APPENDIX A

Case Study 4 - continued

To further investigate case study 4, we manually tracked down the code change which rendered the test case obsolete and found out that it is related to the use of a modified constructor of the *URL* class in method *openStream* in *StdXMLReader.java*. Noting that if the new constructor is replaced by the original one, $[[s_4]_{btr}, 2, \_]_{rtr}$ and $[<[s_4]_{btr}, [s_4]_{btr}>]_{str}$ would then be covered. Shown below are: (a) the original code in *NanoXML_v1*, and (b) the modified code in *NanoXML_v3* that renders test case {Parser1_vw_v1.java, testvw_29.xml} obsolete.

`public Reader openStream(String publicID,`
`  String systemID)`
`  throws MalformedURLException,`
`  FileNotFoundException,`
`  IOException`
`{`
`  systemID = "file:" + systemID;`
`  URL url = new URL(systemID);`
`  . . .`
	(a)

`public Reader openStream(String publicID,`
`  String systemID)`
`  throws MalformedURLException,`
`  FileNotFoundException,`
`  IOException`
`{`
`  URL url = new URL(this.currentSystemID, systemID);`
`  . . .`
	(b)

APPENDIX B

Case Study 8 - continued

Bug ID	Description
LOGBRIDGE-2	The bug is an inconsistency between log levels that causes bug JBCTS-634 to resurface. If JBCTS-634 was guarded with a *UCov* test requirement, this might have been avoided.
RAILO-2351	The bug was fixed and then resurrected 14 months later. A null value is not allowed as a value with a configuration that specifically allows "full null support".
JBSEAM-1501	The developer could not reproduce the constructor exception anymore, closed the issue, and documented that the team should watch for a resurrection. Evidently, the bug resurrected, and the issue was opened and fixed six weeks later.
JBESB-3305	Cache session issue seen with the *JBoss Messaging Queue* module, then resurrects while integrating with the *IBM Messaging Queue*.
MODE-921	Two dependency jar files contain files with the same names. The issue resurfaced (see MODE-925). *UCov tr* syntax needs to be extended to support runtime environment syntax to be able to avoid this resurrection.
JBIDE-10755	The issue is a wrong invocation of the module. The resurrection is recorded but the original bug is not mentioned. The team was worried about similar wrong invocations in other parts of the project and documented a need to watch for a resurrection.
ORG-1066	The issue resurfaces in a production release after few months and gets reopened. The resurrection issue was noticed, however, was not reproducible, and the team related this to a caching mechanism obscure to them. They close the issue with an acknowledgement that they do not know whether this was fixed or not.
WELD-316	This is an issue that the testing team could not duplicate. They closed it and they are waiting for it to resurface to reopen it. We think that a better strategy is to introduce a *UCov* test requirement even if that is not covered now, and wait for it to be covered, rather than wait for a client to report another failure that can be reproduced.
JBESB-851	This was mistaken to be a resurrected bug. It took 3 days to figure out it is not. A better bug and bug fix documentation mechanism such as *UCov* might have saved that time.
GUVNOR-215	Confusion about a resurrected bug caused the team to work on the wrong track and delayed the actual fixes.
JBNMAN-188	The bug was first thought to be a resurrection of another

	bug, however, when the prescribed workarounds were tried, the fail persisted and then the actual work started. An accurate bug/bug fix description as a test requirement would have helped to avoid that. *UCov* current syntax does not support that since the issue is a build and runtime environment issue.
JBREM-1043	This is a configuration issue that was fixed before and resurrected. The team could not locally reproduce, worked around it with a different configuration, and similarly to issue JBNMAN-188 closes awaiting another resurrection to be reported by a client.
ISPN-3307	-

APPENDIX C

Example with a Simple User-Defined Test Requirement

Method *isPrime(int x)* is meant to return *true* if *x* is a prime number, and *false* otherwise. P_1 is a faulty implementation of *isPrime(int x)* where the bug is in statement s_0.

```
// P1
public static boolean isPrime(int x) {
        if (x <= 1) return false;
        else if (x == 2) return true;
        else {
          int UpperLimit = (int) (Math.sqrt (x) +1);
          for(int divisor = 2 ; divisor <= UpperLimit ; divisor++ ){
    s0          if(x % divisor != 0) {  // bug: should be ==
    s1              return false;
                }
          }
    s2    return true;
        }
}
```

Assume that we are set to incrementally build a test suite T that achieves full statement coverage. Starting with x = 1, we select 6 test cases that cumulatively cover all the statements in P_1, namely, t_1:{1, false}, t_2:{2, true}, t_3:{3, true}, t_4:{4, false}, t_5:{5, true}, and t_6:{6, false}. Note how due to the bug at s_0, t_3 and t_6 return unexpected values, i.e., they are failing test cases. And since t_4 and t_5 do not increase coverage, they are ignored, thus, leading to T_1 = {t_1, t_2, t_3, t_6}, which yields full statement coverage.
As a result of t_3 and t_6 the bug is revealed and fixed in P_2. Also, applying *UCov*, t_3 is coupled with test requirement $[<[s_0]_{btr}, [s_2]_{btr}>]_{str}$ and t_6 is coupled with test requirement $[<[s_0]_{btr}, [s_1]_{btr}>]_{str}$.

```
// P2
public static boolean isPrime(int x) {
        if (x <= 1) return false;
        else if (x == 2) return true;
        else {
          int UpperLimit = (int) (Math.sqrt (x) +1);
          for(int divisor = 2 ; divisor <= UpperLimit ; divisor++ ){
    s0          if(x % divisor == 0) {  // bug is fixed
    s1              return false;
                }
          }
    s2    return true;
        }
}
```

In case P_2 is refactored into P_3 shown below, *UCov* detects that the intents of t_3 and t_6 are not preserved anymore, since s_0 is not executed in either case. Consequently, the user would replace t_3 by t_7 = {7, true} which covers $\{s_0, s_2\}$, and t_6 by t_9 = {9, false} which covers $\{s_0, s_1\}$. Now the test suite becomes T_2 = $\{t_1, t_2, t_7, t_9\}$ as opposed to T_1 = $\{t_1, t_2, t_3, t_6\}$. Note how if the user kept T_1, s_0 and s_1 (and the bug fix) would not be exercised.
Alternatively, if instead of applying *UCov*, the tester tried to achieve full statement coverage, she would realize that T_1 = $\{t_1, t_2, t_3, t_6\}$ is deficient for P_3, and that any test suite that achieves full coverage, would actually cover the bug fix. That is, in this example, full statement coverage is as effective as *UCov*.

```
// P3
public static boolean isPrime(int x) {
        if (x <= 1) return false;
        else if (x == 2) return true;
        else if (x % 2 == 0)      return false; // Added Code
        else {
          int UpperLimit = (int) (Math.sqrt (x) +1);
          for(int divisor=3 ; divisor <= UpperLimit ; divisor+=2){ // modified code
    s0          if(x % divisor == 0) {
    s1              return false;
                }
          }
    s2    return true;
      }
}
```