is part of the Graphics Component.

The major code control systems used with Smalltalk today acknowledge the levels of components under different names. Team/V calls low level components packages and higher level components clusters. ENVY does not map directly in this way. It calls low level components applications or subapplications depending on the reuse strategy and the high level components configurations or applications.

My view is that these are all components and that a single construct is sufficient. Low-level components just do not contain any subcomponents. ENVY could drop the distinction between application and subapplication while Team/V could merge cluster and packages.

The decomposition of an application into components is not an easy task and there is no right answer. Having too many is confusing and costly and having too few inhibits parallel development. The criteria to be considered in defining a component are very similar to those used in defining classes. Does the component do one thing? Is it maintainable by one person? Is it cohesive? Does it have a narrow interface?

SQA uses a component differently than development uses that component. We suggest that the component is the unit of acceptance and rejection. This moves SQA activities into the production process. These activities must determine the quality of each component.

SOFTWARE COMPONENT

We define a software component as code encapsulated with documentation and tests. The key concept is that the code itself is not verifiable or reusable without its documentation and its tests.

If this sounds very old, it is. As an industry we have *known* this for a long time. We have not acted on that knowledge. The question in my mind is how to build process and structure that encourages the practice. If both documentation and tests are an active part of the development then the practice is encouraged. In particular, components are reusable only when they include documentation and tests.

From the point of view of testing, the primary goal of documenting the software component is to present the public interface. This should include message definitions including the returned objects, required message sequences and error conditions. A major SQA task is to determine if the component tests cover the component public interface. The vehicle for this is the documentation.

My experience is that the process of *writing* the documentation improves the code, *if* the developer corrects errors, renames methods and fleshes out behavior as part of the documentation effort.

At a minimum, component tests must exercise the component public interface (verification). More complete testing will do a number of things: (1) test to determine if all code is executed (coverage); (2) test to determine if reaching limits does not fail (stress); (3) test the error handling (failure analysis); or (4) test the internal details (implementation). A primary SQA task is to make sure that the component tests are sufficient to determine if the component meets the project quality standards.

In the past, we often found the developers writing the lowest level tests, unit tests, and leave the fleshing out of tests to SQA, system tests. With Smalltalk that divide should be less clear. I view the SQA tasks as work to be done to delivery a quality software component. Who exactly performs those tasks is not too important.

My experience is when developers write and use tests, their code improves, exactly the same as when they write documentation. Testing will uncover errors. The availability of tests also makes incremental development work better. Developers will know when their rework is complete.

The automotive worker now knows that quality is an issue for him or her to address directly. Quality is built in. Development of software in Smalltalk needs to do the same.

In the next article, I will take up the public/private problem and discuss regression testing.

Jeff McKenna is Founder and President of MCG Software, Inc. of Wilsonville, OR. MCG Software offers testing frameworks for Smalltalk. Jeff has been involved with software for more than 33 years and been involved with Smalltalk since 1982. Jeff was chairman of OOPSLA '94. He may be reached at mckenna@acm.org.