BLOC: a Trait-Based Collections Library – a Preliminary Experience Report

Tristan Bourgois, Jannik Laval, Stéphane Ducasse, Damien Pollet

BLOC: a Trait-Based Collections Library – a Preliminary Experience Report

Tristan Bourgois Jannik Laval Stéphane Ducasse Damien Pollet

RMoD Project-Team, Inria Lille – Nord Europe — Université de Lille 1 — CNRS UMR 8022

tr.bourgois@laposte.fr, {jannik.laval, stephane.ducasse, damien.pollet}@inria.fr

Abstract

A trait is a programming construct which provides code reusability. Traits are groups of methods that can be reused orthogonally from inheritance. Traits offer a solution to the problems of multiple inheritance by providing a behavior-centric modularity. Since traits offer an alternative to traditional inheritance-based code reuse, a couple of questions arise. For example, what is a good granularity for a Trait enabling reuse as well as plug ease? How much reuse can we expect on large existing inheritance-based hierarchies?

In this paper we take as case study the Smalltalk Collection hierarchy and we start rewriting it from scratch using traits from the beginning. We show how such library can be built using traits and we report such a preliminary experience. Since the Collection library is large, we focused and built the main classes of the library with Traits and report problems we encountered and how we solved them. Results of this experience are positive and show that we can build new collections based on the traits used to define the new library kernel.

1. Introduction

A trait is a programming construct which provides code reusability. Traits are groups of methods that can be reused orthogonally from inheritance. Traits offer a solution to the problems of multiple inheritance by providing a behavior-centric modularity. [SDNB03, DNS+06].

There are different trait model variations. In the original model, *Stateless traits* [SDNB03, DNS+06], traits only define methods, and no instance variables. *Stateful traits* [BDNW07] extend this model and let traits define state. *Freezable traits* [DWBN07] extend stateless traits with a visibility mechanism. In the context of this paper, when we use the term *trait* we mean *Stateless trait*. The reader unfamiliar with traits may read the appendix Section A for a rapid introduction to stateless traits.

Black et al. refactored the Squeak Smalltalk collection [BDN+07] hierarchy and showed a gain of 12% of code reuse [BSD03]. Still, their solution closely followed the inheritance-based collection hierarchy. Cassou et al. rewrote the Smalltalk stream hierarchy from scratch [CDW09]. They showed that traits support the reuse of code between a new kernel and a backward compatible one based on the same traits. Ducasse et al. reused and composed unit tests out of traits [DPBC09].

Problem: The goal of this paper is to experimentally verify the original claims of code reuse with traits, in the context of a forward engineering scenario. More specifically, our experiment looks for answers to questions that arise when using traits in practice: What is a good trait granularity which favors reuse as well as ease of reuse? Is the composition mechanism good enough to deal with common composition scenarios? What do we gain from using traits? When is it better to define a trait versus a class? Do we need state in traits?

Our approach was to redesign, from scratch, a new collection library based on traits. We identified traits for collections based on the work of Cook, who specified collection behavior [Coo92], and by analyzing the ANSI Smalltalk standard [ANS98]. Since elementary aspects of collections behavior are represented as traits, building new collections based on the composition of such traits is possible. We report on the creation of such new collections.

The paper contributions are:

- The identification of problems in the existing Collections library.
- The design of BLOC, a new library of collections composed from traits.
- Assessing whether traits act as reusable elements to define a library, and checking that the obtained design is clearer and more modular.
- Identifying trait related reuse.

In Section 2, we present the working hypotheses that drive this work. Then in Section 3, we highlight the existing Pharo Collection hierarchy and its modularity problems. Section 4 presents the hierarchy of traits based on the Collection behavior. Section 5 gives usage examples of the Collection Traits library to show the reusability of traits. In Section 6, we discuss the validity of traits, and finally we discuss related work in Section 7 before concluding in Section 8.

2. Working Questions

In this paper, we try to answer the following questions:

1. Trait granularity. Understanding whether a trait has a good size is a difficult topic. On one hand, we would like to reuse a coherent and a potentially large group of behaviors, but on the other hand we may want to only use part of the behavior to plug it into another scenario. Since there is no definitive answer and the answers will depend on the context and domain, we cannot draw immediate solutions. We would like, however, to empirically get an understanding of the granularity of traits that maximize reusability.

IWST'10 September 12, 2010, Barcelona, Spain.

2. Trait reusability. The ideas beside traits are modularity and reusability. A non-reusable trait is useless. The question is how much code can be reused in the Collections library.
3. Trait modularity. Can we define traits as effective building blocks? The idea is to have a library of traits to easily compose new classes from different traits and obtain specific behaviors.
4. Can we identify guidelines to assess when trait composition should be preferred over inheritance? This is an important question for class modularity. Inheritance has a strong impact in a system structure, whereas traits seem to be more difficult to understand without documentation.
5. Do traits need state? In the original model, traits do not have state, but in the context of collections, we want to understand whether the initialization of state in the class is a problem. A related question is to which extent mixing-like solutions that include state are better from a user point of view [BC90].
6. What are the trait limits do we encounter? Traits are a new approach, we enumerate limits and problems we encountered during the study.

3. Collections in Pharo

In Smalltalk, the Collections library is a central part of the system; it is used in the whole system, from the core to the UI. We have chosen the Collections library from Pharo because it is a complete library with a lot of different behavior [BDN+09, Gol84].The hierarchy is composed by more than seven levels, which exhibits reuse of behavior within branches of the hierarchy, but also across different branches. The Collections library in Smalltalk defines a rich set of behavior that we need: hash with HashedCollection, unicity with Set, sequence with OrderedCollection, order, identity with Identity related classes ... One of the problems is that elementary behaviors are often defined in a branch of the hierarchical inheritance structure and this forces their duplication across branches. For example, Dictionary inherits from HashedCollection, but dictionaries have both hashed and indexed behaviors, therefore there are some duplicated methods, like at:. The case of Dictionary shows that Traits could be a good design for Collections.

3.1 The Collections Library

The collection classes form a loosely-defined group of general-purpose subclasses of Collection and Stream. The group of classes that appears in the [GR83] contains 17 subclasses of Collection (Figure 1), and had already been redesigned several times before the Smalltalk-80 system was released. This group of classes is often considered to be a paradigmatic example of object-oriented design. In Pharo, the abstract class Collection has 101 subclasses, but many of these (like Bitmap, FileStream and CompiledMethod) are special-purpose classes crafted for use in other parts of the system or in applications, and hence not categorized as *Collections* by the system organization. In this paper, we use the term *Collections Hierarchy* to mean Collection and its 47 subclasses that are also in the categories labelled Collections-*.

3.2 Cook Analysis

We based our decomposition of the Collections library into traits on the one of Cook [Coo92]. Cook decomposes the Collections library in several behaviors, such as UpdatableCollection, IndexedCollection, ExtensibleCollection ... (shown in Figure 2). To decompose the Collections hierarchy, he uses the different messages and protocols that the classes define. What interested us in his work is the different behaviors that he defined.

On Figure 2 you can see the different behaviors Cook defines for collections, and the methods he selects to define each behavior.

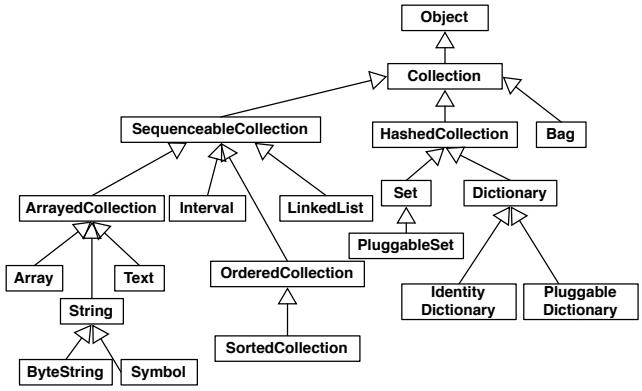

Figure 1. Current Collection hierarchy in Smalltalk

This work gives a first approach to a possible Collections library decomposition.

3.3 Single inheritance hampers reusability

The reusability in the Collection Library is limited to single inheritance. As with the example of Dictionary explained before, multiple inheritance is not available and the sole possibility without Traits is to copy and paste behaviors which are not in the same hierarchical branch.

The Collections library does not provide reusability. This library was built to be used, not extended by recomposition of elementary behaviors. If we want to create a new collection, the choice is not easy: what is the sole parent class of the new collection? Do we choose a generic class or a more specific one? The complex hierarchy of the Collections library does not help. Moreover, sometimes we need behaviors from different branches of the inheritance tree.

A simple example: OrderedSet. The new library built out of traits should support the definition of new collections easily. For example we want to be able to create an OrderedSet, a collection of unique ordered elements, which mixes the properties of a Set and OrderedCollection.

To create such a new collection, with the existing library, we have two choices either we inherit from Set and duplicate code from OrderedCollection, or we inherit from OrderedCollection and duplicate code from Set. In either case we must duplicate code because multiple inheritance does not exist in Smalltalk. This example reveals several problems:

- Of single inheritance limitations,
- necessity of code duplication, and
- lack of reuse.

Our work helps to avoid this problem. We create a library of traits which can be used and reused, either to recreate the existing collections, or to create new ones, as shown by the case studies we performed with the classes Dictionary and OrderedSet.

4. Overall design of BLOC

4.1 Traits

Traits are sets of methods designed to be reused as a group in classes. To define additional behaviors in a class, the class can compose a set of traits. A trait requires methods that are necessary to use the trait. Traits do not define state, instead, they require accessors. A complete explanation of Traits is available in Appendix A.

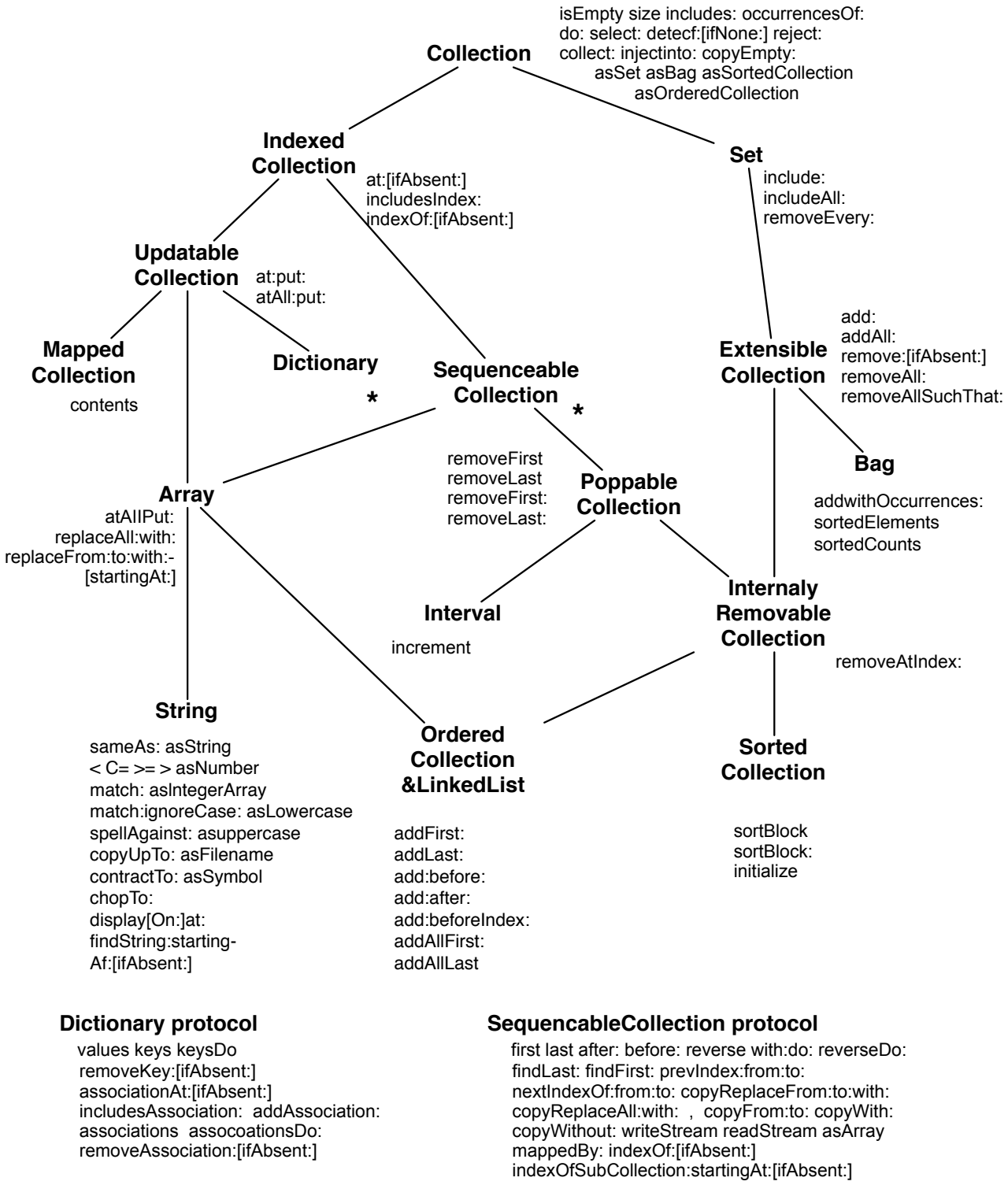

Figure 2. Hierarchy of Collection by Cook

To define a trait, we just send the message named:uses: to the class Trait, specifying the new trait name as well as the traits it is build upon. In the following code, the Trait TOrderedAccessing is defined with the use of behavior from the Trait TSequenceableAccessing.

```
Trait named: #TOrderedAccessing
    uses: TSequenceableAccessing
```

To define a class using traits, the class should inherit from its superclass. It should list the traits is used and provide the methods that should be defined. Here the class OBOrderedCollection inherits from Collection and uses some predefined traits such as TOrderedAccessing, TSort or TOrderedCopying. Then specific methods should be defined.

```
Collection subclass: #OBOrderedCollection
      uses: TOrderedAdding + TOrderedAccessing + TSort
+ TOrderedIterate + TOrderedCreation + TOrderedCollection
+ TOrderedRemoving + TOrderedCopying
+ TSequenceableTesting + TOrderedUpdatable
      instanceVariableNames: 'array firstIndex lastIndex'
      classVariableNames: ''
      poolDictionaries: ''
      category: 'BLOC-OrderedCollection'
```

4.2 BLOC Elementary Traits

The design of our Collection library is completely different from the inheritance-based one. There is only one level of inheritance: each collection class is a subclass of the abstract class Collection. It brings a semantical means and a collection gets the generic behavior of Collection (like atRandom:, anyOne or ,). Then a new collection becomes specific by adding Traits from the library of Traits.

In Figure 3, the collection BCOrderedCollection is composed of the traits: TOrderedAdding, TOrderedAccessing, TOrderedEnumerating, TOrderedUpdatable, TOrderedCreation, TOrderedCollection, TOrderedRemoving, TOrderedCopying, TSequenceableTesting.

To specify the main collections: OrderedCollection, Set, SortedCollection, Dictionary, Interval and Array, we created traits representing the behaviors defined by protocols proposed in the "Pharo by Example" book [BDN+09]. We created 9 different categories of traits presented in Table 1. Each of these categories can be defined (not necessarily) for each main collection.

If we take the case of Dictionary, Dictionary is a subclass of HashedCollection and needs behaviors from SequenceableCollection to be indexable. If we put all the behavior of one class in one trait, we have to cancelled some methods not used in Dictionary.

4.3 Methods: primary vs. secondary

In Smalltalk, traits do not have state. Our design supports this separation between traits and object state access. Indeed, trait methods should still access object state. In fact, we isolate state access by defining methods in the class. Then, traits use these methods. This concept follows encapsulation. It allows us to make traits independent from the state and the structure of a collection.

To make it, we define two types of methods (see Figure 4): primary and secondary methods. Primary methods access directly object state. They are accessors, but also more complex methods with processing to avoid the time consumption of accessors. Secondary methods use only other methods without accessing directly state. With this differentiation, the Traits library can be used in new collections with different structure. The primary methods are *required methods* of Traits, so when we create new collection we have to define the structure and these methods.

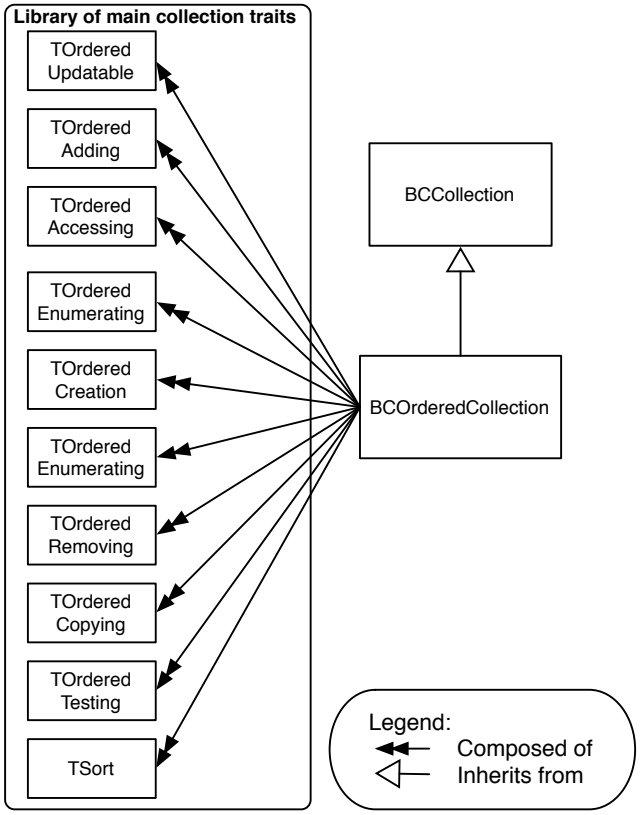

Figure 3. Overall structure

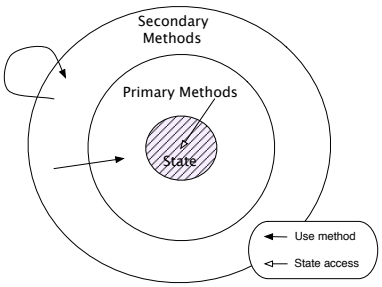

Figure 4. Encapsulation used to compose a collection

Therefore having a total abstraction of the state enhances reuse. Note that with this approach Traits do not require accessors directly since trait methods do not access state directly.

A primary method is not necessarily a required method. Primary methods are useful for the collection, it provides primary processing. Let's look at an example: In OrderedCollection, the method insert:before: accesses the state of the collection but this method is not used directly by the trait methods. It is used by other primary methods such as add:beforeIndex: and add:afterIndex:. So, insert:before: is a primary method which is not declared as required by the trait.

4.4 Composition Map

We defined the different behaviors of main collections and implemented them as traits. Table 2 shows all main collection in Traits, for each of them we defined the traits defined in Table 1. This way we could recreate the collections but based on elementary charac-

TXXXAccessing	Contains methods to accessing to the element(s) of the collection.	at:, ...
TXXXAdding	Contains methods to add element(s) in the collection.	add:, addAll:, ...
TXXXUpdating	Contains methods to change one or several elements in the collection.	at:put:, ...
TXXXRemoving	Contains methods to remove element(s) in the collection.	remove:, remove:ifAbsent:, ...
TXXXCopying	Contains methods to copy the collection.	copy, copyWith:, ...
TXXXTesting	Contains methods which test the collection or the elements in.	includes: isEmpty, ...
TXXXCreation	Contains methods to create the collection (class methods)	with:, new:, ...
TXXXEnumerating	Contains methods to iterate on the collection.	do:, select:, ...
TXXXCollection	Contains methods which are specific of the behavior of the collection.	hash, findElementOrNil, ...

Table 1. Elementary traits for composing collection behavior

TIndexed	Most indexed collections can retrieve elements with at:.
TSequenceable	Instances of all subclasses of SequenceableCollection start from a first element and proceed in a well-defined order to a last element.
THashed	Collection hashed used an hash function to store and access elements. This trait uses methods scanFor: and findElementOrNil:.
TOrdered	The *Ordered* behavior represents collections which are indexed and sequenceable.
TSet	There are methods for a set of unique elements and without nil.
TDictionary	It represents the behavior of a dictionary *i.e.,* it is an indexed collection which uses key as index. Keys permit to have attached elements. The couple key→element is stored in the collection.
TArrayed	It is a collection with a fixed size. It has the same behavior than OrderedCollection without growing behavior. This behavior is represented by a lack of the Trait: TArrayedAdding does not exist.

Table 2. Principal behavior-specific traits for collections in Pharo Smalltalk

teristics which can be recomposed and reused to create new collection.

We made a map of traits composition. In Figure 5, there is the map of the category Accessing. It has some similarity with the current Collection hierarchy, with a principle difference: there are multiple use of Traits which represent multi-inheritance.

5. Case studies

In this Section we present how the new kernel (*i.e.,* the trait library for the core classes) let us define new collection by recomposing and extending traits.

5.1 OrderedSet

We would like to define a new collection named OrderedSet that on the one hand offers a hash-based access and on the other hand an ordered access to its elements. Note that this collection is different from an UniqueOrdered collection that makes sure that its elements are ordered and not duplicated. For this goal, we create the new collection OrderedSet with the library of traits designed previously. As explained in Section 4.3, we simply create the structure to the new collection, create primary methods and use necessary traits. For this case study, we use two traits: TOrdered and TSet.

The following code shows how OrderedSet is defined. It uses all TOrdered traits and all TSet traits expect a few methods which are defined in both TSet and TOrdered. For example addAll: is defined in TOrdered, so the one in TSet is not needed.

```
Collection subclass: #OrderedSet
        uses: TSetAdding - {#addAll:} + TSetArithmetic + TSetTesting - {#=. #isSequenceable} + TSetIterate - {#doWithIndex:. #select:thenCollect:} + TSetRemoving - {#removeAll} + TSetAccessing - {#atRandom:} + TSetCopying - {#copyEmpty. #copyWith:. #copyWithout:} + TUnique + TSetCollection + TSetCreation + TOrderedAdding + TOrderedAccessing + TSort + TOrderedIterate + TOrderedCreation + TOrderedCollection + TOrderedRemoving + TOrderedCopying + TOrderedError + TSequenceableTesting + TOrderedUpdatable
    instanceVariableNames: 'array arrayO tally firstIndex lastIndex'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'BLOC'
```

OrderedSet reuses 70 methods (Figure 6). Then, we only have to reimplement or change the 38 required methods because the structure is particular: it contains two arrays to encode the two specific behaviors: hash access and order.

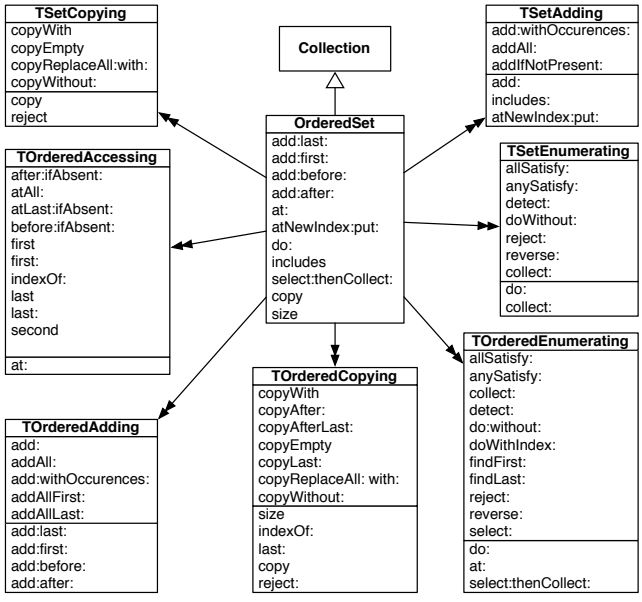

Figure 6. Map of traits used by OrderedSet

This example confirms the reusability of the BLOC library. Indeed, we create the new collection in less than two hours. In addition we do not have duplication code.

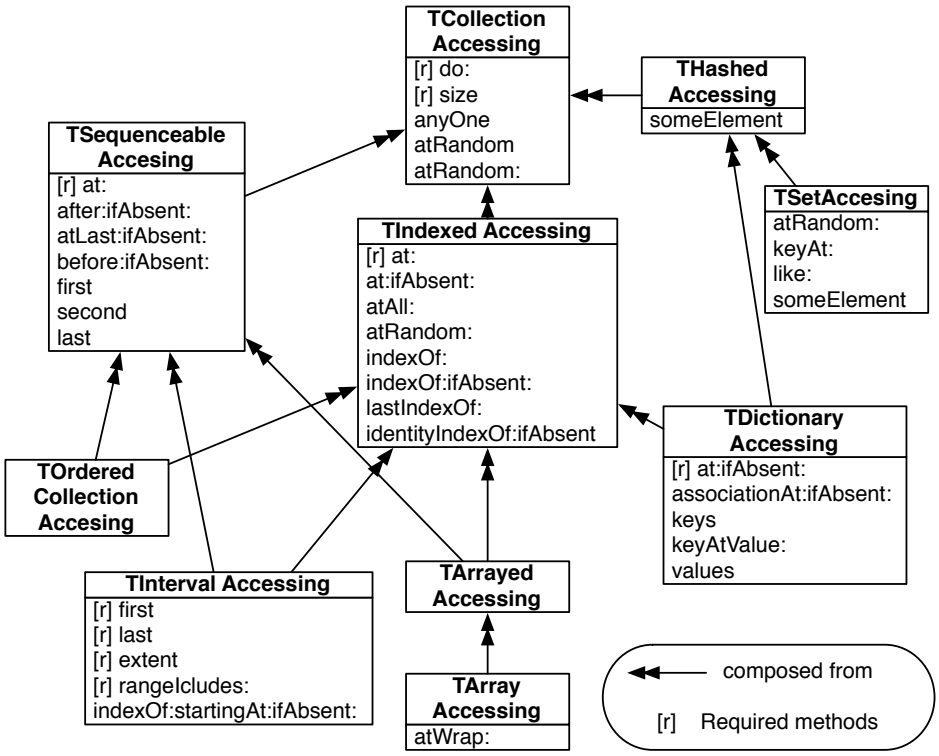

Figure 5. New trait hierarchy [r] represents required methods which are provided by primary methods or other Traits (secondary methods)

5.2 Dictionary

When creating the design of BLOC, we redefined certain existing collections such as Set, OrderedCollection, Interval .... This way we avoided code duplication. Dictionary is an example of good refactoring. In the existing library, Dictionary inherits from HashedCollection to have hash function. In addition, Dictionary is an indexed collection. This behavior is duplicated from SequenceableCollection and its subclasses.

In the original Dictionary class, there is some duplicated code, such as do: and associationsDo: which provides the same algorithm. With BLOC, Dictionary class uses 2 groups of traits TIndexed and THashed and defines some methods specific to the Dictionary.

```
Trait named: #TDictionaryAccessing
    uses: THashedAccessing + TIndexAccessing
    category: 'BLOC-Dictionary'
```

We redefined using traits the following collections: OrderedCollection, Interval, SortedCollection and Set. Now this redesign does not systematically improved existing code since some classes like Set did not present duplicated code. However, redesigning them and using traits (1) brings uniformity to the library, (2) core classes are the first clients of the traits they use, (3) it avoids duplication between such traits and their future clients. Finally an important point of the new design is that the use of traits did not hamper efficiency of the collection.

During the creation of BLOC, we discovered a difficulty: how to transform existing methods invoking super into traits. Indeed, invoking super in a trait a sign of not totally rethought functionality since it means that the trait is designed to be plugged in a hierarchy where the superclass is somehow fixed by the API it should offer to the trait. This is against the trait philosophy to be orthogonal to inheritance. Since the existing hierarchy is heavily based on inheritance, we had to face such situations. For example, OrderedCollection uses the method asSortedArray. This method uses super to call asSortedArray of SequenceableCollection but in our new hierarchy we only inherits from Collection. Therefore we have to redefine all the methods which use super to call a method that by construction may not be in the superclass. This example shows that traits are useful to avoid problems from the single inheritance. Indeed, traits permit to simulate a multiple inheritance without state.

5.3 Reusability Comparison

Table 3 presents how much the core traits are reused. For each trait it presents the number of client classes, the number of required methods and the number of methods that the trait provides. We see a good ratio provided/required for most traits, except for Interval. There are multiple reasons for this difference: Interval is more specialized than the other collections. As a consequence a large part of its API is tailored towards specific behavior and methods access directly the interval underlying structure. This explains the larger number of required methods. Note that this is a consequence of our design decision to avoid accessors and their associated cost. In presence of a JIT such point could be changed.

Table 5 presents some metrics which compare the same functionalities in the Pharo implementation and in BLOC. Note that the table presents the sum of traits for a given category: for example, TOrdered is the sum of all the traits related to the Ordered behavior. Which one indicates that BLOC has much more classes and Traits than Pharo collections and (number of methods) show that the amount of code is smaller in BLOC than in original library. BLOC has 10.9% less methods than the corresponding Pharo collection library. This means we avoided reimplementing a lot of methods by putting them in Traits. Finally, we can deduce from which

Trait	client classes	required methods	provided methods	ratio prov. / req.
TCollection	3	10	92	9,20
TSequenceable	3	5	55	11
TIndexed	3	2	51	25,50
THashed	2	7	11	1,57
TOrdered	1	6	28	4,66
TSet	1	3	21	7
TDictionary	1	9	45	5
TArrayed	1	0	21	
TInterval	1	16	6	0,375

Table 3. BLOC-trait reusability.

Trait	required methods	provided methods	ratio prov. / req.
TSequenceableAccessing	1	17	17.0
TSequenceableCollection	1	3	3.0
TSequenceableConcatenation	0	1	
TSequenceableCopying	1	16	16.0
TSequenceableCreation	0	2	
TSequenceableIterate	2	9	4.5
TSequenceableRemoving	3	0	0.0
TSequenceableTesting	3	1	0.33
TSequenceableUpdatable	5	6	1.2

Table 4. Sequenceable-trait reusability.

that the design of BLOC is better: there are fewer cancelled methods and there are half as many methods less in BLOC than in Pharo.

	Pharo	BLOC	$\frac{Pharo-\text{BLOC}}{Pharo}$
# Classes and Traits	8	84	-950%
# Methods	510	454	10,9%
# Cancelled Methods	6	2	66%
# Reimplemented Methods	79	36	54%

Table 5. Some metrics comparing BLOC and pharo collection kernel.

6. Discussions

Granularity of traits. There is no definitive answer to the good granularity of traits but what we learn is that to enable reuse fine-grained traits are mandatory. Indeed, if we want to avoid duplicating code, the traits have to be small. Now pushing the idea the extreme, we could have one trait for one method. In such a case each method will be defined once. Now this is clearly not a good idea since we want also traits to represent a abstraction or a partial behavior. Adequate granularity is defined by the context. We have found a good granularity for trait in our context. Table 4 displays all traits related to SequenceableCollection. Some traits have no provided methods, because these methods are provided in TCollection. Note that the number of provided methods is variable and depends on the behavior provided.

Trait reusability. The reuse of trait depends on the behaviors. Indeed, for the collection we have a good reusability. We can now easily create different collections with BLOC. But all source code could not be reused because some methods depend on the underlying structure. In our solution, we removed a lot of reusability constraints except for methods which access state.

Trait composition vs Inheritance. One of the questions when building a system with traits is to decide when to use inheritance and when to use traits. In the Collection hierarchy (see Section 3.1), defining a class or inheriting from a class does not make sense since some of its state cannot be used or its behavior should be canceled. This is a clear motivation for using traits. Most of the time, however, the decision is not that easy to take, and the designer has to assess whether potential clients may benefit from the traits, i.e., if the defined behavior can be reused in another hierarchy.

Traits with state. In our work, we looked at the importance to have state in traits. In the context of collections we think that it is not necessary. If state is included in a trait, it also includes constraints for the implementation of future classes. In our context, to have state in traits is not necessary because of the definition of primary methods. The initialization of the state and its recomposition when used by different clients is also a problem that we did not assess but that should not be neglected.

Trait problems and limits. During our experience, we detected some limits and problems related to traits. The first problem we had was the lack of browser or tools for traits. Indeed, it is difficult to see traits, which classes use them, documentation, required methods, ... Traits are arbitrarily in the use: clause of the class definition. Therefore it becomes difficult to read what traits are used by the class.

7. Related work

Traits. We already compared our approach with the few work refactoring existing code using traits. Now we want to summarize the key differences.

Cassou et. al [CDW09] rewrote the Stream Smalltalk hierarchy from scratch. What is interesting is that they obtained a kernel based on traits that can be assembled to reproduce the old kernel as well as express a completely new design.

Ducasse et. al reuse and compose unit tests out of traits for the collection hierarchy [DPBC09]. This work is closer to our approach since they focused on identifying elementary collection behavior. Then they used these elementary behavior to assemble tests for traits.

[BSD03] proposed a refactoring of the existing collection but they were bound to the existing hierarchy. The work presented in this paper was focusing more on rethinking the collection as assembly of composable behaviors.

[LDA05] and [BBN08] proposed to use FCA to help automatically refactoring and identifying traits in Smalltalk and Java programs. The results are not as good as a manual approach because design is complex and FCA is just an indication that some methods could be optimally reused.

Automatic code reorganization of non traits code . We now present the approaches that automatically transform existing libraries using Formal Concept Analysis (FCA) or other techniques. FCA was used in different ways.

Godin [GMM+98] developed incremental FCA algorithms to infer implementation and interface hierarchies guaranteed to have no redundancy. To assess their solutions they used structural metrics. They analyzed the Smalltalk Collection hierarchy. One important limitation is that they consider each method declaration as a different method and thus cannot identify code duplication. Since the resulting hierarchies cannot be implemented in Smalltalk because of single inheritance, it would be interesting to understand whether their results could be indication for traits.

Snelting and Tip analyzed a class hierarchy by making the relationship between class members and variables explicit [ST98]. By analyzing the hierarchy client *usage*, they detected design anoma-

lies such as class members that are redundant or that can be moved into a derived class. From this client perspective, Streckenbach infer improved hierarchies in Java [SS04]. They proposed solution that should be further be manually adapted. The tool proposes the reengineer to move methods up in the hierarchy to work around multiple inheritance situations generated by the generated lattice. The resulting refactoring is behavior preserving only with respect to the analyzed client programs.

Moore [Moo96] proposes automatic refactoring of Self inheritance hierarchies. Moore factors out common expressions in methods. Resulting hierarchies do not contain any duplicated expressions or methods. Moore's factoring creates methods with meaningless names which is a problem if the code should be read. The approach is more optimizing method reuse than creating coherent composable groups of methods.

Casais uses an automatic structuring algorithm to reorganize Eiffel class hierarchies using decomposition and factorization [Cas94]. In his approach, he increases the number of classes in the new refactored class hierarchy. Dicky *et al.* propose a new algorithm to insert classes into a hierarchy that takes into account overridden and overloaded methods [DDHL96].

The key difference from our results is that all the work on hierarchy reorganization focuses on transforming hierarchies using inheritance as the only tool. In contrast, we are interested in exploring other mechanisms, such as explicit composition mechanisms like traits composition in the context of mixin-like languages. Another important difference is that we don't rely on algorithms, to obtain the design.

8. Conclusion

In this paper we assessed the traits in a reuse context. We refactored the collection library to create a library of traits which can be composed into the behavior of main collections. This work represents a preliminary experience. A lot of questions has been raised, with no answer in this work. The need of stateful traits or the granularity is defined depending on the case. However, this study confirmed some goals of traits. The results of modularity and reusability offered by traits are good on the collection library. As future work, we need to better investigation of how to use traits, how to better define granularity. It is also important to define a browser to navigate between Traits, classes, behavior, documentation.

References

[ANS98] ANSI, New York. *American National Standard for Information Systems – Programming Languages – Smalltalk, ANSI/INCITS 319-1998*, 1998. http://wiki.squeak.org/squeak/uploads/172/standard_v1_9-indexed.pdf.

[BBN08] Lorenze Bettini, Viviana Bono, and Marco Naddeo. A trait based re-engineering technique for java hierarchies. In *PPPJ 2008*, ACM International Conference Proceedings. ACM Press, 2008.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, October 1990.

[BDN+07] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Squeak by Example*. Square Bracket Associates, 2007. http://SqueakByExample.org/.

[BDN+09] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[BDNW07] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits. In *Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006)*, volume 4406 of *LNCS*, pages 66–90. Springer, August 2007.

[BSD03] Andrew P. Black, Nathanael Schärli, and Stéphane Ducasse. Applying traits to the Smalltalk collection hierarchy. In *Proceedings of 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, volume 38, pages 47–64, October 2003.

[Cas94] Eduardo Casais. Automatic reorganization of object-oriented hierarchies: A case study. *Object-Oriented Systems*, 1(2):95–115, December 1994.

[CDW09] Damien Cassou, Stéphane Ducasse, and Roel Wuyts. Traits at work: the design of a new trait-based stream library. *Journal of Computer Languages, Systems and Structures*, 35(1):2–20, 2009.

[Coo92] William R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *Proceedings of OOPSLA '92 (7th Conference on Object-Oriented Programming Systems, Languages and Applications)*, volume 27, pages 1–15. ACM Press, October 1992.

[DDHL96] Hervé Dicky, Christoph Dony, Marianne Huchard, and Thérèse Libourel. On Automatic Class Insertion with Overloading. In *Proceedings of OOPSLA '96 (11th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications)*, pages 251–267. ACM Press, 1996.

[DNS+06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, March 2006.

[DPBC09] Stéphane Ducasse, Damien Pollet, Alexandre Bergel, and Damien Cassou. Reusing and composing tests with traits. In *Proceedings of the 47th International Conference Objects, Models, Components, Patterns (TOOLS-Europe'09)*, pages 252–271, jun 2009.

[DWBN07] Stéphane Ducasse, Roel Wuyts, Alexandre Bergel, and Oscar Nierstrasz. User-changeable visibility: Resolving unanticipated name clashes in traits. In *Proceedings of 22nd International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 171–190, New York, NY, USA, October 2007. ACM Press.

[GMM+98] Robert Godin, Hafedh Mili, Guy W. Mineau, Rokia Missaoui, Amina Arfi, and Thuy-Tien Chau. Design of Class Hierarchies based on Concept (Galois) Lattices. *Theory and Application of Object Systems*, 4(2):117–134, 1998.

[Gol84] Adele Goldberg. *Smalltalk 80: the Interactive Programming Environment*. Addison Wesley, Reading, Mass., 1984.

[GR83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[LDA05] Adrian Lienhard, Stéphane Ducasse, and Gabriela Arévalo. Identifying traits with formal concept analysis. In *Proceedings of 20th Conference on Automated Software Engineering (ASE'05)*, pages 66–75. IEEE Computer Society, November 2005.

[Moo96] Ivan Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *Proceedings of OOPSLA '96 (11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications)*, pages 235–250. ACM Press, 1996.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[SS04] Mirko Streckenbach and Gregor Snelting. Refactoring class hierarchies with KABA. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 315–330, New York, NY, USA, 2004. ACM Press.

[ST98] Gregor Snelting and Frank Tip. Reengineering Class Hierarchies using Concept Analysis. In *ACM Trans. Programming Languages and Systems*, 1998.

A. Appendix: Traits in a Nutshell

To ease the understanding of this paper we added this section which presents traits in a nutshell. This part is taken from [DWBN07] and is not part of the current article. It is just added here for sake of completeness and understanding the ideas presented in the paper.

Reusable groups of methods. Traits are units of behaviour. They are sets of methods that serve as the behavioural building block of classes and primitive units of code reuse [DNS$^+$06]. In addition to offering behaviour, traits also *require methods*, *i.e.,* methods that are needed so that trait behaviour is fulfilled. Traits do not define state, instead they require accessor methods.

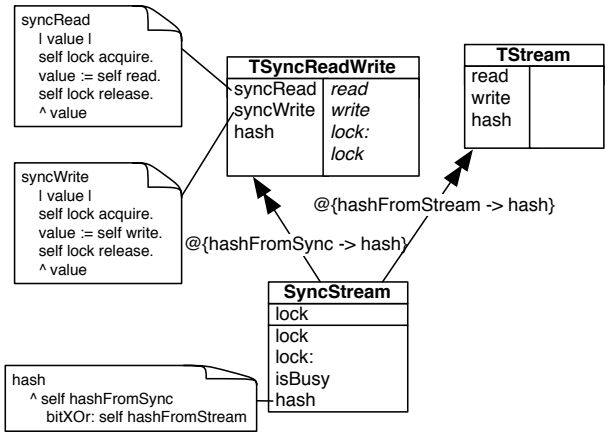

Figure 7. The class SyncStream is composed of the two traits TSyncReadWrite and TStream.

Figure 7 shows a class SyncStream that uses two traits, TSyncReadWrite and TStream. The trait TSyncReadWrite provides the methods syncRead, syncWrite and hash. It requires the methods read and write, and the two accessor methods lock and lock:. We use an extension to UML to represent traits (the right column lists required methods while the left one lists the provided methods).

Explicit composition. A class contains a super-class reference, uses a set of traits, defines state (variables) and behaviour (methods) that glue the traits together; a class implements the required trait methods and resolves any method conflicts.

Trait composition respects the following three rules:

- Methods defined in the composer take precedence over trait methods. This allows the methods defined in a composer to override methods with the same name provided by the used traits; we call these methods *glue methods*.
- Flattening property. In any class composer the traits can be in principle in-lined to give an equivalent class definition that does not use traits.
- Composition order is irrelevant. All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

Conflict resolution. While composing traits, method conflicts may arise. A *conflict* arises if we combine two or more traits that provide identically named methods that do not originate from the same trait. There are two strategies to resolve a conflict: by implementing a (glue) method at the level of the class that *overrides* the conflicting methods, or by *excluding* a method from all but one trait. Traits allow method *aliasing*; this makes it possible to introduce an additional name for a method provided by a trait. The new name is used to obtain access to a method that would otherwise be unreachable because it has been overridden [DNS$^+$06].

In Figure 7, the class SyncStream is composed from TSyncReadWrite and TStream. The trait composition associated to SyncStream is:

TSyncReadWrite alias *hashFromSync* $\rightarrow$ *hash*
+ TStream alias *hashFromStream* $\rightarrow$ *hash*

The class SyncStream is composed of (i) the trait TSyncReadWrite for which the method hash is aliased to hashFromSync and (ii) the trait TStream for which the method hash is aliased to hashFromStream.

Method composition operators. The semantics of trait composition is based on four operators: sum ($+$), override ($\triangleright$), exclusion ($-$) and aliasing (alias $\rightarrow$) [DNS$^+$06].

The *sum* trait TSyncReadWrite + TStream contains all of the non-conflicting methods of TSyncReadWrite and TStream. If there is a method conflict, that is, if TSyncReadWrite and TStream both define a method with the same name, then in TSyncReadWrite + TStream that name is bound to a known method conflict. The + operator is associative and commutative.

The *override* operator ($\triangleright$) constructs a new composition trait by extending an existing trait composition with some explicit local definitions. For instance, SyncStream overrides the method hash obtained from its trait composition.

A trait can *exclude* methods from an existing trait using the exclusion operator $-$. Thus, for instance, TStream $-$ {read, write} has a single method hash. Exclusion is used to avoid conflicts, or if one needs to reuse a trait that is "too big" for one's application.

The *method aliasing* operator alias $\rightarrow$ creates a new trait by providing an additional name for an existing method. For example, if TStream is a trait that defines read, write and hash, then TStream alias *hashFromStream* $\rightarrow$ *hash* is a trait that defines read, write, hash and hashFromStream. The additional method hashFromStream has the same body as the method hash. Aliases are used to make conflicting methods available under another name, perhaps to meet the requirements of some other trait, or to avoid overriding. Note that since the body of the aliased method is not changed in any way, an alias to a recursive method is not recursive.

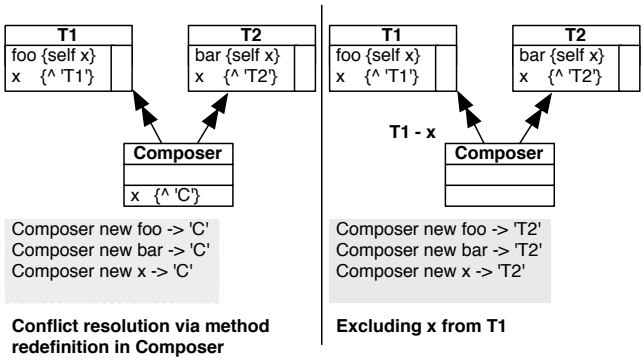

Figure 8. Trait conflict resolution strategies: either via method redefinition or via method exclusion.