Reusing and Composing Tests with Traits

Stéphane Ducasse[1], Damien Pollet[1], Alexandre Bergel[1], and Damien Cassou[2]

[1] RMoD team, INRIA Lille – Nord Europe & University of Lille 1
Parc Scientifique de la Haute Borne – 59650 Villeneuve d'Ascq, France
[2] Phoenix team, University of Bordeaux 1
Building A29bis – 351, cours de la libération – 33405 Talence, France

Abstract. Single inheritance often forces developers to duplicate code and logic. This widely recognized situation affects both business code and tests. In a large and complex application whose classes implement many groups of methods (protocols), duplication may also follow the application's idiosyncrasies, making it difficult to specify, maintain, and reuse tests. The research questions we faced are (i) how can we reuse test specifications across and within complex inheritance hierarchies, especially in presence of orthogonal protocols; (ii) how can we test interface behavior in a modular way; (iii) how far can we reuse and parametrize composable tests.

In this paper, we compose tests out of separately specified behavioral units of reuse —traits. We propose *test traits*, where: (i) specific test cases are composed from independent specifications; (ii) executable behavior specifications may be reused orthogonally to the class hierarchy under test; (iii) test fixtures are external to the test specifications, thus are easier to specialize. Traits have been successfully applied to test two large and critical class libraries in Pharo, a new Smalltalk dialect based on Squeak, but are applicable to other languages with traits.

Keywords: Reuse, Traits, Multiple Inheritance, Tests, Unit-Testing.

1 The Case

One fundamental software engineering principle is to favor code reuse over code duplication. Reusing unit tests is important, because they are valuable executable specifications that can be applied to classes in different inheritance hierarchies. How can we factor tests and reuse them to validate the conformance of different classes to their common protocols? By protocol, we mean an informal group of methods, that is implemented orthogonally to inheritance hierarchies[1]. This is a typical situation where single inheritance hampers code reuse by forcing developers to copy and paste [4].

Still, there are some techniques to reuse tests: one way is to write unit tests that are parametrized by the class they should test. This way, it is possible to define tests for protocols (group of methods) and interfaces and to reuse

[1] We avoid the term *interface* because protocols are typically of finer granularity and implemented in a more flexible way, e.g., incompletely or with naming variations.

them for the different classes that implement such interfaces. Even if this does not seem widely used, JUnit 4.0 offers a parametrized test runner to run tests on a collection of fixtures returned by a factory method. Another approach is to define hook methods in the generic protocol test case, which can be overridden in a test case subclass for each implementor of that protocol. Both approaches have the limit that they organize tests per protocol rather than per implementor. This encourages to duplicate fixtures —which are indeed specific to the implementor— across protocols, but also makes implementor-specific adjustments difficult: for instance, to cancel inherited tests, subclasses may have to define phony ones[2].

Traits, pure composable units of behavior[8], recently got some attention and have been introduced in several languages including Squeak [12], AmbientTalk [7], Fortress [10], the DrScheme object layer [9], Slate and Javascript[3]. Traits are groups of methods that can be reused orthogonally to inheritance. A class inherits from a superclass and may be composed of multiple traits.

To evaluate the expressivity of traits on real case studies, several works redesigned existing libraries using traits [3,5]. So far, traits have been applied to and used when defining and designing applications. Since traits are composable units of behavior, the question whether traits may be successfully applied for testing naturally raises.

The research questions that drive this paper are the following ones:

– Are test traits reusable in practice? If each feature was tested by a separate trait, how much test reuse could we obtain? What is a good granularity for test traits that maximizes their reusability and composition?
– How far should a test fixture be adapted to specific requirements?
– How far should a test be parametrized to be reusable? What is the impact on the maintainability of test code?

To answer these research questions, over the last two years, we performed the following experiment: while designing a complete new implementation of the stream library of the Squeak open-source Smalltalk [5], we defined test traits (i.e., traits that define test methods, and which can be reused by composition into test classes). In addition, we initiated the development of a new body of tests for the collection library [6], in preparation for a future complete redesign. We defined a large body of tests and reused them using traits. In this paper, we report our results which are now part of the Pharo Smalltalk[4].

The contributions of this article show that traits are really adapted to specifying and reusing tests, especially in the context of multiple inheritance. A test trait defines an abstraction over its test data (a fixture), and a set of test methods. Test traits may then be composed and adjusted to build concrete test classes that can parallel the domain code organization.

[2] And canceling tests is not easily possible with the JUnit 4.0 approach.
[3] respectively `http://slate.tunes.org` and `http://code.google.com/p/jstraits`
[4] `http://www.pharo-project.org`

The article is structured as follows: Section 2 shortly presents xUnit, and expose the problems we are currently facing; Section 3 describes the libraries we selected for our study; we then explain our experimental process (Section 4) and present some selected examples and situations (Section 5); in Section 6, we present the traits we defined, and how they are reused to test the classes of the libraries; finally, we discuss the benefits and limits of our approach, before exploring related work and concluding (Sections 7, 8, 9).

2 Illustrating the Problem

In this section we briefly present the implementation principles of the xUnit frameworks, some examples of problems found in the Squeak libraries.

2.1 xUnit in a Nutshell

The xUnit family of testing frameworks originated with its Smalltalk incarnation: SUnit. In essence, a unit test is a short piece of code which stimulates some objects then checks assertions on their reaction [2,14]. The stimulated objects are collectively called a test fixture, and unit tests that have to be executed in the context of a given fixture are grouped with it into a test case. The framework automates executing the unit tests, ensuring each is run in the context of a freshly initialized fixture.

In SUnit, a test case is a subclass of TestCase which defines unit tests as methods, and their shared fixture as state. In the example below[5], we define such a subclass, named SetTest, with instance variables full and empty. We then define the method setUp, which initializes both instance variables to hold different sets as a fixture, and the unit test method testAdd, which checks that adding an element works on the empty set.

```
TestCase subclass: #SetTest
  instanceVariableNames: 'full empty'

SetTest >> setUp
  empty := Set new.
  full := Set with: 5 with: 6.

SetTest >> testAdd
  empty add: 5.
  self assert: (empty includes: 5).
```

[5] Readers unfamiliar with Smalltalk might want to read the code aloud, as approximate english. Message sends interleave arguments with keywords of the message name: receiver message: arg1 with: arg2 (the message name is message:with:). Messages are sent from left to right with priorities: unary, then operator, then keyword messages, so that self assert: 120 = 5 factorial does not need parentheses. A dot separates full statements, and semi-columns cascade several messages to a single receiver: receiver msg1; msg2. Single quotes denote 'strings', double quotes denote "comments". #foo is a symbol, and #(a b 42) is a literal array containing #a, #b, and 42. Curly braces are syntactic sugar for building arrays at runtime. Square brackets denote code blocks, or anonymous functions: [:param | statements]. The caret ^ result returns from the method.

2.2 Analyzing the Squeak Collection Library Tests

As explained above, in absence of parametrized unit tests, it is not simple to reuse tests across hierarchies (i.e., to run tests against classes in separate hierarchies). Indeed, the well-known limits of single inheritance apply to tests as well, and this is particularly true when we want to reuse tests for protocol compliance. Since protocol implementations often crosscut inheritance, a developer cannot simply rely on inheritance and is forced to either copy and paste or use delegation [8]. In addition to such common problems we wanted to understand the other problems that may arise when programmers cannot reuse tests. We studied the Squeak collection library tests and we complement the problems mentioned above with the following points:

Test duplication. We found redundancies in testing some features, not due to straight code duplication, but to duplicated test logic. The programmers probably were not aware of the other tests or had no means to reuse them. For example, the feature *"adding an element in a collection after a particular element in the collection"* is implemented with the add:after: method in LinkedList and OrderedCollection. This feature is tested in LinkedListTest and OrderedCollectionTest.
For example, compare the following tests:

```
LinkedListTest >> test09addAfter
  | collection last |
  collection := LinkedList new.
  last := self class new n: 2.
  collection add: (self class new n: 1); add: last.
  self assert: (collection collect:[:e | e n]) asArray  = #(1 2).
  collection add: (self class new n: 3) after: last.
  self assert: (collection collect:[:e | e n]) asArray  = #(1 2 3).

OrderedCollectionTest >> testAddAfter
  | collection |
  collection := #(1 2 3 4) asOrderedCollection.
  collection add: 88 after: 1.
  self assert: (collection =  #(1 88 2 3 4) asOrderedCollection).
  collection add: 99 after: 2.
  self assert: (collection =  #(1 88 2 99 3 4) asOrderedCollection).
```

Logic duplication is a problem since it limits the impact of tests and costs more in maintenance.

No systematic feature testing. In most of the cases, the collection tests were written to assess the compliance of Squeak with the ANSI standard. However, instead of an overall effort, the tests are the result of punctual and independent initiatives. The only way to assess which feature a test method covers, is by its name and by what messages the test method sends. As a consequence:

- some features are tested only for one particular type (e.g., after: and after:ifAbsent: are only tested for instances of OrderedCollection in SequenceableCollectionTest);

- some features are tested twice for the same type, in different classes (e.g., in both SequenceableCollectionTest and OrderedCollectionTest, testCopyWith tests the same copying features for OrderedCollection).

Testing ad hoc behavior. Since the tests were not reused, some tests ended up covering uninteresting implementation decisions, e.g., the default capacity of newly created collections. While it would be an interesting test if it was generic, applied class by class it often leads to ad-hoc values being documented in test methods. This practice can even be counter productive since tests that assume fixed values will break when the value changes, and require unnecessary fixes [14].

3 Experimental Context: Two Large Libraries

We experimented on two Smalltalk librairies structured around large hierarchies: streams and collections . We selected them for the following reasons: (i) they are complex and essential parts of a Smalltalk system, (ii) they mix subtyping with subclassing, (iii) they are industrial quality class hierarchies that have evolved over 15 years, and (iv) they have been studied by other researchers [13,6,11,4].

We identified the problems cited above in the Squeak open-source Smalltalk [12]. Squeak, like all Smalltalk environments, has its own implementation of such libraries which are solely based on single inheritance without traits. In Squeak, the abstract classes Stream and Collection have around 40 and 80 subclasses, respectively, but many of these (like Bitmap or CompiledMethod) are special-purpose classes crafted for use in the system or in applications. Here, we only take into account the core classes of each hierarchy (Figures 1 and 2).

3.1 Streams

Streams are used to iterate over sequences of elements such as sequenced collections, files, and network streams. Streams may be either readable (with methods like next and skip), or writable (with methods like write and flush), or both. There are different kinds of streams to read/write different kinds of data (characters, bytes, pixels. . .) and to iterate over different kind of sequences (collection, single file, compressed file, picture. . .). This multiplicity of properties implemented in a single inheritance object-oriented language involves either a combinatorial number of classes or trade-offs. Since the first solution is hardly realizable, all Smalltalk dialects chose the second approach with trade-offs (copy and paste, too many responsibilities, methods implemented too high in the hierarchy, unused superclass state) [5].

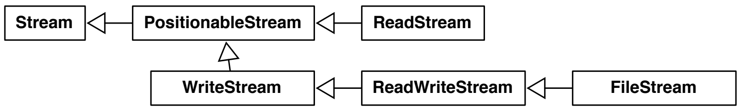

Fig. 1. The Squeak core stream hierarchy

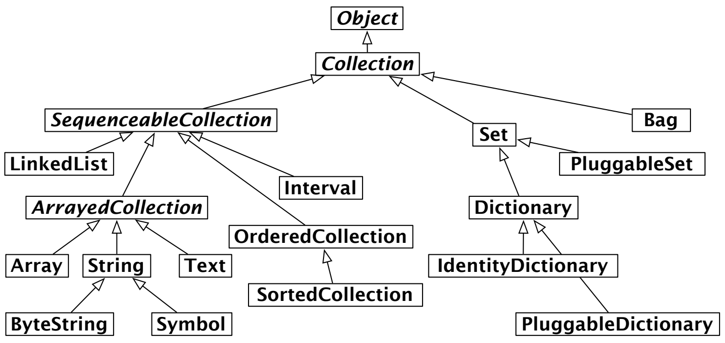

Fig. 2. Some of the key collection classes in Squeak

3.2 The Varieties of Collections

In Smalltalk, when one speaks of a collection without being more specific about the kind of collection, he or she means an object that supports well-defined protocols for testing membership and enumerating elements. Here is a summary. All collections understand *testing* messages such as includes:, isEmpty and occurrencesOf:. *All* collections understand *enumeration* messages do:, select:, collect:, detect:ifNone:, inject:into: and many more.

Table 1 summarizes the standard protocols supported by most of the classes in the collection hierarchy. These methods are defined, redefined, optimized or occasionally even cancelled in Collection subclasses[6].

Large set of different behaviors. Beyond this basic uniformity, there are many different collections. Each collection understands a particular set of protocols where each protocol is a set of semantically cohesive methods. Table 2 presents some of the different facets of collections and their implementation. For overall understanding, the key point to grasp is that protocols presented in Table 1 crosscut the large set of different behaviors presented in Table 2. Here is an enumeration of the key behaviors we can find in the Smalltalk collection framework:

Sequenceable: Instances of all subclasses of SequenceableCollection start from a first element and proceed in a well-defined order to a last element. Set, Bag and Dictionary, on the other hand, are not sequenceable.

Sortable: A SortedCollection maintains its elements in sorted order.

Indexable: Most sequenceable collections are also indexable. This means that elements of such a collection may be retrieved from a numerical index, using the message at:. Array is the most familiar indexable data structure with a fixed size. LinkedLists and SkipLists are sequenceable but not indexable, that is, they understand first and last, but not at:.

[6] Note that canceling methods in subclasses is a pattern that exists outside the Smalltalk world. The Java runtime cancels many methods in its collection framework to express immutability by throwing UnsupportedOperationException.

Table 1. Standard collection protocols

Protocol	**Methods**
accessing	size, capacity, at: *anIndex*, at: *anIndex* put: *anElement*
testing	isEmpty, includes: *anElement*, contains: *aBlock*, occurrencesOf: *anElement*
adding	add: *anElement*, addAll: *aCollection*
removing	remove: *anElement*, removeAll: *aCollection*, remove: *anElement* ifAbsent: *aBlock*
enumerating	do: *aBlock*, inject: *aValue* into: *aBinaryBlock*, collect: *aBlock*, select: *aBlock*, reject: *aBlock*, detect: *aBlock*, detect: *aBlock* ifNone: *aNoneBlock*
converting	asBag, asSet, asOrderedCollection, asSortedCollection, asArray, asSortedCollection: *aBlock*
creation	with: *anElement*, with:with:, with:with:with:, with:with:with:with:, withAll: *aCollection*

Keyed: Instances of Dictionary and its subclasses may be accessed by non-numerical indices. Any object may be used as a key to refer to an association.

Mutable: Most collections are mutable, but Intervals and Symbols are not. An Interval is an immutable collection representing a range of Integers. It is indexable with Interval>>at:, but cannot be changed with at:put:.

Growable: Instances of Interval and Array are always of a fixed size. Other kinds of collections (sorted collections, ordered collections, and linked lists) may grow after creation.

Accepts duplicates: A Set filters out duplicates, but a Bag will not. This means that the same elements may occur more than once in a Bag but not in a Set. Dictionary, Set and Bag use the = method provided by the elements; the Identity variants of these classes use the == method, which tests whether the arguments are the same object, and the Pluggable variants use an arbitrary equivalence relation supplied by the creator of the collection.

Contains specific elements: Most collections hold any kind of element. A String, CharacterArray or Symbol, however, only holds Characters. An Array will hold any mix of objects, but a ByteArray only holds Bytes. A LinkedList is constrained to hold elements that conform to the Link accessing protocol, and Intervals only contain consecutive integers, and not any numeric value between the bounds.

4 Experimental Process

Experience with traits [4,5] shows that they effectively support reuse. We identified the problems of Section 2 in Squeak, a dialect of Smalltalk. Our ultimate

Table 2. Some of the key behaviors of the main Squeak class collection

Implementation kind	
Arrayed	Array, String, Symbol
Ordered	OrderedCollection, SortedCollection, Text, Heap
Hashed	Set, IdentitySet, PluggableSet, Bag, IdentityBag, Dictionary, IdentityDictionary, PluggableDictionary
Linked	LinkedList, SkipList
Interval	Interval
Sequenceable access	
by index	Array, String, Symbol, Interval, OrderedCollection, SortedCollection
not indexed	LinkedList, SkipList
Non-sequenceable access	
by key	Dictionary, IdentityDictionary, PluggableDictionary
not keyed	Set, IdentitySet, PluggableSet, Bag, IdentityBag

goal is to redesign core libraries of Squeak, favoring backward compatibility, but fully based on traits. Our results are available in Pharo, a fork of Squeak that provides many improvements, including a first redesign of the stream library [5]. The current paper takes place in a larger effort of specifying the behavior of the main collection classes, as a preliminary of designing a new library.

Our approach to reuse tests is based on trait definition and composition. In both cases, our experimental process iterated over the following steps:

Protocol identification and selection. We selected the main protocols as defined by the main classes. For example, we took the messages defined by the abstract class Collection and grouped them together into coherent sets, influenced by the existing method categories, the ANSI standard, and the work of Cook [1,6].

Trait definitions. For each protocol we defined some traits. As we will show below, each trait defines a set of test methods and a set of accessor methods which make the link to the fixture. Note that one protocol may lead to multiple traits since the behavior associated to a set of messages may be different: for example, adding an element to an OrderedCollection or a Set is typically different and should be tested differently (we defined two traits TAddTest, TAddForUniquenessTest for simple element addition). Now these traits can be reused independently depending on the properties of the class under test.

Composing test cases from traits. Using the traits defined in the previous steps, we defined test case classes by composing the traits and specifying their fixture. We did that for the main collection classes, i.e., often the leaves of the Collection inheritance tree (Figure 2). We first checked how test traits would fit together in one main test class, then applied the traits to other classes. For example, we defined the traits TAddTest and TRemoveTest for adding and removing elements. We composed them in the test cases for OrderedCollection.

Then we created the test cases for the other collections like Bag, etc. However, the tests would not apply to Set and subclasses, which revealed that variants of the tests were necessary in this case (TAddForUniquenessTest and TRemoveForMultiplenessTest).

5 Selected Examples

We now show how we define and compose traits to obtain test case classes.

5.1 Test Traits by Example

We illustrate our approach with the protocol for inserting and retrieving values. One important method of this protocol is at:put:. When used on an array, this Smalltalk method is the equivalent of Java's a[i] = val: it stores a value at a given index (numerical or not). We selected this example for its simplicity and ubiquity. We reused it to test the insertion protocol on classes from two different sub-hierarchies of the collection library: in the SequenceableCollection subclasses such as Array and OrderedCollection, but also in Dictionary, which is a subclass of Set and Collection.

First, we define a trait named TPutTest, with the following test methods:

```
TPutTest >> testAtPut
    self nonEmpty at: self anIndex put: self aValue.
    self assert: (self nonEmpty at: self anIndex) == self aValue.

TPutTest >> testAtPutOutOfBounds
    "Asserts that the block does raise an exception."
    self should: [self empty at: self anIndex put: self aValue] raise: Error

TPutTest >> testAtPutTwoValues
    self nonEmpty at: self anIndex put: self aValue.
    self nonEmpty at: self anIndex put: self anotherValue.
    self assert: (self nonEmpty at: self anIndex) == self anotherValue.
```

Finally we declare that TPutTest requires the methods empty, nonEmpty, anIndex, aValue and anotherValue.

Methods required by a trait may be assimilated as the parameters of the traits, i.e., the behavior of a group of methods is parametrized by its associated required methods [8]. When applied to tests, required methods and methods defining default values act as customization hooks for tests: to define a test class, the developer must provide the required methods, and he can also locally redefine other trait methods.

5.2 Composing Test Cases

Once the traits are defined, we define test case classes by composing and reusing traits. In particular, we have to define the fixture, an example of the domain objects being tested. We do this in the composing class, by implementing the methods that the traits require to access the fixture. Since overlap is possible between the accessors used by different traits, most of the time, only few accessors need to be locally defined after composing an additional trait.

The following definition shows how we define the test case ArrayTest to test the class Array:

```
CollectionRootTest subclass: #ArrayTest
   uses: TEmptySequenceableTest + TIterateSequenceableTest + TIndexAccessingTest
      + TCloneTest + TIncludesTest + TCopyTest + TSetAritmetic
      + TCreationWithTest + TPutTest
   instanceVariableNames: 'example1 literalArray example2 empty collection result'
```

ArrayTest tests Array. It uses 9 traits, defines 10 instance variables, contains 85 methods, but only 30 of them are defined locally (55 are obtained from traits). Among these 30 methods, 12 methods define fixtures.

The superclass of ArrayTest is CollectionRootTest. As explained later the class CollectionRootTest is the root of the test cases for collections sharing a common behavior such as iteration. ArrayTest defines a couple of instance variables that hold the test fixture, and the variables are initialized in the setUp method:

```
ArrayTest >> setUp
   example1 := #(1 2 3 4 5) copy.
   example2 := {1. 2. 3/4. 4. 5}.
   collection := #(1 -2 3 1).
   result := {SmallInteger . SmallInteger . SmallInteger . SmallInteger}.
   empty := #().
```

We then make the fixture accessible to the tests by implementing trivial but necessary methods, e.g., empty and nonEmpty, required by TEmptyTest:

```
ArrayTest >> empty                    ArrayTest >> nonEmpty
   ^ empty                               ^ example1
```

TPutTest requires the methods aValue and anIndex, which we implement by returning a specific value as shown in the test method testAtPut given below. Note that here the returned value of aValue is absent from the array stored in the instance variable example1 and returned by nonEmpty. This ensures that the behavior is really tested.

```
ArrayTest >> anIndex                  ArrayTest >> aValue
   ^ 2                                   ^ 33
```

```
TPutTest >> testAtPut
   self nonEmpty at: self anIndex put: self aValue.
   self assert: (self nonEmpty at: self anIndex) = self aValue.
```

These examples illustrate how a fixture may be reused by all the composed traits. In the eventuality where a trait behavior would require a different fixture, new state and new accessors could be added to the test class.

The class DictionaryTest is another example of a test case class. It also uses a slightly modified version of TPutTest. This trait is adapted by removing its method testAtPutOutOfBounds, since bounds are for indexed collections and do not make sense for dictionaries. The definition of DictionaryTest is the following:

```
CollectionRootTest subclass: #DictionaryTest
    uses: TIncludesTest + TDictionaryAddingTest + TDictionaryAccessingTest
        + TDictionaryComparingTest + TDictionaryCopyingTest
        + TDictionaryEnumeratingTest + TDictionaryImplementationTest
        + TDictionaryPrintingTest + TDictionaryRemovingTest
        + TPutTest - {#testAtPutOutOfBounds}
    instanceVariableNames: 'emptyDict nonEmptyDict'
```

DictionaryTest uses 10 traits and defines 2 instance variables. 81 methods are defined in DictionaryTest for which 25 are locally defined and 56 are brought by the traits. For this class, a similar process happens. We define the setUp method for this class. Note that here we use a hook method classToBeTested, so that we can also reuse this test case class by subclassing it.

```
DictionaryTest >> setUp
    emptyDict := self classToBeTested new.
    nonEmptyDict := self classToBeTested new.
    nonEmptyDict
        at: #a put: 20;
        at: #b put: 30;
        at: #c put: 40;
        at: #d put: 30.

DictionaryTest >> classToBeTested
    ^ Dictionary
```

And similarly we redefine the required methods to propose a key that is adapted to dictionaries:

```
DictionaryTest >> anIndex                DictionaryTest >> aValue
    ^ #zz                                    ^ 33
```

5.3 Combining Inheritance and Trait Reuse

It is worth noting that we do not oppose inheritance-based and trait-based reuse. For example, the class CollectionRootTest uses the following traits TIterateTest, TEmptyTest, and TSizeTest (see Figure 3). CollectionRootTest is the root of all tests, therefore methods obtained from these three traits are inherited by all test classes. We could have defined these methods directly in CollectionRootTest, but we kept them in traits for the following reasons:

– Traits represent potential reuse. It is a common idiom in Smalltalk to define classes that are not collections but still implement the iterating protocol. Having separate traits at hand will increase reuse.

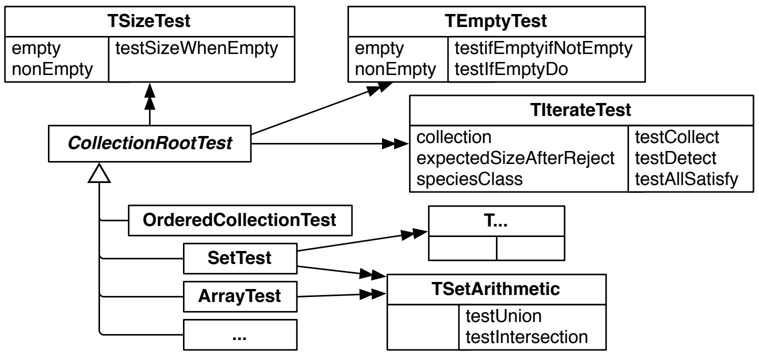

Fig. 3. Test reuse by both trait composition and inheritance (the representation of traits shows required methods on the left and provided methods on the right)

- A trait is a first class entity. It makes the required methods explicit and documents a coherent interface.
- Finally, since our plans are to design a new collection library, we will probably reuse these protocol tests in a different way.

6 Results

We now present the results we obtained by reusing tests with traits. Section 6.3 discusses the questions we faced during this work.

6.1 In the Nile Stream Library

Excluding tests, Nile is composed of 31 classes, structured around 11 traits: TStream, TDecoder, TByteReading, TByteWriting, TCharacterReading, TCharacterWriting, TPositionableStream, TGettableStream, TPuttableStream, TGettablePositionableStream, TPuttablePositionableStream. More details about the design of Nile may be found in the literature [5]. Nile has 18 test classes among which the ones listed on Table 3 use traits.

The columns of Table 4 and Table 6 describe:

Users: how many test case classes use each trait, either directly by composition or indirectly through both inheritance and composition;

Methods: the trait balance in terms of required methods vs. provided tests[5];

Ad-hoc: defined requirements or redefined methods vs. overridden tests;

Effective: the number of unit tests executed during a test run that originate from the trait in column 1.

For instance, TPuttableStreamTest is composed into 7 test case classes in total, through 4 explicit composition clauses; it requires 3 methods and provides 5 unit

[5] The small numbers indicate any additional non-test methods that the traits provide.

Table 3. Trait compositions in the Nile tests (only these 6 test classes use traits)

Test case class	Traits composed
CollectionStreamTest	TGettablePositionableStreamTest + TPuttablePositionableStreamTest
FileStreamTest	TGettablePositionableStreamTest + TPuttablePositionableStreamTest
HistoryTest	TGettableStreamTest + TPuttableStreamTest
LinkedListStreamTest	TGettableStreamTest + TPuttableStreamTest
RandomTest	TGettableStreamTest
SharedQueueTest	TGettableStreamTest + TPuttableStreamTest

Table 4. Use, structure, adaptation, and benefit of test traits in Nile

Test trait	Users	Methods[5]	Ad-hoc	Effective
	dir. / inh.	req. → prov.	mth. → tests	unit tests
TGettablePositionableStreamTest	2 / 4	1 → 2	4 → 0	8
TGettableStreamTest	5 / 8	1 → 10	8 → 0	80
TPositionableStreamTest	2 / 8	1 → 9	8 → 0	72
TPuttablePositionableStreamTest	2 / 4	0 → 1	0 → 0	4
TPuttableStreamTest	4 / 7	3 → 5 +1	22 → 0	35

tests and an auxiliary method. The test classes provide 22 definitions for its required methods: 3 requirements implemented in 7 test cases, plus 1 redefinition of the auxiliary method. Overrides are to be expected since the fixtures often can not be defined in a completely abstract way; none of the tests had to be adapted, though, which is good. In total, the 5 tests are run for each of the 7 test cases, so TPuttableStream generates 35 unit tests.

6.2 In the Collection Library

The collection hierarchy is far richer than the stream hierarchy. It contains several behaviors, often orthogonal, that are intended to be recomposed. As previously, Tables 5 and 6 describe how we composed and defined the test traits. The coverage of the collection hierarchy is in no way complete, and we expect to define other traits to cover behavior, like identity vs. equality of elements, homogeneous collections, weak-referencing behavior...

When writing the test traits, we decided to make do with the collection classes as they exist, so the traits are much less balanced than with Nile. For instance, TGrowableTest only applies to collections that reallocate their storage, so we tested it only for OrderedCollection. However this situation is due to a lack of time since several other collections exhibit this behavior. In contrast, TIterateTest and

Table 5. Trait composition in the collection hierarchy tests

Test case class	**Traits composed**
ArrayTest	TEmptySequenceableTest + TIterateSequenceableTest + TIndexAccessingTest + TCloneTest + TIncludesTest + TCopyTest + TSetAritmetic + TCreationWithTest + TPutTest
BagTest	TAddTest + TIncludesTest + TCloneTest + TCopyTest + TSetAritmetic + TRemoveForMultiplenessTest
CollectionRootTest	TIterateTest + TEmptyTest + TSizeTest
DictionaryTest	TIncludesTest + TDictionaryAddingTest + TDictionaryAccessingTest + TDictionaryComparingTest + TDictionaryCopyingTest + TDictionaryEnumeratingTest + TDictionaryImplementationTest + TDictionaryPrintingTest + TDictionaryRemovingTest + TPutTest − {#testAtPutOutOfBounds}
IntervalTest	TEmptyTest + TCloneTest + TIncludesTest + TIterateSequenceableTest + TIndexAccessingTest + TIterateTest − {#testDo2. #testDoWithout}
LinkedListTest	TAddTest − {#testTAddWithOccurences. #testTAddTwice} + TEmptyTest
OrderedCollectionTest	TEmptySequenceableTest + TAddTest + TIndexAccessingTest + TIncludesTest + TCloneTest + TSetAritmetic + TRemoveForMultiplenessTest + TCreationWithTest + TCopyTest + TPutTest
SetTest	TAddForUniquenessTest + TIncludesTest + TCloneTest + TCopyTest + TSetAritmetic + TRemoveTest + TCreationWithTest − {#testOfSize} + TGrowableTest + TStructuralEqualityTest + TSizeTest
StackTest	TEmptyTest + TCloneTest − {#testCopyNonEmpty}
StringTest	TIncludesTest + TCloneTest + TCopyTest + TSetAritmetic
SymbolTest	TIncludesTest + TCloneTest − {#testCopyCreatesNewObject} + TCopyPreservingIdentityTest + TCopyTest + TSetAritmetic − {#testDifferenceWithNonNullIntersection}

TEmptyTest are composed by nearly all test cases, so they have a much higher reuse. We also had to adapt composed tests more: while the fixtures are defined abstractly, they have to be specialized for each tested class and, sometimes, we excluded or overrode test methods in a composition because they would not work in this particular test case.

Table 6 shows that traits can achieve important code reuse. The presented results should also be interpreted with the perspective that the collection hierarchy is large and that we did not cover all the classes. For example, several kind of dictionary (open vs. closed implementation, keeping order) exist and were not covered. Therefore the results are definitively encouraging.

Table 6. Use, structure, adaptation, and benefit of test traits in the collection hierarchy

Test trait	**Users**	**Methods**[5]	**Ad-hoc**	**Effective**
	dir. / inh.	req. → prov.	mth. → tests	**unit tests**
TAddForUniquenessTest	1 / 1	3 → 4	3 → 0	4
TAddTest	3 / 4	3 → 7	10 → 1	28
TCloneTest	9 / 11	2 → 3	18 → 0	33
TCopyPreservingIdentityTest	1 / 1	1 → 1	1 → 0	1
TCopyTest	6 / 7	2 → 5	12 → 0	35
TCreationWithTest	3 / 3	1 → 7	3 → 0	21
TDictionaryAccessingTest	1 / 2	3 → 13	3 → 0	26
TDictionaryAddingTest	1 / 2	3 → 4	3 → 0	8
TDictionaryComparingTest	1 / 2	0 → 1	0 → 0	2
TDictionaryCopyingTest	1 / 2	3 → 2	3 → 0	4
TDictionaryEnumeratingTest	1 / 2	3 → 9	3 → 0	18
TDictionaryImplementationTest	1 / 2	0 → 8	0 → 1	16
TDictionaryPrintingTest	1 / 2	3 → 2 +1	3 → 0	4
TDictionaryRemovingTest	1 / 2	3 → 4 +1	3 → 0	8
TEmptySequenceableTest	2 / 2	3 → 6 +3	7 → 0	12
TEmptyTest	6 / 13	2 → 8	22 → 0	104
TGrowableTest	1 / 1	5 → 3	5 → 1	3
TIdentityAddTest	1 / 1	2 → 1 +1	3 → 0	1
TIncludesTest	8 / 10	5 → 6	41 → 0	60
TIndexAccessingTest	3 / 3	1 → 13	3 → 2	39
TIterateSequenceableTest	2 / 2	3 → 3	6 → 0	6
TIterateTest	2 / 9	7 → 20 +2	49 → 6	180
TPutTest	3 / 4	4 → 3	12 → 1	12
TRemoveForMultiplenessTest	2 / 3	1 → 1	2 → 1	3
TRemoveTest	2 / 4	2 → 4	6 → 0	16
TSizeTest	2 / 9	2 → 2	14 → 0	18
TStructuralEqualityTest	2 / 1	2 → 4	2 → 0	4

6.3 What Did We Gain?

In the introduction of this paper we stated some research questions that drove this experiment. It is now time to revisit them.

– *Are test traits reusable in practice? If each feature was tested by a separate trait, how much test reuse could we obtain? What is a good granularity for test traits that maximizes their reusability and composition?*

  We created 13 test classes that cover the 13 classes from the collection framework (Table 5). These 13 classes use 27 traits. The number of users for each trait ranges from 1 to 13. Globally, the traits require 29 unique selectors, and provide 150 test and 8 auxiliary methods.

  In the end, the test runner runs 765 unit tests, which means that on average, reuse is 4.7 unit tests run for each test written. If we do not count just tests but all (re)defined methods, the ratio to unit tests run is still 1.8.

Moreover, since the classes we selected often exhibit characteristic behavior, we expect that once we will cover much more cases, the reuse will increase.

- *How far should a test fixture be adapted to specific requirements?* We had to define specific test fixtures. For example, to test a bag or a set we need different fixtures. However, we could first share some common fixtures which were abstracted using trait required methods, second we could share them between several traits testing a given protocol. It was not our intention to reuse test fixtures optimally, though.

  To optimize the fixture reuse, we could have followed a breadth-first approach by collecting all the constraints that hold on a possible fixture (having twice the same element, being a character...) before writing any tests. However, this approach makes the hypothesis that the world is closed and that a fixture can be really shared between different test classes. We took a pragmatic approach and wanted to evaluate if our approach works in practice. In such a context, we had to define specific fixtures but we could share and abstract some of the behavior using required methods.
- *How far should a test be parametrized to be reusable? What is the impact on the maintainability of test code?*

  In the test traits, one out of 6 methods is required. Those unique 29 requirements lead to 237 implementations in the test classes to define the concrete fixtures, but often they are trivial accessors. The difficulty was in striking a balance between reusing fixtures constrained to test several aspects, or defining additional independent ones.

7 Discussion

Threats to validity. The collection hierarchy is complex and dense in terms of the represented behavior. As we showed in Section 3.2, collections possess many similar elementary facets: order and objects (OrderedCollection), order and characters (String), no order and duplication (Bag), no order and uniqueness (Set)... therefore we imagine that the potential of reuse is higher than in normal domain classes. Still we believe that lots of systems designed with interfaces in mind would benefit from our approach. For example, user interface or database mapping frameworks also often exhibit such multiple inheritance behavior.

Factoring test code or grouping test data? As said in the introduction, without traits, it is still possible to test protocols and interfaces orthogonally to classes, provided the test case are parametrizable, like in JUnit 4.0 (See Section 8.2). With this approach, test cases only test a specific protocol but are applied to each domain class that should respect that protocol. Alternatively, with the traditional approaches like JUnit, generic test code can be factored through inheritance or auxiliary methods.

The natural question is then what is the advantage of using traits vs. parametrized test cases. The JUnit scheme implies that generic test code must be grouped with the code that passes the test data, either all in one class, or in one hierarchy per tested protocol. Reusing generic test code is indeed a typical case of implementation inheritance: to group the tests for several protocols and one domain class together, one needs either multiple inheritance or delegation and lots of glue code. In contrast, with test traits defining the generic test code, it is possible to compose the tests for several protocols into a class that defines the test data for a single domain, thus nicely separating both generic and domain-specific parts of the tests code. Moreover, since the domain-specific code controls trait composition, it may ignore or redefine the generic test code on a case-by-case basis.

Traits with a single use. In the current system, some test traits are not reused; for instance, the ones dealing with specific OrderedCollection behavior (e.g., removing the last element, or inserting an element at a precise position) are only used by OrderedCollectionTest, so we could have defined the methods directly in this class. Whether it makes sense or not to have such specific protocol tests grouped as a trait is a design question —which is not specific to test traits but applies to using traits in general and to interface design.

We believe that it still makes sense to define such cohesive test methods as a trait. The current collection library has a large scope but misses some useful abstractions; even if a trait is not reused currently, it is still a potentially reusable cohesive group of methods. For example, there is no collection which simultaneously maintains the order of elements and their uniqueness; if a new UniqueOrderedCollection is implemented, test traits specifying the ordering and uniqueness protocols will make it easy to ensure that UniqueOrderedCollection behaves like the familiar Set and OrderedCollection.

The case of implementation or specific methods. When writing test traits we focused on public methods. In Smalltalk, since all methods are public, it is difficult to know if one should be covered or not. Methods may be categorized into three groups: (i) implementation and internal representation methods, (ii) methods in a public interface specific to a given collection, and (iii) unrelated method extensions. We think that tests for the first category are not worth reusing and should be defined locally in the particular test case. As mentioned above, we believe it is worth to create a trait for the second group because it proactively promotes reuse. The last group of methods are convenience methods, so they belong to a different library and should be tested there. For example, OrderedCollection >> inspectorClass should be covered by the tests for the GUI library —where it belongs.

Benefits for protocol/interface design. Our experience trying to identify common protocols highlighted a few inconsistencies or asymmetries; for instance OrderedCollection >> ofSize: and Array >> new: both create a nil-initialized collection of the given size, but OrderedCollection >> new: creates an *empty* collection that can grow up to the given *capacity*. We imagine that these inconsistencies could appear because the tests were not shared between several classes. Since protocol tests are easily reused specifications, we believe they support the definition of more consistent interfaces or uniform class behavior.

Designing traits for tests vs. for domain classes. When designing traits for domain classes (as opposed to test traits), we often found ourselves defining required methods that could be provided by other traits. The idea there is that the method names act as join points and glue between traits, propagating behavior from one trait to another.

However, in the context of this work, it is better to design traits with required methods named to avoid accidental conflicts with methods of other traits. This way, it is easier to provide specific values and behavior in the context of the specific collection class under test. If a value or behavior must be shared, a simple redirection method does the trick.

Pluggable fixtures and test traits. To support reuse between test traits, we tried to share fixtures by making them pluggable. However, it was sometimes simpler to define a separate fixture specific to a given protocol. Indeed, each protocol will impose different constraints on the fixture, and devising fixtures that satisfy many constraints at the same time quickly becomes not practical. It is also important to understand how far we want to go with pluggability; for example, we did not use the possibility to execute a method given its name (which can be trivial in Smalltalk using the perform: message). We limited ourselves to use required methods to change collection elements, because we wanted to favor readability and understandability of the tests, even at the cost of potential reuse.

8 Related Work

8.1 Inheritance-Based Test Class Reuse

The usual approach to run the same tests with several fixtures is to write the tests in terms of an abstract fixture and to redefine setUp to provide various concrete fixtures. This approach can be extended with Template/Hook Methods to organize and share tests within a class hierarchy; for example, the Pier and Magritte frameworks [15] use this approach. Their test suites total more than 3200 SUnit tests when run, for only about 600 actual test methods defined; it is then interesting to understand the pros and cons of the approach.

The principle is the following: the test hierarchy mirrors the class hierarchy (e.g., MAObjectTest tests MAObject). Some tests are abstract or just offer default values and define hook methods to access the domain classes and instances under test (e.g., a method actualClass returns the domain class to test, and instance and others return instances under test). This approach allows one to test different levels of abstraction. In the leaves of the hierarchy, the tests defined in the superclass can be refined and made more precise. While this approach works well, it shows the following limits:

- Sometimes, test methods have to be cancelled in subclasses.
- When the domain exhibits multiple inheritance behavior, this approach does not support reuse: it exhibits the same limits as single inheritance in face of need for multiple inheritance reuse. Indeed, besides copy/paste or delegation, there is no specific mechanism to reuse tests.

This approach may be improved to reuse protocol tests. The idea is to define one test case class for each protocol and define an abstract fixture. Then for each of the classes implementing the protocol, a subclass of the test case class is created and a fixture is specifically defined. The drawback of this approach is that the fixture has to be duplicated in each of the specific subclasses for each protocol.

8.2 Parametrized Test Classes in JUnit 4.0

JUnit 4.0 provides a way to parametrize a test class intended to run a set of test cases over a different data. For example, the code excerpt given below shows a test class that verifies the interface IStack for two implementations. The method annotated with @Parameters must return a collection of test data objects, in this case instances of the stack implementations JavaStack and ArrayStack. The Parameterized test runner instantiates the test class and run its tests once for each parameter.

```
@RunWith(Parameterized.class)
public class IStackTest {
  private IStack stack;
  public IStackTest(IStack stack) { this.stack = stack; }

  @Parameters public static Collection<Object[ ]> stacks() {
    return Arrays.asList(new Object[ ][ ] {
      { new ArrayStack() },
      { new JavaStack() } });
  }
  @Test public void newStackIsEmpty() throws Exception {
    assertTrue(stack.isEmpty());
  }
}
```

When the fixtures are easy to build, the parametrized test runner is thus a very convenient alternative to the usual xUnit approach of redefining setUp in subclasses. However, this is not so clear if we consider complex fixtures composed from several interrelated domain values, like we needed in our tests, and the @Parameters annotation introduces a dependency from the test code to the implementations: to test a new IStack implementor, one must modify IStackTest — or subclass it, which is the non-parametrized approach.

In contrast, test traits group the protocol-level test code in an independent entity, while test classes encapsulate details specific to the tested implementation like building the fixture. Especially, test classes control the composition of tests from several traits, and can define additional ad-hoc tests. This enables a workflow where the protocol can be documented by a generic test trait, and implementation tests can be organized in test classes that parallel the domain class hierarchy.

9 Conclusion

Single inheritance hampers code reuse of business and testing code. Currently, programming languages lack constructs that help reuse across different unit tests. We propose test traits, an approach based on traits which reduces code duplication and favors composition of feature testing code. Our approach is particularly adapted to test protocols in class hierarchies with many polymorphic classes, and is applicable to other trait-based languages. We applied test traits to two large and critical Smalltalk libraries; in average we reused each test 4.7 times. This experiment shows definitive advantages of our approach: test reuse across several classes, test composition, simple fixture parametrization. We will pursue our effort to fully cover the collection hierarchy, and we expect to get higher test code reuse.

References

1. ANSI. American National Standard for Information Systems—Programming Languages—Smalltalk, ANSI/INCITS 319-1998 (1998)
2. Beck, K.: Simple Smalltalk testing: With patterns, `http://www.xprogramming.com/testfram.htm`
3. Black, A.P., Schärli, N.: Traits: Tools and methodology. In: ICSE (2004)
4. Black, A.P., Schärli, N., Ducasse, S.: Applying traits to the Smalltalk collection hierarchy. In: OOPSLA, vol. 38, pp. 47–64 (2003)
5. Cassou, D., Ducasse, S., Wuyts, R.: Traits at work: the design of a new trait-based stream library. Journal of Computer Languages, Systems and Structures 35(1), 2–20 (2009)
6. Cook, W.R.: Interfaces and specifications for the Smalltalk-80 collection classes. In: OOPSLA, vol. 27, pp. 1–15. ACM Press, New York (1992)
7. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-oriented programming in ambientTalk. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 230–254. Springer, Heidelberg (2006)
8. Ducasse, S., Gîrba, T., Wuyts, R.: Object-oriented legacy system trace-based logic testing. In: European Conference on Software Maintenance and Reengineering (CSMR 2006), pp. 35–44. IEEE Computer Society Press, Los Alamitos (2006)
9. Flatt, M., Finder, R.B., Felleisen, M.: Scheme with classes, mixins and traits. In: AAPLAS (2006)
10. The Fortress language specification, `http://research.sun.com/projects/plrg/fortress0866.pdf`
11. Godin, R., Mili, H., Mineau, G.W., Missaoui, R., Arfi, A., Chau, T.-T.: Design of class hierarchies based on concept (Galois) lattices. Theory and Application of Object Systems 4(2), 117–134 (1998)
12. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: The story of Squeak, a practical Smalltalk written in itself. In: OOPSLA, pp. 318–326. ACM Press, New York (1997)
13. LaLonde, W., Pugh, J.: Subclassing $\neq$ Subtyping $\neq$ Is-a. Journal of Object-Oriented Programming 3(5), 57–62 (1991)
14. Meszaros, G.: XUnit Test Patterns – Refactoring Test Code. Addison-Wesley, Reading (2007)
15. Renggli, L.: Magritte — Meta-described web application development. Master's thesis, University of Bern (2006)