Automatic test factoring for Java

David Saff Shay Artzi Jeff H. Perkins Michael D. Ernst

MIT Computer Science and Artificial Intelligence Lab
The Stata Center, 32 Vassar Street
Cambridge, MA 02139 USA
{saff,artzi,jhp,mernst}@csail.mit.edu

Abstract

Test factoring creates fast, focused unit tests from slow system-wide tests; each new unit test exercises only a subset of the functionality exercised by the system test. Augmenting a test suite with factored unit tests should catch errors earlier in a test run.

One way to factor a test is to introduce *mock objects*. If a test exercises a component T, which interacts with another component E (the "environment"), the implementation of E can be replaced by a *mock*. The mock checks that T's calls to E are as expected, and it simulates E's behavior in response. We introduce an automatic technique for test factoring. Given a system test for T and E, and a record of T's and E's behavior when the system test is run, test factoring generates unit tests for T in which E is mocked. The factored tests can isolate bugs in T from bugs in E and, if E is slow or expensive, improve test performance or cost.

We have built an implementation of automatic dynamic test factoring for the Java language. Our experimental data indicates that it can reduce the running time of a system test suite by up to an order of magnitude.

1. Introduction

It is desirable for a test suite to contain small, fast, focused tests. A unit test is one example of a focused test: it exercises one component (such as a single class) without relying on any other component. Focused tests execute quickly, so they can provide fast feedback, and they can be run frequently. Focused tests isolate errors to a small amount of code, easing debugging by concentrating a developer's attention on a smaller set of places that might be the source of an error. Because they are faster and there are more of them, focused tests are more amenable than system tests to test selection and test prioritization, which can further reduce the amount of time before the test outcome is known.

Often, focused tests are not available. Instead, a software system may have system tests: long-running, end-to-end tests that exercise much of the functionality of the entire system. System tests have their own advantages. System tests tend to be easier than unit tests for people to create and understand. Because there are fewer of them, they are easier to manage. They are less brittle in the face of changes, such as modification of an internal interface. They tend to be more comprehensive, both because they cover more code and because they create more complex data structures that may expose additional errors. Some so-called unit tests are actually system tests in disguise: a test of one component may utilize other parts of a system. (A true unit test would use stubs or mock objects for all interaction with the rest of the system.)

Our research aims to provide the benefits of focused tests to a developer who has only written system tests. In particular, we propose a technique, *test factoring* [12], for automatically converting a system test into a collection of focused tests. Test factoring creates fast, focused unit tests from slow system-wide tests; each new unit test exercises only a subset of the functionality exercised by the system tests. The focused tests can augment (but are not intended to replace) the system tests. When a system test is modified, or the system itself is changed in a way that is incompatible with the factored tests, the focused tests can be automatically re-generated.

Test factoring takes three inputs: (1) a program, (2) a system test, and (3) a partition of the program into the "code under test" (for which factored tests are desired) and the (untested) "environment". The output of test factoring is a set of factored tests for the code under test. Running the factored tests does not execute the "environment" part of the original program, only the "code under test" part. The test factoring procedure can be repeated, varying the program, the system test, or the partition.

Our approach to test factoring replaces the environment part of the program by mock objects. Mock objects, like stubs, simulate an expensive resource, but mock objects also assert that they are used in a specific way [4]. If the simulation is faithful to the expensive resource, then a test that utilizes the mock object rather than the expensive resource can be cheaper (e.g., faster). Some examples of expensive resources that might be replaced by mock objects are: large or slow computational resources such as databases; data structures and disks (setting them up in exactly the required state may be difficult, or side effects may be unacceptable); network communication (whose costs include delay, the need for extra hardware such as remote computers and network infrastructure, and the difficulty of isolating irrelevant effects); hardware resources; and human attention.

Developers have long constructed stubs and mock objects by hand. Our contribution is the automatic creation of mock objects via a dynamic, capture–replay technique. The capture stage executes the system test, recording all interactions between the code under test and the environment in a "transcript". During replay, the code under test is executed as usual. However, at each point that it would have interacted with the environment, no computation is performed; instead, the value that was observed during the capture stage (and was recorded in the transcript) is used.

Test factoring complements other techniques for reducing the cost of testing. Test selection [7] runs only those tests that are possibly affected by the most recent change, and test prioritization [16] runs first the tests that are most likely to reveal a recently-introduced error. For test suites with long-running or expensive tests, selection and prioritization are insufficient. We propose augmenting them with test factoring, which from each large test generates multiple unit tests that can be run individually and are amenable to test selection and prioritization.

Section 2 describes our test factoring procedure, which creates mock objects, and Section 3 describes the capture–replay technique that underlies it. Sections 4 and 5 present the details of our implementation, which works on Java programs, and Section 6 presents a crucial optimization. Section 7 describes our experiments and their results. The paper concludes with a discussion of future and related work and a recap of contributions.

2. Test factoring via mock objects

Our prototype implementation of automatic test factoring operates via automatic creation of mock objects. A mock object, which is a stub that requires that it is used in particular ways, has a subset of the functionality of a real object — for instance, the mock object may be able to respond to only specific queries.

Suppose that a software system is composed of two parts, T and E; we write the system as "$\mathsf{T}|\mathsf{E}$". T (the code under test) makes calls into E (the environment) and uses the results that E returns.[1] The tests for T (or for the system as a whole) do not depend on the full functionality of E; rather, the tests exercise E in a particular way. This is not a design goal of the tests, but a consequence of the way that T and E are designed to interact, and of use of a finite set of tests.

After a change to T, the system should be re-tested. If the change does not affect E or the way that T and E interact, then testing just T is sufficient. In particular, it may be faster and cheaper to run the tests not on $\mathsf{T}|\mathsf{E}$ (that is, the original system), but on $\mathsf{T}|\mathsf{E_m}$, where $\mathsf{E_m}$ is a mock version of E. If $\mathsf{E_m}$ faithfully simulates as much of E's functionality as T uses, then the result of the tests will be the same as if they had been run on the original system. A mock object $\mathsf{E_m}$ that is constructed for a specific set of tests is not necessarily appropriate for a different set of tests; a different mock object $\mathsf{E'_m}$ may be required.

One common implementation of a mock object incorporates a lookup table, which we call a "transcript". The transcript contains a list of expected method calls: each entry consists of the method name, the arguments, and the return value. The mock object maintains an index into the transcript; for each method call to the mock object, the mock object verifies that the method name and arguments are the same as those of the current transcript entry (throwing a ReplayException if they are not), returns the current transcript entry's result value, and increments the index. Section 8.1 notes ways in which the transcript can be generalized and made more flexible, so that the mock object permits certain calls to be reordered or otherwise modified.)

[1] Our prototype implementation handles all interactions, including calls from E to T, shared state such as public variables, use of static methods and variables, aliasing, etc.; see Section 5.1.

2.1 Test factoring inaccuracies

Given an accurate transcript for $\mathsf{E_m}$, it is fairly easy to ensure that a test executed on $\mathsf{T}|\mathsf{E_m}$ gives the same result as the test executed on $\mathsf{T}|\mathsf{E}$. However, there is little point in running such a test: T is already known to pass the tests. The goal of testing is to gain information about a changed software system. When T is changed to $\mathsf{T'}$, it is desirable to test $\mathsf{T'}$. $\mathsf{T'}|\mathsf{E_m}$ should give a faster answer than $\mathsf{T'}|\mathsf{E}$, but it is not guaranteed to produce the *same* answer: it can produce an accurate result ("pass" or "fail"), a false success, a false failure, or a ReplayException (described below).

A false success occurs when the factored test $\mathsf{T'}|\mathsf{E_m}$ passes but the system test $\mathsf{T'}|\mathsf{E}$ fails. A false failure occurs when the factored test fails but the system test passes. Different test factoring approaches can take different approaches to reducing one or the other type of error, possibly trading them off against one another. Our prototype implementation never yields false successes or false failures; however, an implementation can trade off false successes and failures against ReplayExceptions, for instance by permitting calls to be reordered or by inferring the environment's response to a call that was never observed in practice [12]. It is straightforward to use the factored tests in a way that can cope with either false successes or false failures. Recall that factored tests run quickly, compared to the original tests.

- If false successes are expected, then factored tests can be prioritized before system tests. If a factored test correctly fails, then the test suite gives much quicker notification of the error that a developer has introduced. If a factored test falsely passes, the only cost is a brief delay before running the original system test.
- If false failures are expected, then the factored tests can be used to select which system tests to run. If a factored test correctly passes, then the corresponding system test will not be selected, and the suite gives much quicker notification that the developer has not introduced any errors. If a factored test incorrectly fails, then the only cost is the small extra cost of running the factored test.

A ReplayException indicates that the assumption inherent in the test factoring methodology — that $\mathsf{T'}$ uses the environment in the same way that T did — has been violated. The factored test yields a ReplayException if the sequence of calls from $\mathsf{T'}$ to $\mathsf{E_m}$, or the arguments, are different than those that were captured during the training run of $\mathsf{T}|\mathsf{E}$ from which $\mathsf{E_m}$ was created. Handling a ReplayException is straightforward: the full system $\mathsf{T'}|\mathsf{E}$ must be run — both to obtain a test result for $\mathsf{T'}$ and (in the background) to create a new mock object $\mathsf{E'_m}$.

3. Capture and replay technique

We introduce an automatic technique that creates mock objects via a dynamic capture–replay approach. (By contrast, static test factoring could analyze the source code of the program and the system test; it introduces a different set of tradeoffs than dynamic test factoring, and is not considered here.) As noted in Section 1, the three inputs to test factoring are a program, a system test, and a partition of the program into the code under test and the environment.

1. The capture step occurs ahead of time, not at test time. It executes the tests (we assume they pass) in

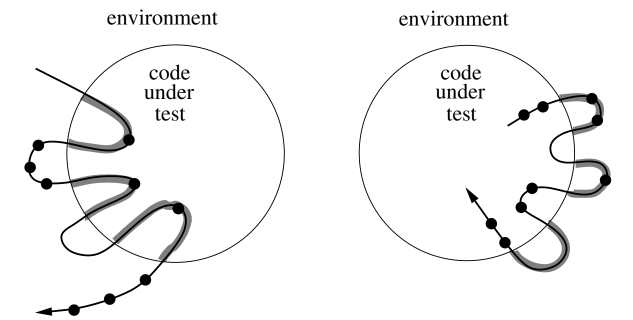

Figure 1: The life-cycle of two objects. On the left, an object is created in the environment; on the right, an object is created in the code under test. The curved line represents the object being passed between the environment and the code under test; the circle is the boundary between them. The wide gray line background indicates where the object must record a transcript (during capture) or be replaced by a mock object (during replay). The black circles represent method calls against the object.

the context of the original system $\mathsf{T}|\mathsf{E}$, and records all interactions between T and E. The resulting transcript indicates, for each call, the method name, the arguments, and the return value. It can be thought of as encoding a transition function for objects in the environment.

2. The replay phase occurs during execution of the factored tests, that is, $\mathsf{T}'|\mathsf{E}_\mathsf{m}$. The system is run as before, but with real objects E replaced by mock objects E_m; the original environment is never executed during the factored test. E_m uses the recorded behavior in order to simulate the environment. Whenever a mock object is called, it checks that it was called with the same arguments as the next entry in the transcript. If so, it returns the value from the transcript; if not, it throws a ReplayException.

Suppose that the `ArrayList` class is part of the environment, and consider an `ArrayList` object l that was created in the environment and passed back and forth between the environment and the code under test; see the left side of Figure 1. During the capture stage, method calls on l that are made from within the code under test must be recorded (in addition to performing the requested operation); method calls on l that are made from within the environment should not be recorded. During the replay stage, method calls on l that are made from within the code under test must return the recorded value; no calls will be made on l from the environment, because the environment is replaced by mock objects.

Objects created in the code under test are treated symmetrically. Such an object is accessed directly from within the code under test; when it is passed to the environment, it records the environment's interactions with it. (See the right side of Figure 1.) As a result, the transcript records all interactions between the environment and the code under test. Since the transcript records both arguments and return values, it could equally well be used to run the code under test in the absence of the environment (which is how we use it), or to run the environment in the absence of the code under test (effectively reversing the roles of the environment and the code under test).

4. Instrumenting Java classes

This section describes our approach to replacing classes and objects in a Java programs by instrumented versions. The capture step uses a replacement class that records behavior to a transcript, and the replay step uses a mock class that reads from the transcript.

Our approach proceeds in two steps. The first step introduces a new interface for every class in the program, and retrofits each class to implement its interface. These interfaces separate type inheritance from implementation inheritance and are useful for a variety of analyses other than test factoring. The second step introduces new classes that implement the interface, and therefore can be used in place of the original (retrofitted) ones.

This section describes our approach. It presents requirements, the interface introduction step, the instrumented classes that implement the interfaces, and finally discusses other approaches to the problem.

4.1 Requirements

The instrumentation technique should handle all of the Java language, including class loaders, native methods, reflection, etc. Since source code is not always available, instrumentation must be performed on bytecode.

It must be possible for an instrumented class to co-exist with the uninstrumented version. For instance, it would be prohibitively difficult to write and debug instrumentation code that was not permitted to use JDK classes such as `ArrayList`. However, it is essential not to "instrument the instrumentation": the instrumentation code must have access to the original classes, to avoid infinite loops and to permit accurate measurements.

Native methods make assumptions about the classes of their arguments and the fields that those arguments contain. Therefore, an invocation of a native method must pass in uninstrumented objects.

Instrumenting the built-in system classes (`java.*`, etc.) presents special difficulties. The JVM hard-codes assumptions about the system classes — for example, adding a field or method to `Object` causes Sun's JVM to crash — so instrumentation must not add or remove any field (nor method, in some classes such as `Object`, `Class`, and `String`). The JVM makes calls into the JDK, much as with native methods, so it is not safe to change the type of any public member (field, method, parameter, return value). As a minor point, system classes cannot be instrumented dynamically, because about 200 classes are loaded by the JVM before any user-supplied code can take effect; we avoid this problem by statically instrumenting file `rt.jar`.

Despite its challenges, we must instrument the JDK, because code under test commonly interacts with the environment via an `ArrayList` or other JDK class. This constrains our implementation strategy. We use the same implementation strategy for user code as well, for uniformity.

Our implementation satisfies all of these requirements.

4.2 Interface introduction

We wish to replace objects in a test execution by different (capturing or replaying) objects. Making the new objects subclasses of the old would permit the original code to run

Before:

```
class C {
  Integer foo(int x) { ... }
  void bar(Date d) { ... }
}
```

After:

```
interface C__iface {
  Integer__iface foo__iface(int x);
  void bar__iface(Date__iface d);

class C implements C__iface {
  Integer foo(int x) { ... }
  Integer__iface foo__iface(int x) { ... }
  void bar(Date d) { ... }
  void bar__iface(Date__iface d) { ... }
}
```

Figure 2: Example class before and after interface introduction.

unmodified, but is problematic or impossible due to final classes and methods, reflection, and similar code constructs. Instead, we perform interface introduction: we change each class reference in the code into a reference to an interface, and change each class to extend that reference. The code can run with the original objects, but any other object that extends the interface can be substituted instead.

Interface introduction creates, for each class `C`, an interface `C__iface`. The methods of `C__iface` are those of `C`, but with `__iface` appended to the end of each name, and with all reference types replaced by their `__iface` versions. Classes are retrofitted to implement the new interface by appending `__iface` to each reference type and method name in the signature or body.[2] These side effects to the original classes have no effect on their behavior. (They do change method calls into interface calls, but that is an implementation detail of the JVM.) For an example, see Figure 2.

Introduction of interfaces does not permit changing the behavior of a few constructs, including accesses of static variables and uses of reflection. Our transformation converts these constructs into calls to special hook routines. Then, when new objects are introduced into the program to replace those, the hook routines can be set appropriately. It would be possible to use this hook technique throughout, changing most program operations into calls to hook routines, but use of interfaces is more efficient and makes the resulting code much more readable.

Classes `Object`, `Class`, and `String` are specially used by the JDK (they are arguments to `Object` methods and native methods), so interfaces cannot be added for them. None are needed for `Object` — it is already the root of the class hierarchy — and we use the hook mechanism for `Class` and `String`.

The interface introduction step only needs be done once for libraries such as the JDK; this is a critical feature that permits the JDK to be statically transformed once rather than once per analysis (which would be impossible in any event, as noted above).

[2]As a special case, built-in system classes, which are instrumented statically rather than dynamically, make a copy of each method so that the original one still remains.

```
class C__capturing implements C__iface {
  C _delegate;
  Integer__iface foo__iface(int x) {
    ... // record arguments
    Integer result = _delegate.foo(x);
    ... // record results
    return wrapOrUnwrap(result);
  }
  void bar__iface(Date__iface d) {
    ... // record arguments (no results to record)
    _delegate.bar(wrapOrUnwrap(d));
  }
  WeakHashMap<C, C__capturing> hm;
  C__iface wrapOrUnwrap(C__iface in) {
    if (in instanceof C__capturing) {
      return ((C__capturing) in)._delegate;
    } else if (in instanceof C) {
      C real = (C)in;
      if (!hm.contains(real)) {
        hm.put(in, new C__capturing(real));
      }
      return hm.get(real);
    } else {
      throw new Error("this can't happen");
    }
  }
}
```

Figure 3: Capturing version of class `C` from Figure 2. Capturing wrappers are only created for classes in the environment.

4.3 Capture and replay classes

In our approach, both the capture and replay steps replace some objects from the environment by different objects that satisfy the same specification. Only those objects that interact with the code under test need to be replaced; for reasons of correctness and performance, other objects from the environment should not be affected.

1. While capturing, the replacement objects are wrappers around the real ones. The wrappers delegate the work to the real objects and record, to a transcript, arguments and return values.

2. While replaying, the replacement objects are mock objects that read from the transcript. A mock object verifies that the arguments are as expected and returns whatever the transcript indicates.

Figure 3 shows an example of a capturing version of a class, which is a wrapper class that delegates all work to the underlying object while recording arguments and return values.

When the program counter is in the code under test, all reachable objects whose type is from the code under test are accessed directly, and all reachable objects whose type is from the environment are accessed via instrumenting wrappers; and symmetrically when the program counter is in the environment.

Whenever the object is transferred from the environment to the code under test (or vice versa), it must be transformed from an instrumenting version to a non-instrumenting version (or vice versa); that is, it must be wrapped or unwrapped. Figure 1 showed this graphically; the line entering and exiting the circle represents program control (and objects) passing into and out of the code under test. Because all interaction between the realms is via method calls (see Section 5.1), the only way for an object to cross the boundary is to be passed as an argument or returned as a result.

4.4 Alternative implementations

We initially hoped to use the Twin Class Hierarchy approach, which is designed specifically to permit instrumentation of Java standard libraries [3]. We found that that approach does not scale, however. The most serious problem is that wrappers must be written by hand for each native method, of which there are a great many used by any realistic program. Another difficulty is that every array (even those in the environment) must be wrapped. This results from the need to simulate, in the twin class hierarchy, the way that arrays are represented in the real class hierarchy. Real arrays are subclasses of `Object`, but twin arrays should be subclasses of `Object__twin`; handling covariance, dimensionality, and casting adds additional complexity, both for the implementation and at run time.

One alternative would be to always instrument classes (so no objects of the original type would appear anywhere in the system), rather than converting them between instrumented and uninstrumented versions when crossing the boundary between the code under test and the environment. The instrumentation could be enabled or disabled depending on whether the program counter was in the code under test or the environment. If implemented as a conditional capture or replay statement inserted into every method in the system, this approach could work (so long as no new fields were introduced), but introduces greater complexity, overhead, and potential for error than making changes only at the boundary. Furthermore, we find it compelling to decide via run-time dispatch whether a particular line of code is being captured or not; this is in the spirit of the underlying object-oriented virtual machine. If implemented as wrappers, then translation is required nonetheless, for two reasons: because the JVM and native methods require uninstrumented classes, and because any use of `this` in the original code would permit an uninstrumented object to escape. We choose to perform this translation at the boundary rather than at every method call in the program. Additionally, wrapping every object in the system (rather than just those that participate in interactions between the code under test and the environment) could lead to unacceptable slowdowns. Finally, universal replacement does not permit the "partially mocked libraries" optimization (Section 6).

A final approach would be to use debugger-based monitoring. Our experience with such an approach indicates that it would be orders of magnitude slower than the approach we chose, for instance because of the need to switch between the debugged process and the monitoring process. Since the purpose of a factored test is to have good performance, we decided that this run-time slowdown as not acceptable.

Concurrently with us, Orso and Kennedy [8] have begun implementing a capture–replay system with similarities to ours. Unlike our algorithm, theirs does not handle important features of Java, such as native methods and reflection, that we found crucial for running real-world Java code. Additionally, we provide an empirical evaluation, whereas they have run their system on a single 3,300-line program but not applied it to any tasks.

5. Capturing all behavior

The basic procedure of Section 3 captures procedure calls between the code under test and the environment. This section discusses how we have addressed other types of interactions between the code under test and the environment.

5.1 Non-method-call communication

Before being run (during capture or replay), the program undergoes a semantics-preserving transformation that replaces array accesses and static and instance field accesses by method calls. For example, field getter and setter methods are added, and a per-class singleton mock object handles constructors and static methods. Client code is then transformed to use those new methods. The reason for this transformation is that method calls are easier and simpler to instrument than the code constructs that they replace.

5.2 Callbacks

The behavior of the environment consists not just of how it is used by the code under test, but also how it uses the code under test. Therefore, calls from the environment to the code under test must be captured; these may be callbacks, or the system test might have started in the environment rather than in the code under test.

We modify the description of the environment's behavior to include in the transcript, for each call, not just the arguments and return value, but also any other interaction with the environment, such as callbacks across the boundary. The replay stage replays the full behavior of the environment, including callbacks, and it checks that the return values of the callbacks are as expected.

5.3 Objects passed across the boundary

Procedure arguments and return values can be objects as well as primitive values. If the code under test manipulates an object whose type is part of the environment, that manipulation counts as an interaction with the environment. It is monitored during trace capture and replayed when running the factored test.

We augment the transcript to include a *crossover cache* of objects that have passed across the boundary. This permits object equality to be determined during capture and maintained (by returning the proper object) during replay. This functionality is handled by the `wrapOrUnwrap` method of Figure 3.

5.4 Arrays

The JVM treats arrays as a hybrid between objects and primitives. They subclass `Object`, and certain methods call be called on them, but they are accessed via special bytecodes rather than by method calls. It is possible, but problematic, to wrap arrays by objects [3]; instead, our analysis also treats them specially. When crossing the boundary between the code under test and the environment, an array is replaced by another array whose element type has been transformed into a capturing or replaying version. The crossover cache relates the two arrays, and operations on the "wrapper" array are translated into the appropriate operation on the original array, plus translation for elements that cross the boundary. This ensures that interaction through aliased arrays is properly reflected on the other side of the boundary.

5.5 Native methods and reflection

The wrapped version of a native method always delegates to the actual code; as noted previously, the original object must be passed to the native code, so passing a wrapper object to a native method will cause an exception. We use partially mocked libraries (Section 6) to ensure that any object from such a library that has not crossed the boundary is not wrapped. This permits native methods to be called from either the environment or the code under test, and in our experience, with a reasonable partition it is extremely rare for an object to cross the boundary before being passed to a native method.

Reflective calls, like native calls, should be provided with original, never wrapped, objects. Our solution is similar: the reflective call is intercepted and the arguments unwrapped if necessary, then the results wrapped if necessary. The reflection mechanism cannot observe wrapped classes, and the invariant is maintained that user code cannot observe an unwrapped object on the other side of the boundary than where it was created.

5.6 Class loaders

Large Java programs frequently use multiple class loaders to control which versions of a class are loaded, to perform transformations, to isolate parts of a program from one another, or for other reasons. For example, the Eclipse IDE, which is written in Java, makes extensive use of class loaders.

Our instrumentation also uses class loaders: when the program requests a class such as `MyClass__capturing`, the class loader loads the class `MyClass`, transforms it (rewriting bytecodes to add interfacing and/or to insert capture or replay logic), creates a fresh wrapper class based on its methods and fields, and returns the wrapper class.

Our implementation handles programs with multiple class loaders by creating a custom class loader for each class loader that the program (dynamically) creates.

6. Partially mocked libraries

As described so far, each class in a program is either part of the code under test or part of the environment. Unfortunately, such a partition is too restrictive. Consider a utility class such as `String` or `ArrayList`. If `String` is part of the code under test, then all uses of `String` by the environment become part of the resulting factored test, increasing its running time. If `String` is part of the mocked realm, then any change to how a tested class uses `String`s (even internally) is likely to prevent replay. Furthermore, during replay, there is no need for each `String` to be replaced by a mock object: the library code itself is probably as fast.

We extend the partition of the program from two parts to three: code under test, environment, and partially mocked libraries. A common choice for the latter is the libraries in `java.*`, `sun.*`, `javax.*`, etc. This realm consists only of classes: each object that is instantiated from the partially mocked libraries is placed in the mocked or tested realm; the same realm as the object that instantiated it. Each *object* of a class in the partially mocked libraries is either in the code under test or in the environment, depending on where it was created, and is transformed as it crosses the boundary just as described in Section 4.3. Such objects that are used entirely within the code under test and never escape to the environment need not be captured or mocked. Objects used for internal storage and computation within the environment, and that never escape to the code under test, will never even be mentioned in the factored test, since they are part of the replaced, simulated logic.

Static fields and methods in the partially mocked libraries are treated as being in the environment. The environment is typically be larger than the code under test, so this choice minimizes the size of the transcript.

Use of partially mocked classes makes the instrumentation more complex, since only some objects of the class are captured and mocked. Use of partially mocked libraries also complicates issues of object equality in the presence of aliasing. Our implementation addresses all these issues; for reasons of space, we omit a detailed discussion.

7. Experiments

This section reports experiments that measure the efficacy of test factoring.

Our methodology satisfies three key desiderata for evaluation of testing tools: the use of real code, real errors, and a realistic testing scenario. The program we studied, Daikon, consists of 347,000 lines of Java code that implements sophisticated algorithms and makes use of Java features such as reflection, native methods, callbacks from the JDK, and communication via side effects. The code was under active development, and all errors were real errors made by the developers; we did not use synthetically generated or inserted errors, which may have quite different characteristics. Finally, our testing scenario uses frequent code snapshots under the assumption that developers wish to test frequently, which some of the developers were already doing. An alternative is to use revision control logs to see how much time could be saved when testing various versions of the code, and we also report results from such a methodology. However, the latter methodology does not indicate the full benefits of a testing technique in real practice, when developers run tests throughout development — and before, not after, committing changes to the repository. It only indicates how much faster a developer could be notified of a test success, not how much faster a developer could be notified of a test failure, which is typically more important.

7.1 Experimental subjects

Our evaluation methodology makes use of a log of actions that are recorded in the background while a developer works, and also a revision control log.

Given the log of developer actions, we can reconstruct the developer's file system at any moment in time. We can determine whether, had the tests been run at that moment, they would have passed or failed (thus, we know when the developer introduced and corrected errors), and how long the tests would have taken to run. Furthermore, we can apply techniques such as test factoring in order to determine their effect on test suite execution: how much faster a developer would have learned the test outcome, had the developer been using that technique. The log also indicates when the developer ran the tests.

This paper reports on data from two developers (one professional and one undergraduate) working independently on the Daikon invariant detector [2] between June 23 and August 20, 2004. Figure 4 summarizes the changes to each developer's copy of the program in the file system. Each moment corresponds to the developer saving (or otherwise

	Code changes		Errors		
	Files	Moments	Episodes	Avg. len.	Time
Dev. 1	254	1231	29	13.1 hours	57%
Dev. 2	259	1274	41	11.9 hours	38%
CVS	262	104	n/a	n/a	n/a

Figure 4: Summary of syntactically correct code changes made by two developers. An error episode begins when the tests would first begin to fail (had they been run at that moment) and ends when they would pass again. The "Time" column indicates what percentage of wall clock time the tests failed.

modifying) a file. We ignore changes to generated files (such as Java `.class` files) and to files that are not part of the program code, but include non-Java files from which Java files are generated. We also ignore the 41% of code changes that caused a compilation error or made the unit tests fail. Such errors may indicate that a developer was in the middle of an edit; in any event, they are easy and fast to discover, and development environments can indicate compilation errors and unit test errors [13] on the fly.

The revision control log indicates the changes that were checked into the CVS repository by each developer working on Daikon (not just the two most active developers, whose file system actions were recorded). We arbitrarily chose to use the CVS data from the period from March 1 to September 1, 2004. Since developers test before checking in changes, this data primarily indicates how much test factoring can speed up indication of test successes, whereas the fine-grained file system log information indicates how much test factoring helps to indicate both test successes and test failures.

7.1.1 Partitions

Test factoring requires a partition of the code into the environment, the code under test, and the partially mocked libraries. At each point in time, we chose the class containing the `main` routine as the environment, the changed classes as the code under test, and all other classes as the partially mocked libraries. Another partition may have been better, but we did not investigate that possibility.

To create the mock environment $\mathsf{E_m}$, we used the version of the program that most recently passed the tests at midnight. In other words, for evaluating test factoring on a particular day, we assume that capture occurred the previous midnight (or earlier if the tests did not pass as of the previous midnight). This simulates a development methodology in which each night, many factored tests are prepared against the eventuality that one or more of them will be needed the next day. Because bad partitions (that generate excessively large transcripts) can be quickly identified, and the same partitions often arise on different days (a total of 126 distinct partitions in our data), this is a reasonable assumption. This approach actually underestimates the benefits of test factoring, because when a ReplayException is encountered, a testing tool should immediately regenerate the factored test, rather than waiting until the next midnight.

Whereas we found a number of partitions that enabled test factoring to produce faster versions of the tests, other partitions were useless for test factoring. In some cases this was because the class that contains the `main` routine was changed, and our heuristic places that class in the environment. In other cases the partition induced extremely large transcripts, because classes in the environment interacted extensively with classes in the code under test. (An example of this is that Developer 1 completely restructured Daikon during the summer. Test factoring would yield correct results, but due to file I/O overheads it would be slower than the original tests.) In yet other cases, limitations in our prototype tool prevented capture from completing; we are working to fix these bugs. Bad partitions were easy to recognize, and in such cases the testing infrastructure would run the original tests, not the factored ones. Therefore, we omit these from the experimental measurements.

Daikon is a very difficult subject for test factoring. The fundamental problem with Daikon is that it processes a huge amount of data, and that data is passed to many parts of the program. A typical run processes a gigabyte of trace data and calls methods from over 100 distinct classes on each sample read from the trace. Equally seriously, no part of Daikon is particularly expensive; its slowness stems from many operations, not from expensive ones. Therefore, it is difficult to find a good partition for Daikon. The nature of the developers' changes (including an extensive refactoring) made matters even worse. (And, our CVS experiments are on larger-than-average changes, since developers accumulate code changes until they check them in to source control.) Therefore, our results on Daikon are extremely encouraging, and we intend to collect data for additional programs in order to see whether, as we expect, they will prove more amenable to test factoring.

7.2 Measurements

7.2.1 Baseline

Daikon contains two sets of tests: unit tests and regression tests. The unit tests primarily cover libraries and other targeted parts of the Daikon codebase. They are automatically executed each time the code is compiled. Since they take less time than compilation itself, test factoring is unlikely to help significantly, so we ignore them for the purposes of this paper: test factoring is most applicable to long-running tests. The regression tests are end-to-end system tests that exercise much more of Daikon. Running the regression tests from scratch takes 60 minutes. Running them with the `make` command, as all Daikon developers do, completes in 15 minutes. This avoids certain unnecessary work performed by front ends, which are not part of the Java codebase and which did not change during the monitored period. This use of `make`'s dependency mechanism can be viewed as a manual test factoring; our goal is to reduce the need for such manual effort.

Our results would be dramatically better if we took 60 minutes, rather than 15 minutes, as our baseline, but such results would be misleading, as they would not be indicative of the benefit in practice to developers who have already made some effort to reduce test times. The developers reported that 15 minutes is still long enough to be a nuisance; they did not run tests as often as they would have liked to, or they wasted time waiting for the tests to complete, or they continued editing while the tests ran, which occasionally made debugging more difficult. In fact, the developers sometimes ran only a portion of the regression tests, which can be viewed as manual test selection; the logs indicated a number of instances in which this practice was unsound, indicating the need for a tool such as ours.

7.2.2 Quantities

The purpose of testing is to indicate that a codebase either passes or fails its tests. A developer who tests frequently (as is universally recommended) expects the tests to usually pass, and they usually do pass. The earlier a developer is informed of an error (a test failure), the easier, faster, and cheaper it is to correct the error; therefore, a key thrust of testing research, including ours, is earlier notification of errors. Developers gain a complementary but lesser benefit from earlier notification of test successes: when uncertain about their code, they can wait less time before proceeding with code changes. (However, developers do not always wait for tests to complete before proceeding with code changes.)

We measure the following quantities.

1. *Test time*, the amount of time required to run the tests.

2. *Ignorance time*, for a particular error and test strategy, is the time between when the error was introduced and the first test failure in the test suite. (An alternative formulation would measure from error introduction to completing running the test suite, but that is not realistic: the developer becomes aware of the error as soon as the first test fails, and often the test suite terminates at that point.) For a given test suite, the ignorance time is at least as large as the run time of the quickest test that exposes the error. If a testing technique fails to expose an error, then its ignorance time is the full duration of the error. A technique might fail to expose an error because the technique is unsound, or because it is too slow and the developer corrected the error before the technique would have notified the developer. (The developer correcting the error is indicated by a file system change that makes the tests pass.)

3. *Waiting time* is the time between starting tests and successfully completing them. Successful test suite completion always requires running the entire test suite. We measure waiting time from every file system change by the developer for which the program passes the tests — that is, all of the moments of Figure 4 that are not within an error episode.

Our measurements account for the time to run the tests, but we ignore the time to compile and to run unit tests. The former is eliminated by the use of incremental continuous compilation, which is supported by many development environments such as Eclipse. The latter is also fast.

7.2.3 Testing methodologies

Our experimental evaluation compares two techniques for reducing ignorance time and waiting time. In each case, we report the ratio between measurements for continuous testing, and measurements for continuous testing augmented with test factoring.

7.2.3.1 Continuous testing.

Continuous testing [11] uses excess cycles on a developer's workstation to continuously run tests in the background, providing rapid feedback about test failures as source code is edited. Continuous testing starts rerunning the tests (terminating the current test execution) when the program recompiles after the developer saves his modifications. If tests take a long time to run, then the benefits of continuous testing are greatly lessened, for it cannot achieve the goal of providing feedback almost immediately after a developer commits an error, which makes the error easier to fix [1, 14]. Test factoring has the potential to help continuous testing achieve these goals.

Continuous testing can be thought of as the most frequent possible testing strategy. However, there is no need for the developer to remember to run the tests, and there is no penalty to the developer in terms of nuisance or distraction.

7.2.3.2 Test factoring.

Test factoring seeks to create additional small tests (*factored tests*) to a test suite that expose the same errors as the large tests, thus reducing ignorance time. It can also reduce waiting time, but see Section 2.1 regarding whether factored tests should be added to, or should replace, an original test suite. In our evaluation, since there are no false positives or false negatives, the factored tests replace the original tests (but if a ReplayException occurs, the original test must be run).

We measured how much test factoring improves continuous testing. Continuous testing is a state-of-the-art technique that provides feedback much faster than when developers run tests manually, so it is a reasonable baseline.

	Test time	Ignorance time	Wait time
Dev. 1	.79	1.56	.59
Dev. 2	.52	1.37	.97
CVS	.09	n/a	.09

Figure 5: Experimental results. Each cell is the ratio of time with continuous testing to time with continuous testing and test factoring, averaged over all points to which test factoring applied. That is, each cell indicates how much improvement test factoring contributes. Smaller numbers are better.

7.3 Results

Figure 5 provides experimental results. Test factoring generally reduces the running time, but it is influenced by the developer's working style. For example, Developer 1 had a larger set of changed files (due to less frequent checkins) and was also making a large architectural change. Developer 2, and other developers whose data are reflected in the CVS logs, made smaller changes at any time.

It is notable that use of test factoring actually increased Developer 1's ignorance time (from 9 to 14 seconds). The reason is that the Daikon developers had already performed manual test prioritization. The first system test in the system test suite was both very fast to run, and it usually failed when the suite failed. Therefore, it was difficult to improve its performance (the factored first system test was little faster than that test itself), and a ReplayException that forced execution of the entire first test would increase ignorance time (by adding the time to run the factored version, after which the test itself had to be run). Even on failing runs, overall test suite execution time was reduced, but ignorance time, which measures only time to the first failure, was increased. A sophisticated testing framework could avoid factoring those system tests, but we did not simulate such a framework.

For the CVS data, the wait time is always the same as the test time. This is not necessarily the case, if the tests are unable to complete before the developer makes the next change. CVS checkins tend to be spaced far enough apart to permit tests to complete between them.

8. Resilience to code changes

A factored test introduces assumptions about the implementation details of the functionality being tested; if those assumptions are violated during program evolution, the factored test may return a false success or a false failure. It is easy to recognize such problems and correct them by re-running the test factoring procedure, but in the meanwhile the factored test is useless for regression testing. We wish test factoring to construct factored tests that are useful for as long as possible — that is, that are resilient to many code changes to the code under test.

For example, consider a test for a method that inserts records into a database. If making calls against the database is slow, the test may be factored to use a mock object that simulates the behavior of the database and ensures that the expected calls are made to the database API. If the code under test is modified to insert the records in a different order or to use a database API call that inserts them all at once rather than one at a time, then the original test will still pass, but the factored test will likely fail, because the mock object receives unexpected calls.

A test factoring procedure can always be extended to eliminate an erroneous assumption. For example, with knowledge of the semantics of the database API, the database mock object could be extended to accept all valid data input call sequences. However, the only way to eliminate *all* assumptions is to turn the factored tests into exact replicas of the original test, eliminating the speed and bug-isolation advantages of test factoring.

8.1 Change language

A change language [12] characterizes some of the ways that a developer may modify a codebase. It is a kind of pattern language that breaks down complex maintenance goals into a set of simple code changes, much like refactorings [5]. We believe that a developer facing a maintenance task consciously or unconsciously uses a change language. If this is the case, then understanding a developer's change language would allow prediction of which changes they are likely to make. This in turn would help to maximize the "lifespan" of factored tests before their assumptions are violated and they become useless.

The change language of a test factoring procedure is the set of program transformations and refactorings for which its factored tests remain valid. The change language of the simple procedure of Section 3 includes any refactoring that does not affect observable behavior. This includes standard refactorings such as Extract Method and Inline Method, but also many wholesale re-implementations that maintain unchanged the order and arguments of method calls and returns, public field accesses, etc.

It is desirable to extend the change language of test factoring to other changes that are common during development tasks. However, the factored tests need not tolerate every possible change or sequence of changes, for three reasons. First, factored test are not expected to last forever; they will be regenerated periodically in any event. Second, non-tolerated changes can be discovered by a static or dynamic analysis, and the system test re-factored. Third, the changes to the code under test are not made by an adversary, but by a developer with a maintenance task in mind. Most of the changes are likely to come from a relatively small set of refactorings, and it is those that are worth considering.

The basic procedure of Section 3 assumes that the expectations and behavior of the environment depends on the order of all calls to mocked objects; none may be added, removed, or reordered. The remainder of this section proposes extensions to the change language handled by our test factoring to include three common and important functionality-preserving changes. A static or dynamic program analysis can indicate where the strict requirements of the transcript table can be relaxed. This reduces the number of false failures, generally without introducing false successes.

8.2 Reordering calls to independent objects

Sometimes, two objects from the environment are independent of each other's state, and calls to these objects can be intermixed in any order without affecting overall behavior. For example, in the Eclipse continuous testing plug-in [13], method `disableContinuousTesting` both deletes failure markers and removes a command from the launch configuration. These could have been done in either order, and a maintenance change that reorders them should not cause a test failure.

Our implementation addresses this problem by creating multiple transcript tables. One table per object would permit maximal reordering, but that would oversimplify in the other direction, treating every mocked object as independent. Instead, we group objects into *state sets*, forcing them to share the same transcript and `MockState`, if one is passed to the constructor for the other. This heuristic is unsound[3], but it has been surprisingly effective in practice. Further research will help to fine-tune it.

8.3 Adding or removing calls to accessors

Accessor methods, which do not change the receiver's state, may be added, deleted, and reordered during maintenance. For example, multiple calls to an accessor might be replaced with a single call to improve efficiency (this is the Replace Query With Temp refactoring).

To accommodate such changes, an analysis labels each method in the environment as read-only, write-only, or read-write with respect to each state set. A static analysis of the environment code would suffice, but we achieve additional precision via a dynamic analysis, by extending the trace to record reads and writes to the state of mocked objects. The transcript table is modified to neither fail nor advance the state when a read-only method is called.

8.4 Changing the order of simple mutators

Some classes have multiple options that must be set through individual setter methods before an action method can be called. For example, before a `Button` object in the SWT framework is displayed, methods `setText`, `setFont`, `addSelectionListener`, and `setSelection` may be called in any order. Reordering calls to these methods should not cause a factored test to fail.

To address this problem, the transcript object can be further modified to allow an unordered set, or *clump*, of method calls to be the trigger to advance to a new state, rather than changing state on every method call that is not read-only.

[3]Static analysis to compute the relation is difficult. For instance, a may-alias analysis would not be enough, since even if two objects are definitely not aliased, they can both reference a third object that is modified by calls to either of the first two.

Choosing the right method calls for advancing the state is important. In our example, methods that read-write tend to be important state-changing operations like `open`, `close`, or `addListener`. This suggests the heuristic of allowing write-only method calls to clump, but advancing the state on read-write method calls.

9. Related work and conclusion

Section 4.4 discussed the most closely related work: other approaches to instrumenting Java files for tasks such as capture and replay.

Mocking is closely related to stubbing, a well-known technique used in testing [4]. However, stubbing is manual; replaces a non-existent, rather than merely expensive, component (stubbing is a pre-implementation approach, and mocking is a post-implementation approach); and is more general: a stub may work with multiple tests, whereas a mock object asserts that it is used in specific ways.

Fowler [5] suggests that when a bug is discovered by a functional test, unit tests should be added that expose the same bug — this introduces a small best test for that bug, should it ever be reintroduced. Test factoring essentially performs exactly that procedure, automatically. The factored tests are useful for informing developers of test results more quickly, but can also be added as standalone unit tests. The greatest challenge is ensuring that the test results are meaningful and lead a developer to the proper error; since test factoring failures are exceptions thrown by the code under test, this should not be too difficult. As suggested in Section 8.1, the transcript file can be viewed as a scripting language for unit tests; we are investigating the possibility of having users edit the factored tests or write their own.

Test suites are rich artifacts in which developers embody substantial knowledge about a software system. This research continues a line of work, advocated by us and others, to mine these artifacts in order to extract useful information from them. Test factoring is just one approach to exploit existing test suites in order to make testing more effective. Related approaches are test selection [7, 6, 9], prioritization [16, 10, 15], augmentation (say, via coverage), etc.

Test factoring mines fast, focused unit tests from slow system-wide tests; each new unit test exercises only a subset of the functionality exercised by the system test. We have described a novel algorithm for test factoring that creates mock objects that simulate the behavior of part of the software system. We have also described other uses for the information, and how to adjust the algorithm to trade off resilience to code changes against false positives or negatives. To our knowledge, our test factoring implementation, which operates on Java programs of substantial size and complexity, is the first practical system for performing partial capture and replay of a Java program. Our experiments show that test factoring can significantly reduce the running time of a system test suite.

References

[1] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[2] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb. 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[3] M. Factor, A. Schuster, and K. Shagin. Instrumentation of standard libraries in object-oriented languages: The Twin Class Hierarchy approach. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 288–300, Vancouver, BC, Canada, Oct. 26–28, 2004.

[4] M. C. Feathers. *Working Effectively with Legacy Code*. Pearson Education, 2004.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[6] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[7] H. K. N. Leung and L. White. Insights into regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 60–69, Miami, FL, Oct. 16–19, 1989.

[8] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *Workshop on Dynamic Analysis (WODA)*, St. Louis, MO, USA, May 17, 2005.

[9] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, Aug. 1996.

[10] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct. 2001.

[11] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *Fourteenth International Symposium on Software Reliability Engineering*, pages 281–292, Denver, CO, Nov. 17–20, 2003.

[12] D. Saff and M. D. Ernst. Automatic mock object creation for test factoring. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'04)*, pages 49–51, Washington, DC, USA, June 7–8, 2004.

[13] D. Saff and M. D. Ernst. Continuous testing in Eclipse. In *2nd Eclipse Technology Exchange Workshop (eTX)*, Barcelona, Spain, Mar. 30, 2004.

[14] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 76–85, Boston, MA, USA, July 12–14, 2004.

[15] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 97–106, Rome, Italy, July 22–24, 2002.

[16] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Eighth International Symposium on Software Reliability Engineering*, pages 264–274, Albuquerque, NM, Nov. 2–5, 1997.