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.

Our implementation of automatic dynamic test factoring for the Java language reduces the running time of a system test suite by up to an order of magnitude.

Categories and Subject Descriptors: D.2.5 (Testing and Debugging): Testing tools

General Terms: Algorithms, Performance, Experimentation, Verification

Keywords: test factoring, mock objects, unit testing

1. Introduction

A focused test exercises only part of a system; for example, a unit test exercises one component (such as a single class) without relying on any other component. A test suite containing small, fast, focused tests has many benefits. 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. Focused tests are easy to prioritize and select from during regression testing because only a small fraction of them will exercise each individual changed component.

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 can be 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.

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* [13], for automatically creating 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 [3]. 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 (for example, 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

ASE'05, November 7–11, 2005, Long Beach, California, USA.

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 [6, 5, 10] runs only those tests that are possibly affected by the most recent change, and test prioritization [18, 11, 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 can be insufficient, because running even the ideal subset of the tests in the ideal order may be costly enough to prohibit running the tests with the ideal frequency. 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.

While we speak here of generating focused tests from whole-program, top-level system tests, test factoring can be applied generally to tests targeting any level of the system, including integration and component tests.

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 a case study using the tools. 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 by creating 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 only to specific queries.

Suppose that a software system is composed of two parts, T and E; we write the system as "T|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.

Consider, as an example, a payroll calculation system that depends on an expensive and slow employee database management system (see Figure 1). The actual payroll calculation algorithm is relatively fast, can be run on the developer's workstation, and changes frequently in response to business requirements. The database system is relatively slow, requires a dedicated hardware resource, and changes very infrequently. During regression testing, the payroll calculator only makes several dozen queries to a test database

[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.

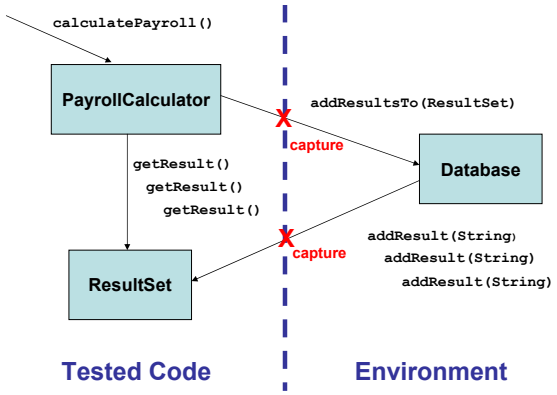

Figure 1: A small three-object software system for calculating payroll, with an example partition. The PayrollCalculator and ResultSet objects are in the code under test. The Database is in the environment. Boxes represent individual objects. Arrows represent references between objects, and are labeled with the method calls made through each reference. The central dotted line is the partition: all method calls across the partition, and only those calls, must be capturerd and replayed correctly.

in the database system, and the same results are always returned. In this system, developers would likely choose the payroll calculator as T, and the database system as E. The expected queries and results are the only aspect of the database system that the regression tests for the payroll calculator depend on.

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 T|E (that is, the original system), but on $T|E_m$, where E_m is a mock version of E. If 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 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 E'_m may be required. In our example, E_m need only return the correct results for the relatively few queries expected during testing to be a faithful mock of E (the database system).

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, 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.

2.1 Test factoring inaccuracies

Given an accurate transcript for E_m, it is fairly easy to ensure that a test executed on $T|E_m$ 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 T', it is desirable to test T'. If $\mathsf{T}'|\mathsf{E}_\mathsf{m}$ produces an answer ("pass" or "fail"), it should produce that answer faster than $\mathsf{T}'|\mathsf{E}$, but it is also possible for $\mathsf{T}'|\mathsf{E}_\mathsf{m}$ to produce a `ReplayException`.

A `ReplayException` indicates that the assumption inherent in the test factoring methodology — that 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 T' to E_m, or the arguments, are different than those that were captured during the training run of $\mathsf{T}|\mathsf{E}$ from which E_m was created. For example, if the payroll calculator from our example was changed so that it issued different queries against the database, the factored test would throw a `ReplayException`. Handling a `ReplayException` is straightforward: the full system $\mathsf{T}'|\mathsf{E}$ must be run — both to obtain a test result for T' and (in the background) to create a new mock object 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 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 procedure 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`.

Returning to our running example, Figure 2 shows how references to objects across the partition are decorated. Since the `ResultSet` is in the code under test, other objects in the code under test, like the `PayrollCalculator`, use undecorated references to the original object. However, objects in the environment, like the `Database`, only have references to a wrapper around the original object, which captures all interaction, and delegates to the original object to actually carry out the requested operations. Thus method calls on the `ResultSet` from the code under test are not recorded, but calls from the environment are. Likewise, calls to the `Database` from other objects in the environment (none are shown) are not recorded, but calls from the code under test

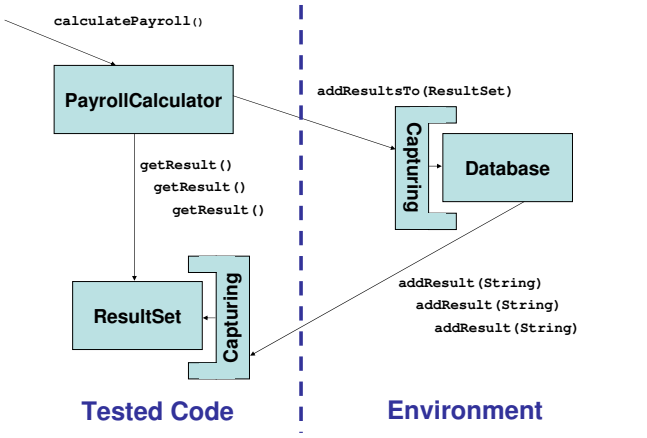

Figure 2: The payroll system, with references across the partition decorated for behavior capture.

are. Since all static references are to interfaces which are implemented by both the original classes and wrappers (see Section 4.2), the decision of which references are decorated can be completely made at runtime.

During replay, all calls to the `Database` that are made from the code under test will return the recorded value; no calls will be made on the `Database` from the environment, because the environment is replaced by mock objects.

Since the transcript records arguments and return values passing both directions across the partition, 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 program with alternative implementations. The capture step uses a wrapper 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, behavior-preserving step introduces a new interface for every class in the program, retrofits each class to implement its interface, and replaces references to the class by references to the interface. These interfaces separate type inheritance from implementation inheritance (which is useful for purposes 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 other possible 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

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__iface foo__iface(int x) { ... }
  void bar__iface(Date__iface d) { ... }
}
```

Figure 3: Example class before and after interface introduction.

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 the bootstrap library (file `rt.jar`). This procedure must be repeated for each new version of the Java runtime libraries — in our implementation, this is a fully automated procedure that takes about half an hour.

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.

4.2 Interface introduction

We wish to replace some object references in a test execution by references to different (capturing or replaying) objects. Making the new objects subclasses of the old would allow some client code to run unmodified, but this is problematic or impossible due to final classes and methods, reflection, and similar code constructs. It also makes it difficult to modify behavior only through some references to an object. Instead, we perform interface introduction: we change each class reference in the code into an interface reference, and change each class to implement the corresponding interface. The code can run with the original objects, but any other object that implements 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 3.

Interface introduction enables the behavior of method calls to be changed, via a program transformation that substitutes different objects that implement the interface but whose methods behave differently. This technique is incapable of 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 static hook routines that are set by the program transformation. Then, when new objects are introduced into the program to replace those, the hook routines can be set appropriately.

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 or program transformation

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 4 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 objects from the code under test are accessed directly, and all objects from the environment are accessed via instrumenting wrappers. The situation is symmetric when the program counter is in the environment. To maintain this invariant, whenever a reference to an object is passed from

[2]As a special case, built-in system classes, which are instrumented ahead of time rather than at runtime, 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 getReferenceTo(result);
  }
  void bar__iface(Date__iface d) {
    ... // record arguments (no results to record)
    _delegate.bar(getReferenceTo(d));
  }

  WeakHasccap<T, T__capturing> cc;
  T__iface getReferenceTo(T__iface in) {
    if (in instanceof T__capturing) {
      return ((T__capturing) in)._delegate;
    } else if (in instanceof T) {
      T real = (T)in;
      if (!cc.contains(real)) {
        cc.put(in, new T__capturing(real));
      }
      return cc.get(real);
    } else {
      throw new Error("this can't happen");
    }
  }
}
```

Figure 4: Capturing version of class `C` from Figure 3. Capturing wrappers are only created for classes in the environment. The crossover cache `cc` and `getReferenceTo` methods are shown parameterized on type T here–in the actual implementation, the necessary casts are inserted before and after each call.

the environment to the code under test (or vice versa), the reference is transformed — direct references to objects are replaced by references to instrumenting wrappers, and vice versa. For method parameters and return values, this is accomplished by the method `getReferenceTo` in Figure 4. Section 5 discusses how all non-method-call communication is either transformed into method calls or handled specially to maintain this wrapper invariant.

4.4 Alternative implementations

The Twin Class Hierarchy approach is designed specifically to permit instrumentation of Java standard libraries [2], by constructing a parallel, independent hierarchy that contains modified copies of each original class in the runtime. However, this approach does not scale. 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 alternative would be to instrument all objects (so no objects of the original type would appear anywhere in the system), rather than selectively introducing wrappers around some object references. The instrumentation could be enabled or disabled depending on whether the program counter is in the code under test or the environment. This approach could work (so long as no new fields were introduced), but introduces greater complexity, overhead, and potential for error than transforming references only at the boundary. 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. It would generate dependencies between the instrumented system classes and test factoring, whereas our approach merely performs interface introduction, allowing the instrumented system classes to be reused by other dynamic analyses. 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. Finally, universal replacement does not permit the "common libraries" optimization (Section 6).

A final approach would be to use debugger-based monitoring. Such an approach is orders of magnitude slower than ours, due to switching 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 is not acceptable.

Concurrently with us, Orso and Kennedy [7] 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 yet applied it to any programming tasks on changing software.

5. Complications in capturing

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 Field access

Before being run (during capture or replay), the program undergoes a semantics-preserving transformation that replaces static and instance field accesses by calls to generated field getter and setter methods, so that field accesses can be captured in the same way as method calls.

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, that occurs between the call and its return. 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 `getReferenceTo` method of Figure 4.

5.4 Arrays

The JVM treats arrays as a hybrid between objects and primitives. They subclass `Object`, and certain methods can 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 [2]; instead, our analysis 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 implementation of a native method is compiled C code that may depend on the exact type of its parameters. Thus, a call to a native method is wrapped with a helper method that casts the generated interface parameters back to their original class types, and then delegates to the actual code. This means that there is no way to capture interactions across the partition that occur from native code. However, in our experience, with a reasonable partition it is extremely rare for native methods to be called on objects passed across the boundary.

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 wrapping class loader around each class loader that the program (dynamically) creates. This wrapping loader uses its underlying loader to load original classes and performs any necessary interface introduction, capturing, or mock object creation before returning the result.

6. Common library optimization

As described so far, each class in a program is either part of the code under test or part of the environment. However, 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 environment, then any change to how a tested class uses `String`s (even internally) is likely to prevent replay. Furthermore, during replay, there is no performance benefit to replacing each `String` 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 common libraries. The common libraries part consists only of classes: each object that is instantiated from the common libraries is placed either in the code under test or environment, always the same part as the object that instantiated it. An object from a common library is transformed as it crosses the boundary just as described in Section 4.3. 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. It is up to the user of test factoring to guide this optimization by selecting good candidates for the common libraries part — these classes should be frequently used, computationally cheap, and not a likely source of bugs or change. For example, we often include the `java.util` collections classes in the common libraries.

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

Use of common libraries makes the instrumentation more complex, since only some objects of the class are captured and mocked. Use of common libraries also complicates issues of object equality in the presence of aliasing. However, our implementation handles these issues, and the results of the optimization are significant. For example, one factored test on Daikon (see Section 7) produces a 29,000 line transcript with the `java.util` collections in the common libraries. Explicitly declaring just `java.util.HashMap` as an environment class increases this to 54,000 lines, and slows replay time by 6%. Declaring `HashMap` as a tested class increases this to over 2.3 million lines, and replay requires much longer than the original system test.

7. Case study

This section reports an experiment measuring 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 (including third-party libraries) that implements sophisticated algorithms and makes use of Java features such as reflection, native methods, callbacks from the JDK, and communication via side effects. Daikon is a tool for detecting potential program invariants through dynamic analysis [1]. The code

	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 5: 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 working time the tests failed.

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 evaluates the technique considering both revision control logs and more frequent code snapshots. The revision control logs give us insight into known correct versions of the program (the Daikon developers ran tests before check-in), but involve larger sets of changed files, and longer times between invocations. Also, since the tests always pass, this 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. The code snapshots show 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, to catch real bugs.

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 during development. 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.

This paper reports on data from two developers (one professional and one undergraduate) working independently on Daikon between June 23 and August 20, 2004. Figure 5 summarizes the changes to each developer's copy of the program in the file system. Each moment corresponds to the developer saving (or otherwise modifying) a source file. We 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 [14] on the fly.

The revision control log indicates the changes that were checked into the CVS repository by all developers working on Daikon (not just the two most active developers, those whose file system actions were recorded). We arbitrarily chose to use the CVS data for 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 common 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 common libraries. This was a reasonable automatic heuristic, which changed over time as the developers focused on different parts of the code. In actual use, we would expect the developers, with some automated support, to make more informed decisions about partitioning, which could improve the results; future work will investigate that possibility.

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 (each automatically-generated partition was re-used an average of 20.7 times during our experiments) 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.

Some of the 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, incomplete code snapshots that happened to compile produced rare corner cases that exposed limitations in our prototype tool, preventing 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.

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 24 end-to-end system tests that exercise much more of Daikon. Running the regression tests from scratch takes about 60 minutes, but varies

with the speed of the workstation used. Running them with the `make` command, as all Daikon developers do, completes in about 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.

For the CVS experiments, we simulated running the tests after each CVS check-in, with and without test factoring. For the experiments based on snapshots from the two developers, we could not predict when developers might change their manual testing practices when given a new tool: as they gained experience, we would expect them to test more frequently, and to build intuition about effective times to test, but the user studies needed to support these intuitions are future work. Thus, as our baseline, we simulated continuous testing [12], which is an automatic technique that runs as many tests as possible as long as the code is in a compilable state. This is more frequent testing than any developer could do manually.

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 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.

We measure the following three quantities.

1. *Test time*, the amount of time required to run the tests.
2. *Time to failure*, 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.)
3. *Time to success* is the time between starting tests and successfully completing them. Successful test suite completion always requires running the entire test suite.

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 Applying test factoring

By adding factored tests to a test suite that expose the same errors as the large tests, test factoring seeks to reduce time to failure. When running the factored test suite, only the factored tests are run. If any factored test results in a replay exception, the corresponding system test is then rerun (but recapturing behavior is left until the following midnight). 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	Time to failure	Time to success
Dev. 1	.79 (7.4/9.4 min)	1.56 (14/9 sec)	.59 (5.5/9.4 min)
Dev. 2	.99 (14.1/14.3 min)	1.28 (64/50 sec)	.77 (11.0/14.3 min)
CVS	.09 (0.8/8.8 min)	n/a	.09 (0.8/8.8 min)

Figure 6: 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 ratios are better.

7.3 Results

Figure 6 provides experimental results. Test factoring generally reduces the running time, but it is influenced by the developer's working style. For the CVS data, the tests always succeed, so the time to success is always the same as the test time.

It is notable that when we considered code snapshots during development, although test time was reduced during successful runs, use of test factoring actually increased time to failure. The reason is that of the errors found running tests during development, most were simple coding errors or incomplete thoughts that would very quickly generate runtime errors or comparison failures in all of the system tests, and the Daikon developers had already performed manual test prioritization, moving the fastest system test to the front, resulting in a low time to failure without the tool. In these "incomplete" states, the behavior of the system often changes in drastic and unpredictable ways, very often causing a factored test to generate a `ReplayException`, forcing execution of the entire test, increasing time to failure (by adding the time to run the factored version, after which the test itself had to be run).

Measuring time to failure this way assumes that `ReplayExceptions` produce no valuable information to the developer. However, having observed that tests that eventually fail are much more likely to produce `ReplayExceptions` than tests that eventually succeed, future work can focus on helping developers use `ReplayExceptions` diagnostically, even while waiting for the final test results, or automatically detecting which tests do not really benefit from test factoring.

We did not systematically measure memory usage during experimentation. However, rerunning a small number of tests confirms that the factored tests consumed no more, and frequently much less, memory than their unfactored counterparts.

Daikon is a very difficult subject for test factoring. The fundamental problem is that Daikon 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 encouraging. We will collect data for additional programs to see whether, as we expect, they will prove more amenable to test factoring.

8. Future work: resilience to code changes

A factored test introduces assumptions about the implementation details of the functionality being tested; in our current implementation, if those assumptions are violated during program evolution, the factored test may return a `ReplayException`. We can generate a new accurate factored test by re-running the test factoring procedure, but in the meanwhile the factored test is useless for regression testing.

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.

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. This can be done by improving the algorithms for creating and running factored tests with sound analysis, or unsound heuristics that predict the likely result of the factored test. We are investigating this as future work — here, we outline some possible techniques.

A false success occurs when the factored test $\mathsf{T}'|\mathsf{E}_\mathsf{m}$ is unsoundly predicted to pass, but the system test $\mathsf{T}'|\mathsf{E}$ fails. A false failure occurs when the factored test is unsoundly predicted to fail, but the system test passes. 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.

- Factored tests in which false successes may occur 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.
- Factored tests in which false failures may occur 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.

8.1 Change language

A change language [13] 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 [4]. 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 tests 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 test factoring reapplied. 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 depend 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 two 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 [14], 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.

This problem could be addressed 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 could 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 initial investigations. Further research will help to fine-tune it.

[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.

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 [4]).

To accommodate such changes, an analysis can soundly label 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 can 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 can then be modified to neither fail nor advance the state when a read-only method is called.

9. Related work

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 [3]. However, stubbing is manual and more general: a stub may work with multiple tests, whereas a mock object asserts that it is used in specific ways.

Fowler [4] 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.

It may be possible to use test factoring as a complement or replacement of traditional GUI capture/replay tools [8, 15, 9, 17]. These tools record user interactions with an application's GUI, like key presses and mouse clicks, and store them in a script that can be replayed against future versions of the application. Test factoring's capture/replay model is more powerful than these tools, however, because it can capture interactions at any level: it is possible to ignore the individual user actions and capture only their effects on the underlying data model. We hope to try this application in future work.

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 [6, 5, 10], prioritization [18, 11, 16], augmentation (say, via coverage), etc.

10. Conclusion

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 case study shows that test factoring can significantly reduce the running time of a system test suite.

References

[1] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):1–25, Feb. 2001.

[2] M. Factor, A. Schuster, and K. Shagin. Instrumentation of standard libraries in object-oriented languages: The Twin Class Hierarchy approach. In *OOPSLA*, pages 288–300, Oct. 2004.

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

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

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

[6] H. K. N. Leung and L. White. Insights into regression testing. In *ICSM*, pages 60–69, Oct. 1989.

[7] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *WODA*, May 2005.

[8] T. J. Ostrand, A. Anodide, H. Foster, and T. Goradia. A visual test development environment for GUI systems. In *ISSTA*, pages 82–92, 1998.

[9] Rational Robot. `http://www-306.ibm.com/software/awdtools/tester/robot/`.

[10] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE TSE*, 22(8):529–551, Aug. 1996.

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

[12] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *ISSRE*, pages 281–292, Nov. 2003.

[13] D. Saff and M. D. Ernst. Automatic mock object creation for test factoring. In *PASTE*, pages 49–51, June 2004.

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

[15] SilkTest. `http://www.segue.com/products/functional-regressional-testing/silktest.%asp`.

[16] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA*, pages 97–106, July 2002.

[17] Mercury WinRunner. `http://www.mercury.com/us/products/quality-center/functional-testing/wi%nrunner/`.

[18] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *ISSRE*, pages 264–274, Nov. 1997.