

# Smalltalk SQA: The Public/Private Problem #2

Jeff McKenna

IN THE PREVIOUS article of this series, we opened up a discussion of testing issues when using Smalltalk. We discussed aspects of GUI and Model testing, and we provided a definition of a software component as code, documentation, and tests. In this article, we discuss the testing of these components in detail, including the role of regression testing.

In discussing components, it is very important that we include the ability to construct complex components as aggregations of simpler components, because this is very common and allows us to define components in a recursive manner. Our prior discussion of a software component emphasized the role of the interface in the definition of the component. The interface must be supplied for an aggregate or complex component, as well as a simple component. This brings us to a problem that I refer to as the Public/Private Problem.

In Figure 1, we show a simple diagram of three classes, A, B, and C, some numbered methods, 1 to 8, and two message sends from outside. Note that the method numbers are arbitrary.

Smalltalk currently defines the interface through the public/private "attribute" of the methods. In Figure 1, methods 7, 1, 4, 3 and 6 would be considered public if the two message sends were the only way the classes are used.

Now consider Figure 2. In this figure, we have constructed two components, I and II, from the supplied classes.

Note that from the point of view of component I, the interface is 1 and 6. From the point of view of component II, the interface is 6 and 7.

Let's look at Class A methods.

## Method 2

Class A Private  
Component I Private  
Component II Private

## Method 4

Class A Public >>A(4)  
Component I Private  
Component II Private

## Method 1

Class A Public >>A(1)  
Component I Public >>I.A(1)  
Component II Private

## Method 6

Class A Public >>A(6)  
Component I Public >>I.A(6)  
Component II Public >>II.A(6)

We are using a simple dot notation to indicate the intersection of the method and the component:

<component>.<class>(<selector>)

From this diagram it is easy to see that the public/private attribute of a method is not a useful construct in determining the interface of a component. Each component must define its own public methods (i.e., its interface). We have found that the identification of this interface is critical to the building of reusable components.

None of the currently available code control systems for Smalltalk support this view of the interface definition of a component. At most they support the public/private attribute of a method. This situation makes it difficult to adequately specify and test a component. I would like to encourage the vendors to add such support to their tools. Such support would move us a long way in the direction of being able to clearly define components.

Note that if the packaging changes, then the interfaces may change fairly dramatically. For example, if compo-

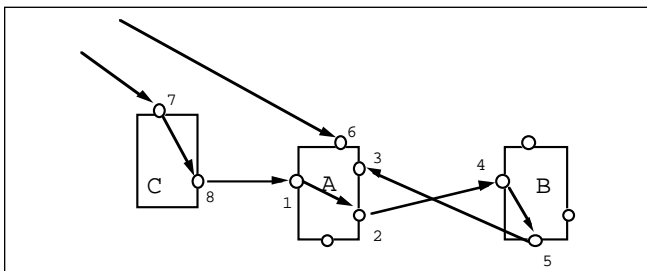


Figure 1.

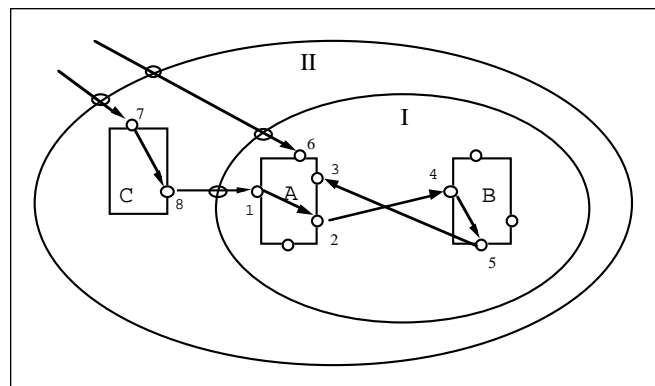


Figure 2.

ment I is changed to be classes A and C rather than classes A and B. As we would expect, the interfaces of II remain II.C(7) and II.A(6). The interface of I, however, changes noticeably. I.C(7) and I.A(4) are added to the interfaces, I.A(6) does not change its status, and I.A(1) is no longer part of the interface.

We can make one further observation regarding aggregate components. If a method defined in a component, C, is private, then it must remain private in any aggregate component that contains the component C. This restriction is often violated, as “interesting” methods are discovered deep within a component structure. One might consider such usage as behavior “leakage,” which is analogous to memory leakage. The tool support requested above would make it possible to detect such leakage.

This view of the public/private problem also solves the difficult problem of considering methods in a class hierarchy. If a component, CA, holds an abstract class, A, and another component, CS, holds a concrete subclass, S, then how do we handle methods that are defined in the class A but only used within the class S? Our view is that all such methods must be considered public to the component CA, just as any methods in S that are required in A to “complete” the abstract class should be considered public to the component CS.

All of this is a bit easier to think about if we just consider a class as a component consisting of a bunch of methods. Thinking of classes in this way also makes it easier to consider loose methods (class extensions).

## TESTING COMPONENTS

With this enhanced definition of a component and its interface, it is now possible to discuss how we might test a component.

Testing only the interface, with no knowledge of the internals, is called black box testing. Most practitioners consider black box testing insufficient because it is typically impossible to test all possible states. Testing with knowledge of the internals is called white box testing. A suggestion has been made that testing the interface with a “little” knowledge of the internals should be called gray box testing!

The decision as to the type of testing (i.e., black, gray, or black box) depends on the testing job being performed. Let us consider these three major tasks performed with tests:

- Unit Testing
- Interface Testing
- Aggregation Testing

Unit testing is typically performed by the developer and should verify that the component functions as designed. This usually means white box testing or, at least, gray box testing.

Interface testing is a term I use to mean testing only the interface. It is important that interface tests are provided to support consolidation and redesign activities. SQA should ensure that interface testing completely exercise the interface of the component.

Aggregation testing is the term that I use when testing aggregations or complex components. In aggregation testing, interface testing of the subcomponents is followed by the unit testing of the component itself. In other words, first test if the pieces still work and then determine if they are working together correctly; bottom-up testing as it were.

In our example, here is the testing sequence for component II:

Unit	Class A
Unit	Class B
Interface	Class A*
Interface	Class B*
Unit	Component I*
Unit	Class C
Interface	Component I**
Interface	Class C**
Unit	Component II**
Interface	Component II

The asterisk indicates the aggregation test of component I and double asterisks indicate the aggregation test of component II.

Of course, in practice, distinctions are never this clear. However, they should be considered when considering the efficacy of testing.

The above sequence works fine when fixing bugs and when adding functions. For each version, existing tests are used as is or are expanded to test for bugs and the new functions.

This is classic regression testing, which can be automated. Automation helps keep systems “no worse” than they were in the prior build. Regression testing in Smalltalk systems appears to have more value than regression testing in classical software development. While I am not exactly sure why, I suspect that it is because inheritance and the distributed nature of Smalltalk systems make the impact of change more difficult to predict. The developer doesn’t know the whole system.

Regression testing does not work as well during consolidation or refactoring. Say component I is significantly changed so that it no longer use classes A and B, but rather uses classes X and Y. Call this new component I'. Also assume that the interface to component I' remains unchanged from I. The test sequence for component I' is now

Unit	Class X
Unit	Class Y
Interface	Class X
Interface	Class Y
Unit	Component I'
Interface	Component I'

The key to note is that the last test is the same as in the original testing. In other words, Interface(I) is the same test as Interface(I'), because the interface has not changed. If unit and interface testing are combined, as many folks do, then this is not true; Unit(I') is clearly not the same as Unit(I).


In practice, this means that if the tests are not separated into the unit and interface components, tests have *no* utility in verifying if the new version can replace

---

the old. In practice this happens all the time. The developer changes class A, changes the class A tests to reflect the change, and then is puzzled when someone else's use of class A breaks. This is because the developer changed the interface and then just changed the tests to match.

Good design and the enforcement of interface contracts reduces the exposure to this type of trouble. Good interface tests can be used to ensure that interface contracts are kept up.

As more software is developed by top-down construction, combining existing components in new ways, the importance of interface testing becomes greater. No longer can the developer of a component use SENDERS to find all the clients of that component. In the extreme view, if the interface changes in any way it is really a new component. New and improved perhaps, but still a new component.

This article has explored the public/private, proposed a solution, and then used that solution to define unit, interface, and aggregate testing. In our next article, we will discuss roles in the testing process with a particular focus on changes in time. 

---

Jeff McKenna is the founder and President of MCG Software, Inc., 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. He was chairman of OOPSLA '94. He may be reached at [mckenna@acm.org](mailto:mckenna@acm.org).