Linguistic Support for Unit Testing

UUCS-07-013

Kathryn E. Gray
University of Utah
kathyg@cs.utah.edu

Matthias Felleisen
Northeastern University
matthias@ccs.neu.edu

Abstract

Existing systems for writing unit tests exploit built-in language constructs, such as reflection, to simulate the addition of testing constructs. While these simulations provide the minimally necessary functionality, they fail to support testing properly in many instances. In response, we have designed, implemented, and evaluated extensions for Java that enable programmers to express test cases with language constructs. Not surprisingly, these true language extensions improve testing in many different ways, starting with basic statical checks but also allowing the collection of additional information about the unit tests.

1. Testing Failure

Stories of catastrophic software failure due to a lack of sufficient testing abound. Proponents of test-driven development regale audiences with these stories to encourage developers to write adequate test suites. These stories, and the availability of testing frameworks such as JUnit, have motivated programmers across the board to develop basic unit test suites.

In writing unit tests, individual test cases should check that applicative methods (also known as observers) compute the expected results and that imperative methods (aka commands) affect the proper parts of the object's state (and nothing else). In addition, programmers need to be concerned with failures due to exceptions, especially ensuring that methods fail gracefully and as expected. While setting up tests for applicative methods tends to be straightforward, testing imperative methods and exceptional situations tends to require complex work. Specifically, it often demands calling a specific sequence of methods, and may benefit from performing a number of specific tests in sequence.

As a result, even with the wide-spread support for testing, programmers still don't develop sufficiently rigorous test suites, because creating and maintaining them remains a large burden. A thorough study of publicly accessible test suites (such as those in sourceforge) suggests that programmers construct few tests that check for modified state and even fewer (practically none) that exercise failure conditions.

We conjecture that a part of the problem in creating test suites lies with the lack of direct linguistic support for testing. This lack of testing constructs in the programming language itself has several symptoms, including silent failures and overly complex test case formulations. More precisely, our analysis shows that programmers fail to adhere to the protocol of the test suite, forgetting to prefix a method name with "test" or specifying formal parameters for testing methods when they take none. In such cases, the unit testing framework often simply ignores the tests without informing the programmer. Similarly, few (if any) programming languages allow the simulation of constructs that make it easy to set up exception handlers for tests of "exceptional" methods. Hence, programmers often don't test such scenarios or, if they do, it becomes difficult to maintain such tests due to the syntactic clutter.

A consequence of the lack of specific testing constructs is that compilers don't understand tests. Reflection constructs—the basis of JUnit—are simply too impoverished to communicate with compilers (properly). As a result, compilers don't gather enough information about the testing process. For failed test cases, information gathering makes testing truly rewarding; in contrast, lack of information makes it difficult to locate the source of bugs and to fix them. Put differently, on failure, testing tools should provide meaningful information on the actual vs desired behavior, the source of the failure, and information regarding the state of the program for the test. This kind of compiler-informed feedback from the testing tool would assist programmers in correcting errors quickly and economically.

To test our conjecture, we have designed and implemented an extension for Java, called TestJava, that includes constructs for testing. The compiler/IDE for TestJava gathers simple pieces of additional information to support error-

```
Def = ...
    | test Name [extends Name] [tests Name, ...] {
        Member ...}

Member = ...
        | testcase Name() { Statement ... }

Expr = ...
     | check Expr expect Expr [within Expr]
     | check Expr expect Expr by Comp
     | check Expr catch Name
     | Expr -> Expr
```

Figure 1. TestJava extensions

correction activities directly. Using TestJava, we can confirm that programmers can specify the intent of their test more concisely than in JUnit, meaning that the resulting program directly expresses the intent of the test. In turn, a compiler can statically analyze and extract information automatically, which users of JUnit must supply in strings or do without. More importantly still, our compiler also demonstrates how other analyses can be correlated to specific tests. Our TestJava compiler automatically inserts calls to a coverage analysis tool for each individual test call. This coverage analysis can then provide per-test coverage information without effort from the programmer. We believe that other analyses could benefit from similar information with limited effort, although we have not yet confirmed this part of our hypothesis.

In this paper, we present the design of TestJava, a comparison between TestJava and JUnit, preliminary experiences with the tool in undergraduate courses, and implementation guidelines. Section 2 presents the design of TestJava, along with emblematic examples. Section 3 presents a comparison of composing tests in TestJava versus JUnit, using examples based on deployed test cases (although the examples are not directly taken from any particular program). Section 5 presents a guide for implementing this language extension in Java. Section 6 presents our experience in implementing TestJava and providing expression-level coverage analysis.

2. TestJava

Supporting testing within the language requires two features: a means of writing testcases, and a means of grouping these testcases to form a rigorous test suite. We extend Java with four expressions that provide support for checking the results of individual computations, and with a new top-level form and new method-like form to group individual checks into unit tests with compile-time guarantees, seen in figure 1.

```
check $e_1$ expect $e_2$ ::
    evaluate $e_2$ to $v_2$
    safe-evaluate $e_1$ to $v_1$
    compare $v_1$ to $v_2$ (using deep equality)

check $e_1$ expect $e_2$ within $e_3$ ::
    evaluate $e_2$ to $v_2$, evaluate $e_3$ to $v_3$
    safe-evaluate $e_1$ to $v_1$,
    compare $v_1$ to $v_2$, using $v_3$ for tolerance

check $e_1$ expect $e_2$ by == ::
    evaluate $e_2$ to $v_2$
    safe-evaluate $e_1$ to $v_1$
    evaluate v1 == v2

check $e_1$ expect $e_2$ by Name ::
    evaluate $e_2$ to $v_2$
    safe-evaluate $e_1$ to $v_1$
    safe-evaluate $v_1$.Name($v_2$)

check $e_1$ catch name ::
    evaluate $e_1$ in catch Throwable $e$
        return e instanceof name
    return $false$

$e_1$ -> $e_2$ ::
  evaluate $e_1$ to $v_1$
  evaluate $e_2$ to $v_2$,
  return if ($v_2$ instanceof boolean) $v_2$ else $true$
```

Figure 2. Informal operational semantics

2.1 Expression forms

The four new expressions assess the correct behavior of evaluating one expression. In the three check expressions, the *Expr* immediately following the `check` keyword is the tested expression. The first expression, `check ... expect`, supports an optional third argument. The *Comp* item in the second expression, `check ... by`, is either a name or `==`. In the final expression, `->`, neither expression receives specialized treatment during evaluation, but information is extracted from both.

In all of the expressions, the test expression evaluates after all the other subexpressions. The `check` expression catches any exception thrown when evaluating the test expression or the comparison performed in a `check ... by` expression, while all other exceptions halt execution as normal. No `throws` clause or `catch` clauses are required due to test expressions. Figure 2 presents an informal semantics for each expression, using the function *compare*, which is explained in subsection 2.1.2, and the evaluator *safe-evaluate*, which catches all exceptions and returns an appropriate (non-expected) value.

Each `check` expression returns a boolean value, while placing different requirements on the types of the input expressions. Figure 3 presents the type rules for these expres-

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2 \quad t_2 \triangleright t_1 \quad t_1 \neq double, float}{\Gamma \vdash \texttt{check}\ e_1\ \texttt{expect}\ e_2\ : boolean}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2 \quad \Gamma \vdash e_3 : t_3 \quad t_2 \triangleright t_1 \quad t_3 \in Num}{\Gamma \vdash \texttt{check}\ e_1\ \texttt{expect}\ e_2\ \texttt{within}\ e_3 : boolean}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2 \quad t_2 \triangleright t_1}{\Gamma \vdash \texttt{check}\ e_1 \texttt{expect}\ e_2\ \texttt{by} == : boolean}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2 \quad t_1 : [boolean\ \texttt{Name}(t_3)] \quad t_2 \triangleright t_3}{\Gamma \vdash \texttt{check}\ e_1\ \texttt{expect}\ e_2\ \texttt{by Name} : boolean}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad n \succ Throwable}{\Gamma \vdash \texttt{check}\ e_1\ \texttt{catch}\ n : boolean}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \texttt{->}\ e_2 : boolean}$$

Figure 3. Type rules. $\succ$: subtype. $\triangleright$: castable.

sions. Since `check` expressions return booleans, tests can be logically connected using standard boolean operations.

2.1.1 Example expressions

The `check` expressions specify assessments of individual properties. In this section, we present example tests of a farm module for a hypothetical board game.

The basic `check` expression compares any two non-floating point values, as in the following expression.

```
check myFarm.harvestCrop() expect new Corn()
```

The call to the `harvestCrop` method is the test and the constructed corn object is the expected value. The `harvestCrop` method must return an object, where `Corn` can be cast to the specified return type. Evaluating this expression performs a built-in comparison of the returned object and corn, checking each field of the two objects. In this comparison, two objects with the same type and no fields are always equivalent.

Comparing floating point values requires the `within` sub-clause, so that an acceptable tolerance can be specified.

```
check myFarm.size() expect 5000.0 within .01
```

The `within` sub-clause can also specify an acceptable tolerance for fields with floating point values when comparing two objects. In this example, we store the size of a farm in a double field initialized by the constructor.

```
check myFarm.doubleAcerage()
  expect new Farm(10000.0) within .01
```

When comparing these farm objects, the floating point fields of the two objects must be within .01 of each other.

When neither the strict comparison of the straight `check` expression nor the tolerance of `within` suffice, the `by` sub-clause allows the programmer to specify a different comparison metric.

```
check myFarm.keepUp()
  expect myFarm.neighbors() by eqSize
```

The `by` clause must name either a method within the class of the tested expression or `==`. In this example, the `Farm` class contains a method accepting another `Farm` and comparing the two objects only by size. Standard methods, such as `equals` or `equalsIgnoreCase`, can also appear where appropriate. Object identity can be tested with `==`, as well as any values that `==` can traditionally compare.

```
check mybank.lookup(myid) expect myFarm by ==
```

In a `check ... expect ...` expression, if a test terminates in an exception, the cooperating test engine records the exception and the `check` produces `false`. When testing for intentional exceptions, the `check ... catch ...` expression records this expected behavior as well as the expected exception type.

```
check myFarm.sell() catch IllegalActionExn
```

This test produces `true` only if the `sell` method fails with an `IllegalActionExn` exception; normal termination or failing with a different exception type both produce `false`.

When a specific check relies on a sequence of actions, the `->` operator reports this dependence to the cooperating test engine, which includes the information in error reports. For example, a method that mutates a field can be checked with a comparison after the mutation.

```
myFarm.water() ->
   check myFarm.waterSupply expect 100
```

After performing the test, if the `waterSupply` does not match the expected level, the error report indicates that the call to `water` may have been faulty. The `->` operator can also specify tests that violate an expected protocol.

```
check (myFarm.rent() -> myFarm.sell())
  catch IllegalActionExn
```

This records that calling `sell` after `rent` should produce an exception, and if it does not, the error report indicates the specified protocol that is erroneously allowed.

Related tests can be grouped using standard conditional operators, allowing related tests that do not necessarily rely on one another to be grouped without impacting the error report.

```
check myFarm.sell() expect 100000) &&
check myFarm.owner() expect "Bank"
```

2.1.2 `check` ... `expect` ... [`within` ...] details

This form compares two values using a built-in comparison function. Evaluation requires a value in the expected position (and within position where applicable) before evaluating the test. The types of the test value and expected value must be castable to each other, as otherwise the two values cannot be equivalent. Floating point values may not appear without the `within` clause, although objects may contain floating point fields.

Comparison of primitive values uses `==`, while comparison of object values (including arrays) uses a function added to the runtime system. This operation deeply compares all fields of the two values, regardless of access privilege. Floating point fields are compared with a default tolerance of `0.0001`.[15]

Using `==` to compare floating point numbers is problematic; a minuscule rounding discrepancy can cause the test value to differ from the expected value. The difference between the two values may be acceptable for any given test; however, the allowable difference varies by application. Therefore, we require the programmer to specify this range using `within` when comparing only floating point values. The comparison of two floating point values uses `Math.abs(test-expected) < range`. The range is capped only by the numeric representation.

2.1.3 `check` ... `expect` ... `by` ... details

This form supports the specification of the comparison operation, which must either be a method within the test object's class or `==`. Any values comparable with `==` may occur when `==` is specified, in all other cases the values must be objects. The test expression's compile-time class must contain a visible method with the given name, which accepts the expected expression's compile-time type. As previously mentioned, exceptions thrown while evaluating the comparison are caught and the `check` expression produces false.

While the base *check* ... *expect* form could encode the behavior of this expression

```
check a.color().equalsIgnoreCase("blue")
   expect true
```

the encoding loses valuable information about the actual and expected values when the test fails. Further, the test engine cannot distinguish between an exception raised by the actual test and an exception raised by the comparison method as both appear to be the test.

2.1.4 `check` ... `catch` ... details

This form anticipates that a tested expression throws the specified exception. Neither the type nor the value of the test expression impact evaluation. A method call without a returned value (`void` methods) can appear in this position. The argument to `catch` must specify a visible class descended from `Throwable`.

When the test expression throws an exception, the thrown value must be a subtype of the declared exception type for the test to return `true`. Otherwise, including when the test expression completes evaluation normally, the test returns `false`.

2.1.5 ... `->` ... details

This final expression represents a sequencing action. The items to the left of `->` are expected to change the state, and the evaluation of the expression on the right depends on these effects. The types of the left and right-hand side of the expression are unconstrained, and any value on the left-hand side is ignored. The result of the `->` expression depends on the result of the right-hand expression. Any non-boolean value produces `true`, while a boolean result is immediately returned. Exceptions are not caught.

In the event the right-hand expression contains failed `checks`, the test engine knows that these computations depend on the left-hand expression. For example in

```
f.add('a') -> check f.next() expect 'a'
```

the compiler extracts information about the call to `add` as well as the call to `next` to report to the test engine. While this program is extensionally equivalent to

```
f.add('a');
return check f.next() expect 'a';
```

this second program fails to specify that the `check` test depends on the call to `add`. With this knowledge, the test engine can generate an error report that automatically informs the programmer that a failure in `f.next` may have occurred due to a failure in `f.add`. In the second program—a typical JUnit program—the programmer must manually generate this information in the form of print statements, adding to the already high burden of constructing test cases.

2.2 Unit test forms

A check expression tests the behavior of one expression against one desired outcome. To form a coherent test case, several check expressions may need to be grouped together. Similarly, a test suite consists of a group of test cases.

Existing unit testing implementations such as JUnit provide such grouping mechanisms by reusing existing language constructs, especially methods and classes. Due to this reuse, these testing systems cannot provide static guarantees about tests, test cases, and test suites and have difficulties relating information gathered at run-time to the source code of tests. In contrast, we provide linguistic grouping mechanisms for testing (see figure 1) that resemble classes and methods but come with restrictions beneficial to testing and test analysis. This section discusses the two major constructs.

2.2.1 Unit test examples

The top-level `test` form resembles a class and can contain any member. A test can specify the class (or classes) that it tests.

```
test Ownership tests Farm, Bank {
   Bank myBank = ...;
   Farm myFarm = ...;
   testcase purchase() {
      return myBank.buy(myFarm,"me") ->
            check myFarm.owner() expect "me";
   }
}
```

The new `testcase` member form resembles a method. The `testcase` declaration requires that the method returns a boolean and expects no arguments. No modifiers may appear on either form. If a `test` contains any constructors, it must contain a public constructor expecting no arguments. Forcing the `test` constructor and all `testcases` to expect no arguments assures that the test engine can instantiate and run all appropriate tests.

A `test` can extend an existing `test`.

```
test OwnerIdentity extends Ownership {
  boolean ownCheck(Bank b, Farm f) {
    return check b.lookup(f.id) expect f;
  }
  testcase ownership() {
    return ownCheck(myBank,myFarm) &&
           ownCheck(newBank,myFarm);
  }
}
```

As in standard inheritance, `OwnerIdentity` can access all visible fields and methods of `Ownership`, including `testcases`. The subclass can override both methods, which can be used to abstract tests, and `testcases`. A `test` also inherits all classes listed in a `tests` clause as classes it tests.

2.2.2 `test Name { ... }` details

The `test` form encapsulates a test suite, testing one or more classes. A `test` is not required to declare any tested classes or extend an existing test. Inheritance of `tests` is analogous to implementation inheritance for classes. Every `test` inherits two public methods, as well as the standard methods from `Object`. The two methods, `init` and `breakdown`, accept no arguments and return no values. During test execution, these methods are called before (`init`) and after (`breakdown`) running.

A `test` cannot specify any attributes, such as `abstract`, nor implement any interfaces. Tests comprise an inheritance hierarchy separate from those of both classes and interfaces. The standard `new` expression can construct an object from a `test`, and the resulting value acts like a standard object. A `test` can appear in any position that a `class` can appear, and follows standard visibility rules.

Visible classes and interfaces specified by a `tests` clause inform the test report system of the extent of the test. Our coverage analysis uses this information to report which methods of the specified classes are (and are not) covered by the current test.

2.2.3 `testcase Name() ...` details

The `testcase` form provides a collection point for individual `check` expressions, and follows the form of a method. A `testcase` may not specify any attributes nor any throws declarations. While `check` expressions automatically catch exceptions generated by test expressions, it is the programmer's responsibility to catch all other exceptions. These exceptions indicate a failure in the implementation of the test suite as opposed to a failure in the tested program, and as such should be brought immediately to the programmer's attention.

A `testcase` may only appear within a `test` body. Each `testcase` can be manually called following the standard method call procedure. Inheritance of a `testcase` is like implementation inheritance. A `testcase` may call its overridden predecessor using `super.Name()`; as in standard inheritance, without such a call, the overridden test case has no impact.

2.3 Information from testing

In order to provide information other than strict pass/fail results, our modified Java compiler (see Section 6) closely co-operates with the test execution engine. Specifically it provides three pieces of information to the test execution engine: first, it informs the test engine which `test` specifications to instantiate; second, it informs the engine which `testcases` to execute; and third, it provides in-depth details about each individual check.

After the compiler provides the initial information, the test engine uses the default constructor to instantiate each `test` in turn. The test engine can use the instantiated class to access `testcase` specific information encoded during compilation. The test engine then causes each `testcase` to execute. Inherited `testcases` run first, followed by the `testcases` of the current `test`. The sequence of test cases within a `test` determines the order of execution. An overridden `testcase` call occurs during calls to the parent methods, where the original method call appears. As the tests run, the generated code gathers information about `check` expressions and communicates it to the test engine.

A `test` succeeds when all its `testcases` succeed, which in turn succeed when the test case returns `true`. On success, the test report specifies the tests run, the successful test cases, as well as the results of coverage analysis when collected.

For failed `check` expressions, the test engine provides additional information. More precisely, the compiler provides the test engine with the source of the `check` expression, the expected and actual values and behavior, and the nature of the comparison. Additionally, the compiler sends along context information, including the method called, the fields accessed or mutated, the constructors called, etc. We are considering an additional extension so that the compiled code also gathers intermediate values generated during the test computation.

3. TestJava vs JUnit

Our linguistic approach to testing has several advantages over testing frameworks that use programming protocols instead. Most importantly, a linguistic approach improves both readability and maintainability, simply by reducing some of the mundane labor. Furthermore, the compiler can support testing with static checks and with run-time monitoring, which is especially important when a `check` fails.

In this section we present support for our thesis. Subsection 3.1 explains our data gathering method. The remaining subsections compare testing in our world with testing in a JUnit world on a point-by-point basis, a summary of this comparison is in figure 4. We believe that this comparison applies to other reflection-based systems, too. Only macro-based systems are comparable in quality with a testing framework based on language extensions [13, 16].

3.1 Methodology

In developing our TestJava extensions, we studied existing test suites in sourceforge to understand common practice. Our investigation only considered libraries written in Java that contained an open test suite of unit tests. Libraries meeting this simple criteria were chosen from different categories including puzzle games, board games, educational tools, interpreters, numerical analysis, internet support, and text editing. Test suites ranged in extent from those testing only one or two methods to those that appeared to cover 90% or more of the program.

During this investigation, we noticed that only 25% of tests for exception-throwing methods included calls causing the exception. And nearly half of these tests in our sample contained small errors that would obscure failure.

To analyze the benefits of TestJava, we present a set of emblematic tests in JUnit and then in TestJava. For readability, our tests cover a hypothetical board game implementation. All examples are presented using Java 1.4 and the corresponding JUnit version, as the examined libraries use these tools. Where more modern techniques address the problems encountered in a test, we additionally present a more modern JUnit implementation.

3.2 Test organization

Conceptually, a unit test system provides two levels of organization, first grouping individual tests that check one behavior (i.e. a function, method, or protocol) and then grouping these tests into larger suites that test related behaviors. This section presents the current practice of such organization, using JUnit test methods and test classes for the two levels, contrasted with the organization possible within TestJava, using `test` and `testcase`.

3.2.1 Test class organization

In most libraries, all test classes exist off a main test package directory. This package's internal hierarchy mirrors the package hierarchy of the implementation. Typically, one test class mirrors one implementation class. A few test classes implement tests that span multiple implementation classes, including tests over abstract base classes as well as conceptually similar but separate classes.

Using the `test` organization of TestJava, programmers can continue using the same test organization as before. The current practice provides a convenient means of indicating the (primary) tested class through the name of the test class. The JUnit system can extract the name of the test class, and thus inform the programmer, via reflection. Our system specifies tested classes directly, freeing the programmer to select a more informative naming convention or more convenient organization without losing information.[1] While the semantics of JUnit do not prohibit these changes, the lack of connection between the test and the implementation seems to discourage them.

3.2.2 Test method organization and abstraction

A typical test method contains either several different calls to one method, or a sequence of calls to several methods. In both circumstances, the name of the test method follows the pattern `testNAME`, where *NAME* refers to the primary method tested. The leading `test` dynamically alerts JUnit that the method is a test. In a more modern implementation, the prefix can be replaced with the `@Test` attribute although the information is still collected through reflection.

Often, the first line of a test method contains a call to display the name of the tested method in a string. These displays contain no information not found in the name of the method; however, their frequent presence indicates that the display provides information over that provided by the standard test run.

Using the `testcase` form, a test method can perform the same tests and follow a similar naming convention as in the updated JUnit form. The source of the test method is noted by the compiler to be included in the error report; this information may alleviate the need for the extra display information observed.

With the `testcase` form, the modifier and method signature are statically checked to ensure correct behavior while executing the test. It is possible for a misspelling to prevent a JUnit test case from running; however, less than 1% of observed methods contained obvious misspellings.

Non-test methods allow programmers to abstract common test calls. In both JUnit and TestJava these methods are unrestricted and can contain assertions or checks respectively. In many libraries, such methods were tagged with a `Test` prefix, and in one-third of these libraries at least one such method may have been intended to be an actual test.

When a method prefixed with `test` accepts parameters (which indicate it is not a JUnit test case), or when a method prefixed with `Test` accepts no parameters but appears to follow the convention for a test case, a programmer must analyze the code to deduce the original intent of the method (potentially requiring information from the original author). The `testcase` form statically distinguishes test methods from support methods, causing a compiler to signal violations that may have otherwise led to confusion.

[1] While a separate test package can simplify the process of bundling software, the specific test form can also inform a bundling process.

Test Organization	JUnit	TestJava	Benefits
Test class placement	Separate package hierarchy that mirrors implementation. One test class per implementation class.	Test classes in same package as implementation. One test class per task or implementation class.	`tests` have package access. Names describe conceptual organization.
Test method organization	Rely on unchecked name (or attribute) and type signature. Often included print out for identification aid.	Checked modifier and signature. Source automatically indicated.	Reduce static mistakes due to checking, and names can follow any style. Pinpoint source of failure.
Test abstraction	Typically identified using 'Test', potentially causing trouble with mistaken roles.	No restrictions, use of 'check' identifies source of method.	No possibility of confusion between support methods and tests.

Test Call Style	JUnit	TestJava	Benefits
Comparing primitives	Use assertEquals with identifying string.	Use `check ... expect ...`	Gain context information without additional work.
Comparing objects Full equality	Write equals, use assertEquals with identifying string.	Use `check ... expect ...`	Use equals for program-appropriate comparisons, gain context information without additional work.
Comparing objects No equals (arrays)	Write comparison, use assertTrue with identifying string	Use `check ... expect ...`	No comparison method to write. Error report contains actual and expected values.
Checking mutation	Write mutation call, sometimes with display. Compare mutated value using assertEquals with identifying string	Connect mutation to appropriate `check` with `->`	Mutation directly mentioned in error report. Tested calls indicated in source.
Checking for exceptions	Throw exception within `try ... catch ...`, use `fail` with identifying string. In modern JUnit, add exception information using class field to attribute property.	Use `check ... catch ...`	Concise call that indicates expectation. Interacts with sequences of tests.
Checking after exception	Throw exception within `try` block, check follow-up condition using `assert` and an identifying string in appropriate `catch` block.	Connect `check ... catch ...` and follow-up check with `->`	Explicit connection between expectation and check, gain context information without additional work.

Figure 4. Current Practice (JUnit) vs TestJava

3.3 Test call style

Individual tests either directly compare two values, or require a sequence of actions to prepare and execute the test. In performing individual components of a unit test, generated values should be checked, mutation should be confirmed, side-effects confirmed, and exceptional behavior guarded against as well as triggered.

3.3.1 Comparing primitives

Comparing primitive values in JUnit uses the `assertEquals` method, typically with a string given to provide information about the particular call.

```
assertEquals("size of farm", myFarm.size(), 500);
```

Due to overloading, an appropriate `assertEquals` method occurs for all primitive values as well as strings. An additional argument can be provided when comparing floating point values to set the tolerance between the two numbers.

Comparing primitive values in TestJava uses `check ... expect ...`, including the `within` clause where required.

```
check myFarm.size() expect 500
```

In the event this test fails, the error report contains information provided by the compiler to specify that the check called the Farm's size method with no arguments, as well as the source of the expression.

In both systems when an individual test fails, an error report announces the actual value received contrasted with the expected value and (conditionally in JUnit[2]) the source of the call. If an optional string is present, JUnit reports the value of the string as well, whereas TestJava always reports context information about the test call. This additional information provided by TestJava alleviates work for the programmer in specifying their test, in modification as well as creation, while providing assurance that each test will be identifiable on failure.

3.3.2 Comparing objects

When comparing two objects whose class contains an appropriate `equals` implementation, JUnit tests use the same `assertEquals` method described above, with the same resulting behavior.

Within our TestJava extension, the same test can either use the standard `check ... expect ...` expression or add the `by` clause if the default comparison does not perform the desired computation. The same benefits apply to this situation as with primitive values.

On occasion, the built-in equals method does not perform an appropriate comparison for the purpose of the test. In these circumstances (arising with arrays and classes without available source), the programmer cannot insert an appropriate comparison into the class. Therefore, the test call either uses a locally written comparison method or compares individual portions of the object one at a time.

For frequent comparisons, programmers create specialized local methods to perform a comparison and then use the `assertTrue` method to assess the response.

```
boolean comp(Crop[] a1, Crop[] a2) {
  boolean res = a1.length == a2.length;
  if (res)
    for(int i; i< a1.length; i++)
       res &= a1[i].equals(a2[i]);
  return res;
}

void testGetCrops() {
  System.out.println("getCrops");
  assertTrue(comp(myFarm.getCrops(3),
                  new Crop[]{new Rice(), ...}));
}
```

The `comp` method correctly assess whether the two `Crop` arrays are equivalent, and the assertion passes the result to the test report. On failure, the error report does not contain information regarding the actual and expected values, since these are not provided to JUnit. This implementation pattern typically contains a string explaining that comp is comparing an array created by the getCrops method.

Since the standard `check` expression does not rely on `equals`, the `comp` method above is unnecessary.

```
testcase getCrops() {
  return (check myFarm.getCrops(3)
          expect new Crops[]{new Rice(), ...})
}
```

The ability to leverage the default comparison saves the effort of writing specialized comparison methods for different objects.

Trying to compare two objects with private fields further highlights the default comparison's benefit. The programmer cannot write a method within their test to adequately compare two such objects; however, the default comparison is not limited by such restrictions.

Occasionally, when comparing two objects that do include an `equals` method, programmers still resort to checking individual pieces of two objects. Consider a method that instantiates a field value to a random number within a specified range, a typical comparison using `equals` may still require an exact match for this value while a test of the instantiation method cannot. Programmers use the floating-point comparison tolerance to specify a test for this situation.

```
Farm b = g.makeFarm();
assertEquals(b.area, 100.0, 50.0);
```

While the `makeFarm` method performs the relevant computation, the individual checks determine the subsequent appropriate values. Following this style, in contrast to writing a separate comparison method, the test report includes some of the expected and actual values.

The `within` sub-clause supports this comparison directly.

[2] JUnit relies on a source trace using the exception handler, if the Java compiler has not been configured to provide the source, the test will not report the location.

```
check g.makeFarm()
    expect new Farm(100.0) within 50.0;
```

The range value of the within carries into the fields of the Farm object and performs the comparison to the area field.

Following the piece-by-piece pattern, programmers risk omitting crucial fields from the local assertions. The risk of omission grows during a program's lifetime. If a programmer later adds a relevant field to the farm, they must remember to add an appropriate comparison into this test (and all tests that compare farms with individual field tests). By using the check expression, the programmer cannot omit a field even if the field did not exist when the test was originally written.

3.3.3 Checking mutation

A typical test of a mutation operation follows the same comparison style as that of the random farm generation described above.

```
Farm f = new Farm(500);
f.divide(100);
assertEquals(f.area, 400);
```

This test ascertains the performance of the divide method. If other method calls are necessary to properly establish the conditions before calling divide, it may become unclear whether those calls form part of the test or are tested in other places and form part of the framework.

A typical mutation method does not produce a value, and so does not produce anything visible to compare against. We address the lack of connection between the source of the mutation and tests concerning it through the use of the -> operator.

```
Farm f = new Farm(500);
return f.divide(100) -> check f.area expect 400;
```

This usage of -> provides a connection between the method that modifies the relevant value and the check of the relevant value. The auxiliary call to the farm constructor is not part of the test, as otherwise the assignment would be connected with an ->.

3.4 Checking exceptions

Typically, when testing a method that may throw an exception, programmers declare a throws clause for the method. Some test suite implementations specifically place method calls with a potential to throw exceptions inside of a try block to explicitly catch the exception and fail with a more informative error string.

Each check expression suppresses the need to catch or throw exceptions caused by the test position, so calling such a method requires no additional programming. Further, if several calls to such a method exist within one test, the specific call that caused the undesired exception is noted in the error report and subsequent calls may be able to continue (depending on the program logic connecting the expressions). This benefits programmers by providing the specific failure cite without requiring the addition of a try block.

In the libraries we studied, intentionally causing and checking for an exception requires the use of a try branch, which occurred with very low frequency. The body of the test call appears within the try, and appropriate checks or returns appear in the body of the appropriate catch. A fail appears in the remainder of the method.

```
try {
  myFarm.sell();
} catch IllegalAct( e) {
  return;
} fail("Exception not thrown");
```

A straight-forward usage of the check ... catch ... form produces the same effect without requiring that the programmer remember the return or provide the error message within the fail call.

The introduction of attributes with Java 1.5 also removes the necessity for a try statement in this situation.

```
@Test(expected = IllegalAct.class) void sell() {
  myFarm.sell();
}
```

This annotation informs the JUnit system to expect an exception to halt the execution of this test method, and that any other behavior is an error. However, for the best results in this interaction, the programmer is limited to one exception call per method.

Some protocols require that multiple exceptions be tested within one method, to ensure that proper side-effects occur during exception handling. To illustrate, the following code tests a protocol about tile placement on a game board.

```
Tile t1 = ..., t2 = ...; Coord c = ...;
b.placeTile(t1,c);
b.placeTile(t2,c);
try {
  b.placeTile(t1,c);
} catch( ContestedCoord e) {
  try {
    b.tileLoc(t2);
  } catch( UnplacedTile e) {
    return;
  }
  fail("UnplacedTile not thrown");
}
fail("ContestedCoord not thrown");
```

When a previously overplayed tile attempts to be placed in the same location, an exception occurs that then causes other interactions to fail. The original code this sample is based on placed the return statement at the end of the method instead of within the catch, but otherwise followed this protocol. The test written in a modified JUnit can avoid the inner try ... catch block, but must contain the first.

An equivalent test using a combination of check ... catch ... and -> avoids the use of try blocks at all.

```
Tile t1= ..., t2= ...; Coord c = ...;
b.placeTile(t1,c);
```

```
return b.placeTile(t2,c) ->
  (check b.placeTile(t1,c) catch ContestedCoord
  -> check b.tileLoc(t2) catch UnplacedTile);
```

The implementation using `check ... catch ...` eliminates the need to provide errors in the event of exceptions not occurring, eliminates the need for `try` blocks (thus eliminating the need for an oddly placed return), and can also continue to perform further checks within the method if needed or desired.

4. Preliminary Experience

In order to gather experience with TestJava, we have recruited two instructors—Viera Proulx at Northeastern University and John Clements at California Polytechnic State University (CalPoly)—of Java courses to introduce the testing constructs and to observe the reactions of the students. Both instructors teach students in second-term courses using the standard `check ... expect ...` form and the `within` sub-clauses for tests.[3] Both have used alternative constructs, including JUnit in prior semesters.

Overall both instructors enthusiastically reported improvements in students' behavior. Proulx used to use a graphical test specification system before progressing to JUnit specifications. In Fall '06, when she taught 30 students in a "catch-up" course, the graphical testing system was replaced with TestJava. She noted that students in this course wrote more tests than in previous years and wrote them with far fewer difficulties than before. Because of this success, she used the system again for the Spring '07 mainstream course. She continued to teach JUnit only because of its status in the rest of the Northeastern curriculum.

The CalPoly course presented Java programming to 25 second-quarter students. Like at Northeastern, the course began with students specifying their tests using TestJava and concluded with them writing JUnit specifications. Clements reported that the use of the `check` expressions worked well within the course and that the error reports students received using the TestJava extension significantly simplified the process of correcting errors when compared to the JUnit reports.

5. Implementation Guide

Implementing TestJava requires augmenting a Java compiler as well as integrating the resulting program with a test execution harness. The language requirements allow either a source-to-source compilation or a compilation to a traditional back-end. Any pairing of a test-engine with the compilation requires hooks connecting the compiled tests and the test-engine, as well as support for comparisons of any objects, regardless of access permissions.

This section outlines the general concerns of compiling the TestJava language, with specific attention to targeting Java source. This includes a discussion of both the test engine and comparison function, as well as implementing the different language features. We also outline the steps necessary to target JUnit as the test engine back-end. In section 6, we discuss our actual implementation, which targets the Scheme programming language.

[3] In accordance with our philosophy of restricting a programming language for introductory students [6, 10], only these forms are presented to the students.

Test Engine Interface

addToTest	TestClass	→	void
runningCheck		→	void
reportFailure	check, fail, call-info, SRC	→	void
reportDepends	SRC-l, call-l, SRC-r, call-r	→	void

check = One of "expect", "within", "by", "catch", "->"
fail = <Obj with actual, expected, range, thrown>
  thrown: boolean; range: Number or Boolean.

TestClass Interface

name	→	String
tests	→	String[]
testsSrcInfo	→	SRC[]
testCaller	→	CallIterator

CallIterator Interface

callNext	→	boolean
nextName	→	String
nextSrc	→	SRC

Figure 5. Test Engine interfaces

5.1 Connecting to a test engine

To obtain the full benefit of the TestJava extensions, the compiled version of each extension must connect to a test engine that drives execution and presents the results, and the compiled sources must provide call backs to support the engine. Figure 5 presents a suitable interface for both the test engine and the call backs.

The test engine contains four exposed methods; a method to register the test classes, a method to register that a check executed, a method to register a dependence between checks, and a method to register that a check has failed. Using the information provided to the first method, the test engine can instantiate and interact with each of the test objects.

Each test class must contain methods, not exposed for ordinary use, that the test engine can use to invoke the test methods and extract information regarding the current test class. This information includes, at a minimum, the name of the test class and the names of all classes declared in a tests clause. Calls to each `testcase` occur through the use of an iterator-like object. The test engine first extracts the name and source location of a test case and then executes the test, accumulating the number of tests that pass versus fail.

Calls to the check-related methods occur within the generated code for each check expression. These calls inform the test engine of the number of checks performed and, in the event of failure,

- where the check appeared,
- why the check failed,
- what form of check was tested,
- and the syntax of the initial test call.

An appearance of the `->` form causes a call to register the dependence, which can be used in generating reports of individual check failures.

5.2 How to compare values

Comparing all fields of two objects requires access privileges greater than are available in general. Two implementation techniques can provide a suitable means to write this comparison method. For security reasons, use of the comparison can be restricted to use within a `check` expression, which is already restricted to occurring within a test.

One technique relies on the compiler augmenting each compiled class with an additional method that performs the comparisons. The compiler ensures that each field in the current class is compared, and defers inherited fields using a `super` call. Each object field is compared using the same method. Primitive values are compared either with == or an appropriate comparison using tolerance. The base implementation of this comparison method, located in `Object` will first compare two values using ==, it will also store identifying information for previously encountered values to allow the comparison to terminate in the presence of cyclic data structures.

The compare method must not be given a name that can conflict with programmer specified methods. We make use of the different set of reserved words between the extended language and basic Java, namely `test` or `check`, to safely name the comparison method.

While the first technique assumes that all compiled Java classes have passed through an augmented compiler, a particular implementation may not be able to assure this due to legacy binaries interoperating with the program. To accommodate these cases, a comparison implementation using reflection or written natively may be required. With suitable JVM settings, even private fields may be exposed through reflection, and thus compared. Using this implementation, programmers may not be able to fully test their implementation on the JVM that the program will ultimately run on; however, this solution should be adequate in cases where the program source is unavailable.

5.3 Compiling tests

Since a `test` closely resembles a `public class` and a `testcase` closely resembles a method, these standard Java forms are the prime targets for compilation. At first glance, these translations are straightforward. A `test` translates directly into a `class`, and a `testcase` translates into a `public` method that declares a `boolean` return.

Each `test` includes two inherited methods that do not appear in `Object`. Therefore, each compiled `test` should inherit from one class containing the base implementations of these methods. This class should not be available for programmers to access in the pre-compiled Java system, so that while running a test suite, non-test classes cannot appear to be tests without the proper information. Additionally, each test class implements the interface outlined in figure 5. Each method from the interface must first call the super version of the overridden method.

5.4 Compiling checks

Each compiled `check` expression requires a translation that delays evaluation of the test expression, handles exceptions opaquely, and calls the engine's report methods. For the rest of this section, we assume the existence of a top level `compare` method that performs a deep comparison.

Wrapping the test subexpression for each `check` in a method call within an anonymous inner class implements the appropriate evaluation delay, while allowing the implementation of the remaining check expression to occur in a separate method implementation that is itself testable. Due to the treatment of exceptions within standard Java, the implementation must declare and handle the widest possible assortment of thrown exceptions.[4]

For the simplest `check` expression

`check` $testExpr$ `expect` $expExpr$

the translation results in an anonymous inner class wrapped around the $testExpr$ that extends a `checkExpect` nested class from the base testing class implementation. The test is conducted through a method within this base class and the compared value, $expExpr$ is passed in as an argument. So the expression becomes

```
new test.checkExpect() {
   Object test() throws Throwable {
     return testExpr; }
   String context() { ... testExpr ... }
   }.run(expExpr, srcLoc)
```

Within this translation, references in the first *testExpr* may require redirection if they are either (implicit or explicit) references to the original `this` or references to local variables that cannot be declared `final`. In the latter case, such variables can be lifted into an inner class referenced within the method. This reference can be declared final, and access to the variables within the remainder of the method and within the translation can be pointed to the correct indirection through the new local variable. Lifting these variables into their own class can permit easier memory separation between different test actions if desired with particular analyses.

[4] This section ignores the problem of discarding exceptions concerning JVM problems, such as out-of-memory exceptions, to keep the presentation focused on the principles.

The second appearance of *testExpr* within the `context` method represents the extraction of the syntactic expression for error reporting. This can either be the exact call the programmer wrote, a summary of the final method or constructor called along with the types of the arguments, or information regarding the initial values of objects and parameters located in this position. This information replaces the user-generated string often accompanying a JUnit assertion.

The definition of `run` calls the `test` method within a `try` block and then compares the values while reporting its activities to the test engine.

```
boolean run( Object expect, SRC src) {
  testEng.runningCheck();
  Object res = new Failure();
  try {
    res = this.test()
  } catch (Throwable t) {
    testEng.reportFailue(e,
                         fail(t, expect), ...);
  }
  boolean answer = compare(res, expect);
  testEng.reportFailure(e,
                        fail(res, expect), ...);
  return answer;
}
```

Where `Failure` is a private local class definition, and `fail` is a method that appropriately packages result information.

The translations for `check ... within` and `check ... by` close-ly follow the translation for the simple expression, with similar `run` definitions. For `check ... within`, the `run` method can be overloaded with suitable numeric second arguments including `int`, `float`, etc., where this argument is passed on to `compare`. The inner class for `check ... by Name` requires two additional methods, to call and identify the provided comparison. In addition, the class must contain two implementations of `run`, the first using == to compare the values and the second calling the comparison method within a `try` block in case the comparison throws an exception.

The `check ... catch` expression requires a translation that generates the `run` method, since Java does not support passing in types. So the translation for

`check` *testExpr* `catch` *givenThrowable*

produces an inner class with three methods, one to return the context, one to return a string of the expected exception's name, and one to perform the computation. The translation produces a call to the latter method.

```
new test.checkCatch() {
   boolean run() {
     testEng.runningCheck();
     try { testExpr; }
     catch ( givenThrowable e) {return true;}
     catch (Throwable e) {
       testEng.reportFailure(c,
                             fail(e, name()),...);
       return false; }
    testEng.reportFailure(...);
    return false;
   }.run()
```

Finally, translating `->` requires wrapping the calls to both expressions in an inner class. The `run` method within this inner class declares any appropriate throws clauses, and calls the `reportDepends` method. This method must be called twice, once to register the current dependence and once to indicate that the `->` expression terminated.

As an alternate implementation approach, that does not utilize anonymous inner classes, subexpressions involved in executing a `check` expression can be lifted to the first statement position above the expression's position. Each expression value can be stored in a temporary variable, with appropriate surrounding `try` statements around the initializations. Such an implementation technique has the advantage of limiting class allocations and inner class creation, but cannot be separately tested and cannot be used to isolate memory affects related to testing.

5.5 Targeting JUnit

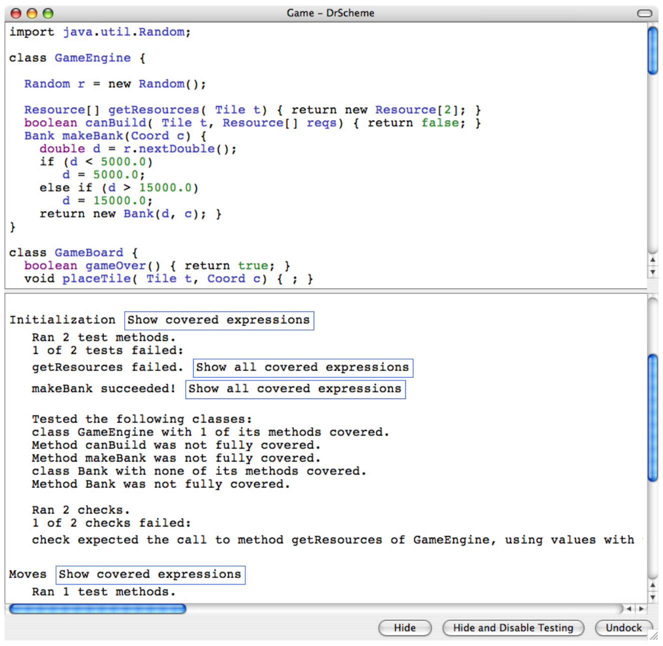
One possible target for the test engine is the JUnit system, where the compiler generates appropriate JUnit assertion calls. This would allow programs to mingle TestJava test suites with older JUnit test suites, as well as focus graphical support on one system.

In order to properly target JUnit, the base testing class must extend a proper JUnit testing class instead of `Object`. This base class must act as both the standard base class and as the integrated test engine, with a modified interface due to the reliance on JUnit. Within the JUnit `setup` and `breakdown` methods, the base class must display initial and summary information from the current class as well as contain a call to provide a full report on `check` behavior up to the current point. Each `testcase` method must append a `@test` attribute to the declaration site, but must return a boolean.

In implementing the `check` expressions, each `run` method must direct information to JUnit using string generation and `assertTrue`. The generated string must contain an appropriately formatted representation of the actual and expected behaviors. As this method operates by raising exceptions that halt execution when the actual value is `false`, these calls must be wrapped in a `try` statement allowing the program to return and produce `false`.

6. Available Implementation

Our implementation of TestJava extends ProfessorJ, our Java compiler for the DrScheme development environment [6, 11]. In this implementation, the Java constructs and the test additions are translated directly into Scheme code, which performs the role of Java bytecode; the translation is based on the design of section 5, though using closures instead of inner classes. Since all compiled code entering our system



```
import java.util.Random;

class GameEngine {

  Random r = new Random();

  Resource[] getResources( Tile t) { return new Resource[2]; }
  boolean canBuild( Tile t, Resource[] reqs) { return false; }
  Bank makeBank(Coord c) {
    double d = r.nextDouble();
    if (d < 5000.0)
       d = 5000.0;
    else if (d > 15000.0)
       d = 15000.0;
    return new Bank(d, c); }
}

class GameBoard {
  boolean gameOver() { return true; }
  void placeTile( Tile t, Coord c) { ; }
```

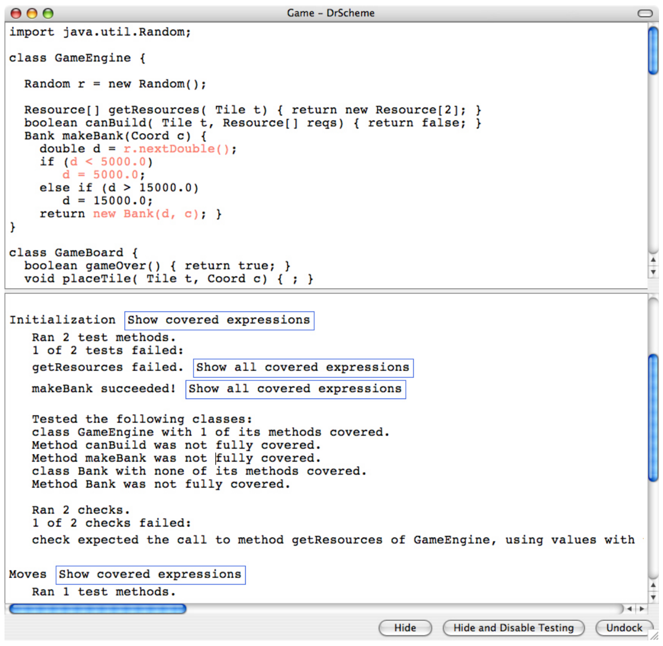
```
Initialization  Show covered expressions
   Ran 2 test methods.
   1 of 2 tests failed:
   getResources failed.  Show all covered expressions
   makeBank succeeded!  Show all covered expressions

   Tested the following classes:
   class GameEngine with 1 of its methods covered.
   Method canBuild was not fully covered.
   Method makeBank was not fully covered.
   class Bank with none of its methods covered.
   Method Bank was not fully covered.

   Ran 2 checks.
   1 of 2 checks failed:
   check expected the call to method getResources of GameEngine, using values with

Moves  Show covered expressions
   Ran 1 test methods.
```

Figure 6. ProfessorJ and test results



```
import java.util.Random;

class GameEngine {

  Random r = new Random();

  Resource[] getResources( Tile t) { return new Resource[2]; }
  boolean canBuild( Tile t, Resource[] reqs) { return false; }
  Bank makeBank(Coord c) {
    double d = r.nextDouble();
    if (d < 5000.0)
       d = 5000.0;
    else if (d > 15000.0)
       d = 15000.0;
    return new Bank(d, c); }
}

class GameBoard {
  boolean gameOver() { return true; }
  void placeTile( Tile t, Coord c) { ; }
```

```
Initialization  Show covered expressions
   Ran 2 test methods.
   1 of 2 tests failed:
   getResources failed.  Show all covered expressions
   makeBank succeeded!  Show all covered expressions

   Tested the following classes:
   class GameEngine with 1 of its methods covered.
   Method canBuild was not fully covered.
   Method makeBank was not fully covered.
   class Bank with none of its methods covered.
   Method Bank was not fully covered.

   Ran 2 checks.
   1 of 2 checks failed:
   check expected the call to method getResources of GameEngine, using values with

Moves  Show covered expressions
   Ran 1 test methods.
```

Figure 7. ProfessorJ & A partially uncovered class

derives from our test-aware compiler, all objects contain a comparison method (invisible to programmers).

Unless a programmer explicitly turns off testing, ProfessorJ runs all test suites, gathers information about their executions, and displays a summary window. Figure 6 presents the outcome of running a skeletal implementation of a board game program. The window in the top of the figure contains the program definitions and tests, the window at the bottom presents the test summary, including coverage information. In this instance, one test has failed and this display contains the following information concerning this test: static information from the call site; the actual values of the call site; the source of the call; and a hyper-link to the program text of the call site.

Following the steps outlined earlier, each `test` class is instantiated and then each `testcase` method is run. The testing tool collects the number of `testcase`s run, as well as the number that actually succeeded. The programmer also sees the specific outcome for each `testcase`. In the event of a failed `check`, the tool receives information collected during compilation from the comparison method.

We have extended our compiler with a simple per-expression coverage analysis for a proof-of-concept experiment; other extensions are easily combined with our testing constructs. When the coverage analysis is enabled, the compiler stores identifying information for each expression successfully executed during the program's run. At its coarsest grain, this allows the programmer to observe which expressions were covered for the entire test suite execution; however, information from the compiler allows finer-grained reports as well. In figure 7, expressions covered by one `testcase` instance are highlighted in a different color than the non-covered program. Coverage reports are available per `test`, `testcase`, and failed `check`. Additionally, this execution reports that a `test` did not fully cover all of the methods that it is declared to test (via the `tests` clause).

While we have chosen to track coverage with a per-expression metric, these language features are not limited to one form of coverage metric nor to only adding coverage analysis to the testing framework. The language extension and implementation provides the hook for further testing information and analysis; our particular choice demonstrates the possible benefits without significant implementation overhead within our system (although collecting per-expression coverage information on each expression evaluation does slow execution of the entire system, and so can be disabled).

7. Related Work

A fair number of programming languages come with library support for unit testing now. All of them are based on the principles first outlined with the SUnit [3] system for Smalltalk. This system provides utilities for building test suites, specifying individual test methods, and for writing down individual test expectations. The JUnit system [4, 8] is an implementation of these principles for Java.

JUnit supports testing with a library of assertion methods, derivable classes, and guidelines for establishing test suites. New tests extend a class from the JUnit library and provide individual test methods, indicated either with a `@Test` attribute[9] or by appending `test` to the name of the method. Individual test classes are optionally grouped by adding information about the tests into a test suite class. A graphical user interface allows testers to control the execution of individual tests and to monitor the execution and results. The JUnit library employs reflection throughout to accomplish its goals. Specifically it extracts all methods of the test class and examines their names and attribute fields to select the appropriate methods. Then, it uses reflection to execute the test methods and, where applicable, to check for the appropriate exception value instance. Programmers may document test cases with `String`-valued fields.

Three *x*unit testing libraries stand out from the rest: SchemeUnit for Scheme [16]; HUnit for Haskell [12]; and Lisp's LIFT system [13]. All of them contain features not found elsewhere and use a distinct implementation technology.

All three systems do closely resemble JUnit. Each provides a set of assertion functions to compare values, failure conditions result in exceptions, and programmers provide additional information in strings or symbols. Both the SchemeUnit and LIFT systems use macros to extend the language with a test form, eliminating the problem of failing to execute a test due to a typo in the name. In addition, in LIFT a new test expression can be dynamically added to a specific `deftest` at any time, allowing programmers to support a combination of grouped tests and tests located near function definitions. However, this feature can increase the difficulty in understanding and maintaining a test suite. Unlike the other *x*units, LIFT supplies a form that anticipates an error, although it does not distinguish between different kinds of errors.

HUnit uses Haskell's lazy-evaluation to its advantage. Expressions in test positions are not forced to evaluate until evaluation of the test function has begun. This allows hUnit to provide more specific information in the event of exceptions. Additionally, laziness allows tests and executing code to exist within the same Haskell module without causing additional overhead — tests evaluate only when an external call forces them to.

Assertions provide a means of determining whether procedures are performing according to expected parameters. Pre and post condition assertions provide support for checking these conditions at the entry and exit points of a procedure or method. While neither assertions nor pre and post conditions necessarily provide a means of testing, they can be used to assist in developing a test suite, especially when tools provide support to facilitate testing.

The Jass Java-extension [2] embeds a pre and post condition language into Java comments. If users provide a set of data, and a set of method calls to execute, the assertions confirm that the methods conform to the programmers expectations. Cheon and Leavens [5] also present an assertion language embedded into Java comments. This system uses the Java Modeling Language [14] to specify the method conditions, and then generates JUnit classes to check these properties around method calls. Users must still supply specific data, added into the JUnit classes.

Both of these systems, like hUnit, allow tests to appear with the tested procedures without negatively impacting standard execution. With the language extension embedded in comments, programmers do not get traditional editor support for writing their tests, and it becomes more difficult to abstract common testing behavior due to the placement and scope of the assertions. Further, as assertions are general conditions of a method behavior, using these properties to drive tests can exclude checks of specific program behavior on particular input. This can increase the difficulty in checking behavior on corner cases and conditional specific output based on input. The JML-JUnit system supports some conditions based on exceptions; however, these assertions suffer from the same problems for checking general output behavior.

The Fortress programming language [1] provides language support for specifying test procedures. An individual test case is marked using a `test` modifier, and a built-in library function provides functions to report and terminate failure conditions. A specific subset of tests can be combined into a test suite by using a library TestSuite object and inserting specific test procedures into this test suite. The body of a test may use an assert call, to verify that a specific test has produced a valid condition. While this language support provides static guarantees to programmers, similar to our `testcase` and `test` forms, the extension does not provide language-level support for representing the individual comparisons of a test, so that similar problems in specifying comparisons and considering exceptions arise.

The above systems provide means for developing test suites but they do not provide support for connection to additional testing analysis, such as coverage. The work of Gaffney *et al.* [7] demonstrates the difficulties in combining a traditional test execution tool with a coverage analysis tool, namely JUnit and Clover. One problem encountered was that the coverage tool did not distinguish between test execution coverage and program execution coverage. This led to more difficulty in interpreting the results. Such work illustrates the potential benefits of automatically providing information to an analysis tool via statically detectable differences between test calls and the actual program.

8. Conclusion

TestJava is only a first step toward a true and proper integration of testing with programming. The language provides constructs for expressing unit tests directly. Still, the experiment has demonstrated that such direct linguistic support for testing helps programmers with testing. Specifically, it facilitates writing down test cases, helps formulate them in a concise manner, and thus increase the chances that maintenance programmers can understand them and keep them up-to-date.

A good part of the increased value is due to increased compiler and run-time support that truly integrated testing constructs can enjoy. With a language extension, the compiler can detect (without speculation) which portions of a program are tests and can thus build in hooks for test analyses and other tools without requiring annotations or other external intervention. Our current implementation explores connecting a per-expression coverage analysis with the test language,

In the near future we intend to continue the exploration of TestJava in several directions. Most importantly, we wish to study how the programming environment—compiler, runtime library, and IDE—can take advantage of the integration but also what the benefits of this integration are.

One analysis that appears easy to add based on integrated testing constructs concerns mechanisms for tracing the execution of the methods of the tested classes. The check expressions already indicate where a particular call produces the correct or incorrect results. The tracing analysis could use this information to specifically target correct or incorrect traces.

Furthermore, simple instrumentation could determine those portions of a tested class that are exercised by a particular `test`. Storing this dependency information with which test-suites would then help the IDE select and compile test suites for regression testing when a particular class is modified.

Similarly, instrumentation from `check` and `->` could inform the IDE about the memory accesses of each test, which in turn could help the compiler roll back any effects for subsequent tests. With reflection-based libraries, such as JUnit, this kind of cross-phase information gathering and optimization is impossible. The programmer would have to supply significant amounts of information, which naturally isn't as reliable as automatically gathered information.

In summary our work has shown that extending the language to support testing does reduce the difficulties in writing test cases, permits easier reading of tests, and provide hooks for beneficial analyses based on test-suites. ProfessorJ, our version of Java with support for TestJava, is freely available at `www.drscheme.org` as part of the DrScheme development environment.

Acknowledgments

We thank Viera Proulx (Northeastern) and John Clements (CalPoly) for providing feedback on their experiences using TestJava in their courses. And we thank Robert Findler, Matthew Flatt, and the Cambridge CPRG for their helpful discussions.

References

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The fortress language specification. Technical report, Sun Microsystems, Inc., 2007.

[2] Detlef Bartezko, Clemens Fischer, Michael Moller, and Heike Wehrheim. Jass — Java with assertions. In *Workshop on Runtime Verification*, 2001. Held in conjunction with CAV'01.

[3] Kent Beck. Simple smalltalk testing with patterns. *The Smalltalk Report*, 1994. `http://www.xprogramming.com/testfram.htm`.

[4] Kent Beck. *JUnit Pocket Guide*. O'Reilly Media, 2004.

[5] Yoonsik Cheon and Gary T Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proc. European Conference on Object-Oriented Programming*, 2002.

[6] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, March 2002.

[7] Chris Gaffney, Christian Trefftz, and Paul Jorgensen. Tools for coverage testing: necessary but not sufficient. *Journal of Computing Sciences in Colleges*, 20(1), 2004.

[8] Erich Gamma and Kent Beck. Junit, testing resources for extreme programming. `www.junit.org`.

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005.

[10] Kathryn E. Gray and Matthew Flatt. ProfessorJ: a gradual introduction to Java through language levels. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 170–177, October 2003.

[11] Kathryn E. Gray and Matthew Flatt. Compiling Java to PLT Scheme. In *Proc. Scheme Workshop*, September 2004.

[12] D. Herington. hUnit. hunit.sourceforge.net.

[13] Gary King. LIFT — the lisp framework for testing. Technical Report 01-25, University of Massachusetts Computer Science Department, 2001.

[14] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *JML: A Notational for Detailed Design*, chapter 12. 1999.

[15] William Press, Brian Flannery, Saul Teukolsky, and William Vetterling. *Error, Accuracy and Stability*, chapter 1.3. Cambridge University Press, 1988.

[16] Noel Welsh, F. Solsona, and I. Glover. SchemeUnit and SchemeQL: Two little languages. In *Proc. Scheme Workshop*, 2002.