Redesigning with Traits: the Nile Stream trait-based Library

Presented at International Conference on Dynamic Languages 2007

Damien Cassou[1], Stéphane Ducasse[1], and Roel Wuyts[2]

[1] LISTIC, University of Savoie, France
[2] IMEC, Leuven and Université Libre de Bruxelles

Abstract. Recently, traits have been proposed as a single inheritance backward compatible solution in which the composing entity has the control over the trait composition. Traits are fine-grained units used to compose classes, while avoiding many of the problems of multiple inheritance and mixin-based approaches.
To evaluate the expressiveness of traits, some hierarchies were *refactored*, showing code reuse. However, such large refactorings, while valuable, may not be facing all the problems, since the hierarchies were previously expressed within single inheritance and following certain patterns. We wanted to evaluate how traits enable reuse, and what problems could be encountered when building a library using traits *from scratch*, taking into account that traits are units of reuse. This paper presents our work on designing a new stream library named Nile. We present the reuse that we attained using traits, and the problems we encountered.
Keywords. Object-Oriented Programming, Inheritance, Refactoring, Traits, Code Reuse, Smalltalk

1 Introduction

Multiple inheritance has been the focus of a large amount of work and research efforts. Recently, traits proposed a solution in which the composite entity has the control and which can be flattened away, *i.e.,* traits do not affect the run-time semantics [1, 2]. Traits are fine-grained units that can be used to compose classes. Like any solution to multiple inheritance, the design of traits is the result of a set of trade-offs. Traits favor simplicity and fine-grained composition. Traits are meant for single inheritance languages. Trait composition conflicts are automatically detected but the composer has the control to resolve these conflicts explicitly. Traits claim to avoid many of the problems of multiple inheritance and mixin-based approaches that mainly favor linearization where conflicts never arise explicitly and are solved implicitly by ordering.

Note that there exist different trait models. In the original trait model, *Stateless traits* [1, 2], traits only define methods, but not instance variables. *Stateful traits* [3] extends this model and lets traits also define state. *Freezable traits* [4] extend stateless traits with a visibility mechanism. In the context of this paper when we use *trait* we mean *Stateless trait*.

Previous research evaluated the usefulness of traits by refactoring the Smalltalk collection and stream libraries, which showed up to 12% gain in terms of code reuse [5]. Other research tried to semi-automatically identify traits in existing libraries [6]. While these are valuable results, they are all refactoring scenarios that investigated the applicability of traits using *existing* systems as input. Usability and reuse of traits when developing a *new* system has not been assessed. Implementing a stream library from scratch is an important experience to test the expressiveness of traits. By doing so we may face problems that may have been hidden in previous experiences and also face a large scheme of trait composition problems.

The goal of this paper is to experimentally verify the original claims of simplicity and reuse of traits in the context of a forward engineering scenario. More specifically, our experiments want to get answers to the following questions that quickly arise when using traits in practice:

- Trait granularity. We want to assess the granularity of traits that maximize their reusability and composition.
- Trait reusability. We want to understand how much code can be reused.
- Can we define traits as composable building units?
- Can we identify guideline to assess when trait composition should be preferred over inheritance?
- To what extent can we fix the problems identified in the current stream hierarchy?
- What trait limits and problems do we encounter?
- Does the use of trait imply an execution cost?

Our approach is based on designing and implementing a non-trivial library from scratch using traits. We decided to build a stream collection library (called *Nile*) that follows the ANSI Smalltalk standard [7] yet remains compatible with the current Smalltalk implementations. The choice for a stream library was motivated by a number of reasons:

- streams exhibit problems linked to the fact that they are naturally modeled using multiple inheritance. In presence of single inheritance the implementors are reduced to duplicated code and other tricks such as canceling methods;
- N. Schärli [5] and A. Lienhard [6] already refactored the Stream library using traits so we can compare with their results;
- streams are an important abstraction of computer language libraries;
- several constraints are imposed by the ANSI Smalltalk standard and the need to remain usable in existing Smalltalk dialects.

Nile is structured around three core traits and a set of libraries. During the definition of the libraries, the core traits proved to have a good granularity: it was easy to obtain each desired functionality composition using the adequate part of the core. Nile has 18% less methods and 15% less bytecodes than the corresponding Squeak collection-based stream library. Moreover, Nile has neither canceled method nor method implemented too high in the hierarchy. There are only three overrides compared to the fourteen of Squeak.

The contributions of the paper are: (1) the design of *Nile*, a new stream library made of composable units, (2) the assessment that traits are good building units for defining libraries and that they enable clean design and reuse through composibility, and (3) the identification of problems when using the traits.

We start by presenting the existing Squeak Stream hierarchy limits and the ANSI Smalltalk standard protocols (Section 2). Section 3 presents an overview of Nile and the core of the library around its three most important traits. Section 4 and Section 5 detail the implementation of the collection-based and file-based stream libraries, respectively. Two other libraries will be presented in Section 6. Section 7 compares our approach with the one of N. Schärli [5]. It analyses the reuse offered by traits as well as performance issues and optimization solutions. Finally, Section 8 presents the problems we identify due to the use of traits.

2 Analyses

In this section, we analyze the existing stream hierarchy of Squeak the open-source Smalltalk [8]. We highlight the key problems and present the ANSI Smalltalk standard.

2.1 Analysis of the Squeak stream hierarchy

Squeak [8], like all Smalltalk environments, has its own implementation of a stream hierarchy. Figure 1 presents the core of this implementation, which is solely based on single inheritance and does not use traits. Note that most Smalltalk dialects reimplemented streams and therefore have similar yet different implementation. For example, even though Squeak and VisualWorks are both direct descendants from the original Smalltalk-80, their stream hierarchies are different since the one in VisualWorks was completely reimplemented.

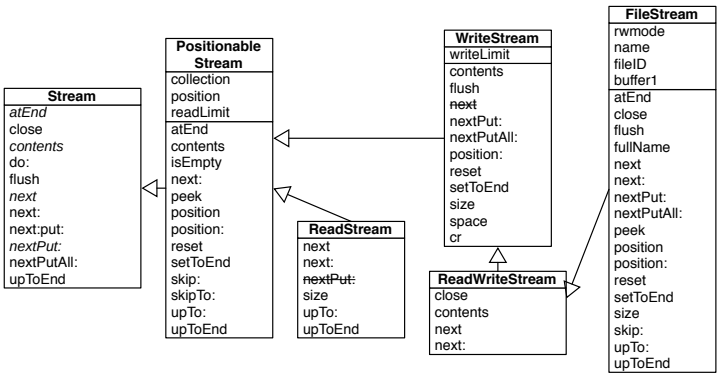

Fig. 1. The Squeak core Stream hierarchy. Only the most important methods are shown.

The existing single-inheritance implementation has different problems that we detail.

Methods implemented too high in the hierarchy. A common technique to avoid duplicating code consists in implementing a method in the topmost common superclass of all classes which need this method. Even if efficient, this technique pollutes the interface of classes which do not want this method. For example, Stream defines nextPutAll: which calls nextPut:

```
Stream>>nextPutAll: aCollection
  "Append the elements of aCollection to the sequence of objects
   accessible by the receiver. Answer aCollection."

  aCollection do: [:v | self nextPut: v].
  ^ aCollection.
```

The method nextPutAll: writes all elements of the parameter aCollection to the stream by iterating over the collection and calling nextPut: for each element. The method nextPut: is abstract and must be implemented in subclasses, and even if Stream defines methods to write to the stream, some subclasses are used for read-only purposes, like ReadStream. Those classes must then cancel explicitly the methods they don't want.[3] This approach, even if it was probably the best available solution at the time of the first implementation, has some drawbacks. First of all the class Stream and its subclasses are polluted with a number of methods that are not available in the end. This complicates the task of understanding the hierarchy and extending it. It also makes it more difficult to add new subclasses. To add a new subclass, a developer must analyze all of the methods implemented in the superclasses and cancel all unwanted ones.

Unused superclass state. The class FileStream is a subclass of ReadWriteStream and an indirect subclass of PositionableStream which is explicitly made to stream over collections (see Figure 1). Then, the instance variables collection, position and readLimit inherited from the PositionableStream and writeLimit inherited from WriteStream are not used for FileStream and all its subclasses.

Simulating multiple inheritance by copying. ReadWriteStream is conceptually both a ReadStream and a WriteStream. However, Smalltalk is a single inheritance-based language, so ReadWriteStream can only subclass one of these. The behaviour from the other one has to be copied, leading to code duplication and all of its related maintenance problems.

The designers of the Squeak stream hierarchy decided to subclass WriteStream to implement ReadWriteStream, and then copy the methods related to reading from ReadStream.

One of the copied methods is next, which reads and returns the next element in the stream. This leads to a strange situation where next is being canceled

[3] In Smalltalk, canceling a method is done by reimplementing the method in the subclass and calling shouldNotImplement from it.

out in WriteStream (because it should not be doing any reading), only to be reintroduced by ReadWriteStream. The reason for this particular situation is due to the combination of next defined too high in the hierarchy and single inheritance.

Reimplementation. In Figure 1, one can see that next: is implemented five times. Not a single implementation uses super which means that each class completely reimplements the method logic instead of specializing it. But this statement should be tempered because often in the Squeak stream hierarchy, methods override other methods to improve speed execution: this is because in subclasses, the methods have more knowledge and, thus, can do a faster job. However, a method reimplemented in nearly all of the classes in a hierarchy suggests inheritance hierarchy anomalies.

2.2 The ANSI Smalltalk standard

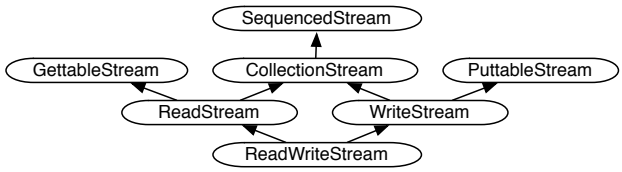

Fig. 2. The ANSI Smalltalk standard stream protocol hierarchy.

Figure 2 shows that even if Smalltalk is a single inheritance language, the ANSI Smalltalk standard [7] defines the different protocols using multiple inheritance. In the standard, streams are based on the notion of sequence values. Each stream has past and future sequence values. The ANSI Smalltalk standard defines a decomposition of stream behavior around three main protocols: GettableStream, SequencedStream and PuttableStream. Table 1 and Table 2 summarize the protocol contents.

The ANSI Smalltalk standard provides a useful starting point for an implementation even if a lot of useful methods are not described. We therefore chose to adopt it for Nile.

About GettableStream>>peekFor:. The standard proposes a definition of peekFor: that most Smalltalk implementations do not follow. The ANSI Smalltalk standard is equivalent to an equality test between the peeked object and the parameter:

```
GettableStream>>peekFor: anObject
  ^ self peek = anObject
```

SequencedStream	
close	Disassociate a stream from its backing store.
contents	Returns a collection containing the receiver's past and future sequence values in order.
isEmpty	Returns a boolean indicating whether there are any sequence values in the receiver.
position	Returns the number of sequence values in the receiver's past sequence values.
position:	Sets the number of sequence values in the receiver's past sequence values to be the parameter.
reset	Resets the position of the receiver to be at the beginning of the stream of values.
setToEnd	Set the position of the stream to its end.

PuttableStream	
flush	Upon return, if the receiver is a write-back stream, the state of the stream backing store must be consistent with the current state of the receiver.
nextPut:	Writes the argument to the stream.
nextPutAll:	Enumerate the argument, adding each element to the receiver.

Table 1. The SequencedStream and PuttableStream protocols defined by the ANSI Smalltalk standard.

GettableStream	
atEnd	Returns true if and only if the receiver has no future sequence values available for reading.
do:	Evaluates the argument with each receiver future sequence value.
next	The first object is removed from the receiver's future sequence values and appended to the end of the receiver's past sequence values. The object is returned.
next:	Does next a certain amount of time and returns a collection of the objects returned by next.
nextMatchFor:	Reads the next object from the stream and returns true if and only if the object is equivalent to the argument.
peek	Returns the next object in the receiver's future sequence values without advancing the receiver's position.
peekFor:	Peek at the next object in the stream and returns true if and only if it matches the argument.
skip:	Skip a given amount of object in he receiver's future sequence values.
skipTo:	Sets the stream just after the next occurrence of the argument and returns true if it's found before the end of the stream.
upTo:	Returns a collection of all the objects in the receiver up to, but not including the next occurrence of the argument.

Table 2. GettableStream protocol defined by the ANSI Smalltalk standard.

Most Smalltalk implementations (including Dolphin, GemStone, Squeak, VisualAge, VisualSmalltalk, VisualWorks, Smalltalk-X and GNU Smalltalk) do not only test the equality but also increment the position in case of equality as shown by the following implementation.

```
peekFor: anObject
  "Answer false and do not move over the next element if it is not equal
   to the  argument, anObject, or if the receiver is at  the end. Answer
   true and  increment the position, if the  next  element  is  equal to
   anObject."

  ^ (self atEnd not and: [self peek = anObject])
        ifTrue: [self next. true]
        ifFalse: [false]
```

This definition lets the following code parse '145', ' 145' and '-145' without problem:

```
stream := ReadStream on: '- 145'.
negative := stream peekFor: $-.
stream peekFor: Character space.
number := stream upToEnd.
```

Regarding the name of SequencedStream. The name SequencedStream is not well chosen, since this protocol provides absolute positioning in the stream. A name evoking this would have been better.

3 Nile overview and core

Nile is designed around a core of traits offering base functionality reflecting the ANSI Smalltalk standard. The core consists of only three traits and it is then used in several libraries that we discuss in detail throughout the paper. File-based streams and collection-based streams are among the most prominent libraries. Other libraries we discuss are support for writing character constants and decoders (streams that can be chained). Figure 3 presents an overview of Nile.

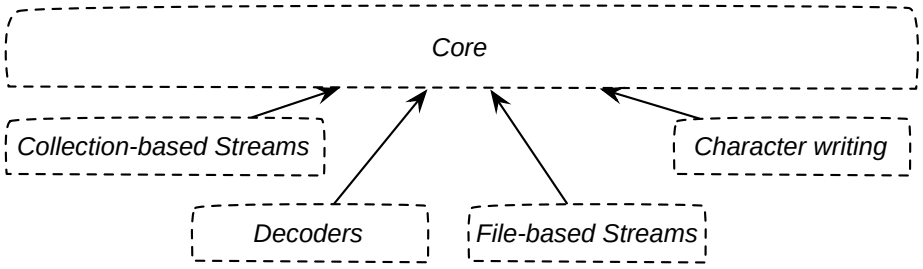

Fig. 3. Overview of Nile: the core and its different libraries.

We designed Nile around three independent traits, reflecting the ANSI Smalltalk standard: TPositionableStream, TGettableStream and TPuttableStream. They are shown in Figure 4.

TGettableStream. The trait TGettableStream is meant for all streams used to read elements of any kind. The trait requires 4 methods: atEnd, next, peek and outputCollectionClass. The method peek returns the following element without moving the stream whereas next reads and returns the following element and moves the stream. The method TGettableStream>>outputCollectionClass is used to determine the type of collection which is used when returning collection of elements as with next: and upTo:.

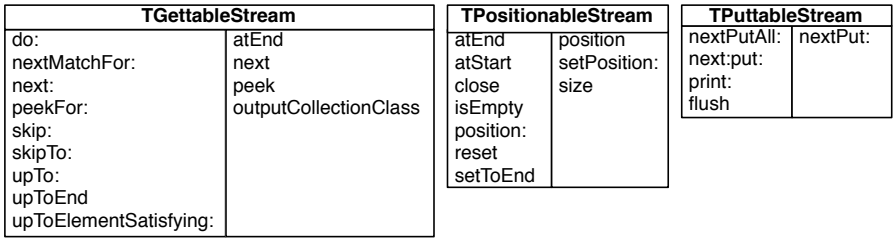

Fig. 4. The Nile core traits.

TPositionableStream. The trait TPositionableStream allows for the creation of streams that are positioned in absolute manner. It corresponds to the ANSI Smalltalk standardSequencedStream protocol; we thought the name TPositionableStream made more sense. The only required methods are size and two accessors for a position variable. We decided to implement the bound verification of the method position: in the trait itself: the parameter must be between zero and the stream size. This means that two methods have to be implemented: a pure accessor, named setPosition: here, and the real public accessor named position: which verifies its parameter value.

TPuttableStream. This trait is the simplest of the Nile library. It provides nextPutAll:, next:put:, print: and flush and requires nextPut:. By default, flush does nothing. It is used for ensuring that everything has been written. Buffer-based streams should have their own implementations.

4 Collection-based streams

To support streaming over collections we implemented a set of dedicated traits and what we call *trait factories* that define their creation protocols. Note that, in contrast to the default Squeak implementation and like in VisualWorks, our implementation actually works with any sequenceable collection, not just Arrays and Strings.

4.1 The traits

The traits TCollectionStream, TReadableCollectionStream and TWriteableCollectionStream implement the collection-based functionalities (as shown in Figure 5 — Note that in the figures traits have their name in bold whereas classes not). They provide all necessary methods required by the core traits, while only requiring 4 new accessors.

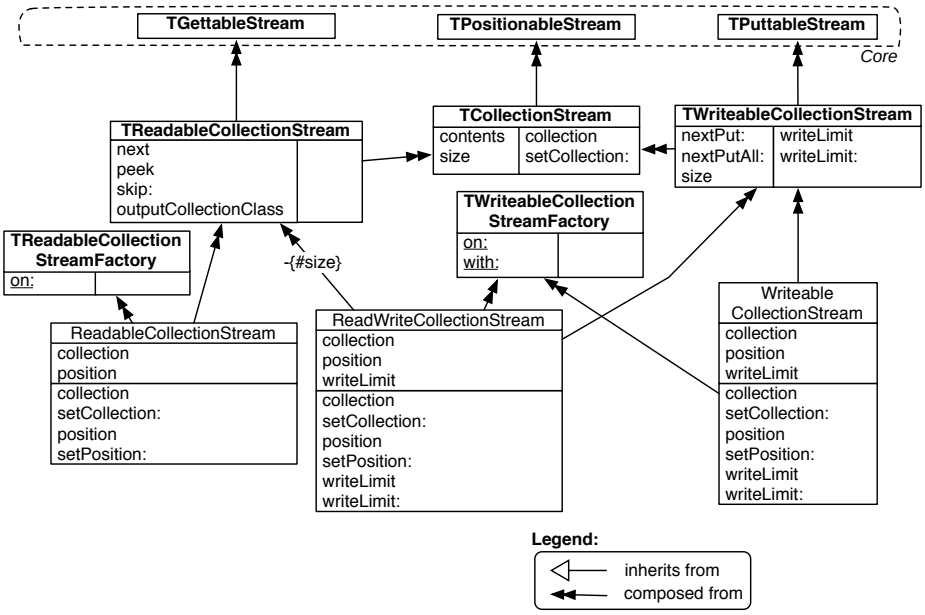

Fig. 5. The collection-based stream library. We use a UML-based notation to represent traits: methods on the right are *required* and methods on the left are *provided*.

TReadableCollectionStream. The trait TReadableCollectionStream helps creating classes which streams over readable collections. It implements the required methods of TGettableStream: next, outputCollectionClass, and peek. It also redefines skip: for efficiency reasons. The required method TGettableStream>>atEnd is provided by TPositionableStream and thus, does not require further work.

TCollectionStream. This trait is inspired by the ANSI Smalltalk standard. It is used for every stream that needs to read from or write to a collection. This trait defines contents and size in terms of two new methods: collection and setCollection:. The former must return the internal collection and the latter provides a setter for this collection. The method size returns the size of the collection.

TWriteableCollectionStream. The trait TWriteableCollectionStream depends on a new instance variable accessible through two accessors writeLimit and writeLimit:. This variable allows the internal collection to be bigger than the number of characters in the stream. This is a common technique used to avoid creation of a new collection each time an object is written to the stream. The TWriteableCollectionStream>>size returns the value of writeLimit and nextPut: writes its parameter at the right position in the collection. The trait also reimplements nextPutAll: for efficiency reasons.

4.2 Trait factories

The ANSI Smalltalk standard defines ReadStreamFactory>>on: and WriteStreamFactory>> with: to create new streams. Basically there are three places where the stream instance creation methods can be defined. The most two natural ones are on the traits TReadableCollectionStream and TWriteableCollectionStream or directly in the classes. Each solution has advantages and disadvantages. Adding the instance creation methods in the two traits helps their reuse. However, this forces all classes interested in these traits to have those same methods, even if they don't need them. If the instance creation methods are implemented in the classes, there will be duplication amongst the different classes.

We chose a third solution and implement the instance creation methods in separate traits. We named those traits "factories" because they support new stream creation.

We developed two factories: TReadableCollectionStreamFactory and TWriteableCollectionStreamFactory. The former implements on: and the latter implements on: and with:. Even if the ANSI Smalltalk standard does not define on: for writeable streams, we decided to implement it following the Squeak and VisualWorks implementations.

4.3 Classes

Traits alone are not enough to create a library. Classes are required to compose and create new instances. The original Squeak hierarchy provides three classes for collection-based streams: ReadStream, WriteStream and ReadWriteStream. Our implementation has equivalent classes with more explicit names: ReadableCollectionStream, WriteableCollectionStream and ReadWriteCollectionStream.

Those classes have nothing more to do than declaring the use of already defined traits, declaring some instance variables and implementing the required accessors.

The only difficulty arises with ReadWriteCollectionStream which has a conflict with the method size. The method size is implemented in both TReadableCollectionStream, obtained from TCollectionStream, and TWriteableCollectionStream. The first implementation reflects the size of the collection whereas the other takes care of the variable writeLimit and the efficient implementation in TWriteableCollectionStream. That's why ReadWriteCollectionStream has to use the implementation of TWriteableCollectionStream. To do this, the class removes the implementation of size coming from TReadableCollectionStream. This can be seen in Figure 5 on the arrow going from ReadWriteCollectionStream to TReadableCollectionStream[4].

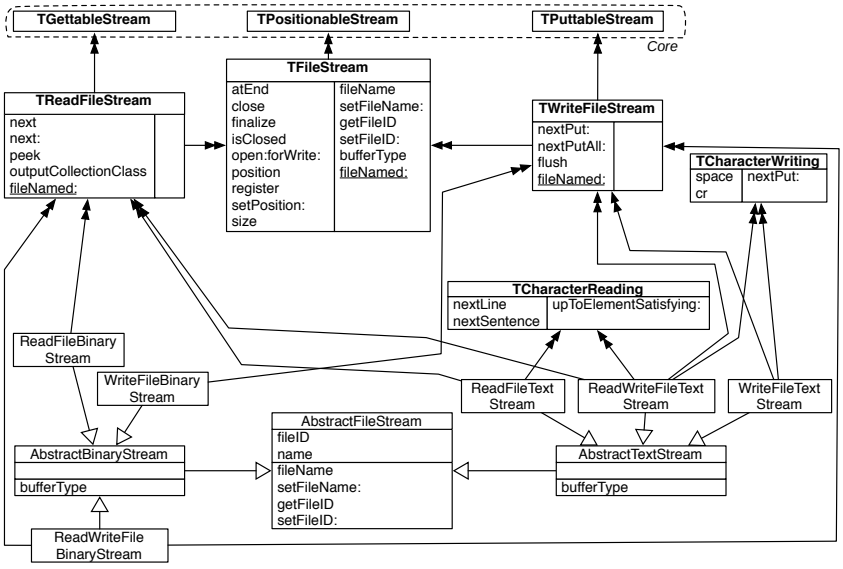

Fig. 6. FileStream implementation.

5 File-based streams

Nile includes a file-based stream library, shown in Figure 6. As with other file-based streams, it allows one to work with both binary and text files, supporting three access modes for each (read, write, and readwrite).

Each kind of file access is represented by a different class: the developer must explicitly choose the class based on what she wants to do with the file: reading, writing or both, in a binary or a text file. That way, the user has only the methods she can send in the interface of the stream. Note that this is a library design choice and it does not impact the way we decompose the behavior into traits.

Each file-based stream should be positionable, that's why the trait TPositionableStream is used by TFileStream. TFileStream is the common trait for all file-based streams. It implements base functionalities for file access and requires four accessors, a bufferType method and an instance creation method fileNamed. The private method bufferType is used to differentiate binary from text files.

The traits TReadFileStream and TWriteFileStream use the reusable traits TGettableStream and TPuttableStream from the core, respectively. They implement the required methods of these traits. Having implemented the reading and writing methods in separate traits instead of classes really helps here. This way, our

[4] The trait model gives the composer the possibility to remove methods through the minus (-) operator.

file-based streams only get the desired methods, not all methods like in the Squeak hierarchy.

At the very bottom of Figure 6, we defined the abstract classes AbstractFileStream, AbstractBinaryStream and AbstractTextStream to factorize instance variables definition and accessors. These abstract classes allow the definition of the six concrete classes with no more work.

Text-based streams use the traits TCharacterReading and TCharacterWriting depending on the type of file access. Even if simple, these two traits help defining methods only where they are needed and in all places where they are needed.

6 Other libraries

In this section we show how traits support reuse by presenting two libraries. We first present how character-related writing methods can be factored out in a trait. And then we describe the trait TDecoder that implements stream composition. Note that Nile offers several other libraries which are summarized later in Table 3.

6.1 Writing characters

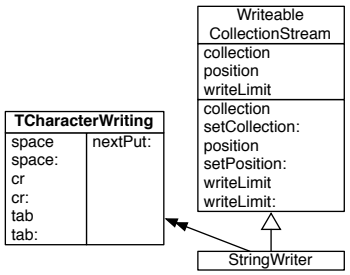

Fig. 7. Writing characters to a string.

In the Squeak hierarchy, the class WriteStream contains methods like space, cr and tab to write specific characters. These methods are only useful in case the user wants to write characters in her stream. If she wants to write binary data then those methods are useless and even pollute the interface of the stream. That's why we chose to implement the character-writing methods in a specific trait TCharacterWriting. Another advantage of using a specific trait is that Nile is then able to give those methods to any class which can write characters such as StringWriter in Figure 7 (a collection-based write stream which is writing characters) or WriteFileTextStream and ReadWriteFileTextStream in Figure 6.

6.2 Decoders

Developers often want to chain several streams. They want to use them like pipes that are connected together. For example, a developer may want a stream to read from a file and another stream which decompresses the first one on-the-fly. We generalized a mechanism which was already available in Squeak for classes like ZipWriteStream and have implemented a trait to support the composition of such decoders. We first present a scenario for such decoders and then describe our implementation.

A decoder is a GettableStream which reads its data from another GettableStream called its input stream. This way decoders can be chained. The decoder can do whatever it wants with the contents of its input stream: for example, it can ignore some elements, it can convert characters to numbers, it can compress or decompress...

Selective number reading. Imagine you have a string, or a file, containing space separated numbers. We can get all even numbers as presented in the code below. Here the developer composes three elementary streams which are subclasses of Decoder which uses the trait TDecoder.

```
| stream |
stream := ReadableCollectionStream on: '123 12 142 25'.
stream := NumberReader inputStream: stream.
stream := SelectStream selectBlock: [:each | each even] inputStream: stream.

stream peek.      ==> 12
stream next.      ==> 12
stream atEnd.     ==> false
stream next.      ==> 142
stream atEnd.     ==> true
```

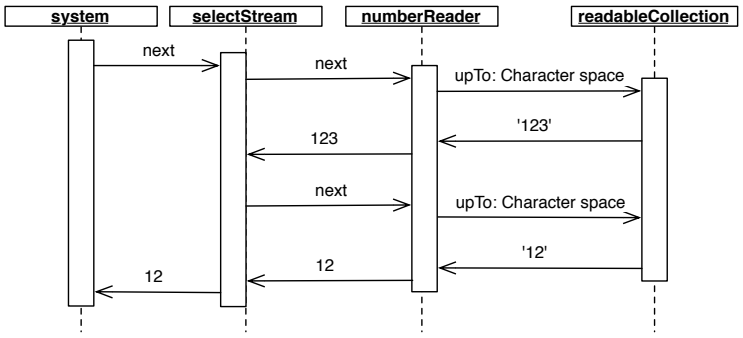

Fig. 8. Chaining streams

Figure 8 illustrates the stream connection. NumberReader transforms a character based stream in a number-based stream. SelectStream ignores all elements in the input stream for which the select block does not answer true.

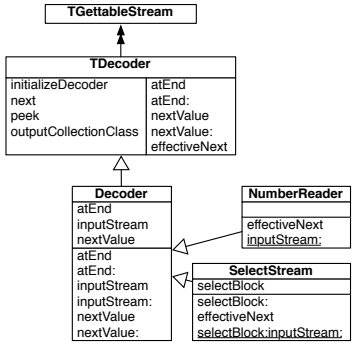

Fig. 9. The decoder and two possible clients.

The trait TDecoder. Figure 9 shows the decoder hierarchy. A decoder is basically a GettableStream, that's why TDecoder uses the trait TGettableStream. We chose to implement the decoding methods in a trait to let developers incorporate its functionalities into their own hierarchies.

TDecoder provides implementations for all required methods of TGettableStream (see Figure 4) but atEnd and it requires four accessors (including atEnd) and the method effectiveNext. This method effectiveNext is where all the work happens. It should read its input stream and return a new element. The method TDecoder>>next calls effectiveNext and catches StreamAtEndErrors for setting the atEnd variable.

Factoring the Nile core in traits proved again to be useful. If we had implemented it using single inheritance we would have been forced to choose a superclass between class Stream, which provides writing methods we don't want, or class ReadStream which only streams over collections, which is not what we want to do with decoders.

7 Discussions

This section compares Nile with other stream implementations, analyzes its performance and discusses where traits did and did not help us.

7.1 Comparison with previous work

There is no previous work building a library from scratch using traits. However, Schärli *et al.* [5] were the first to refactor the collection and stream hierarchies using traits.

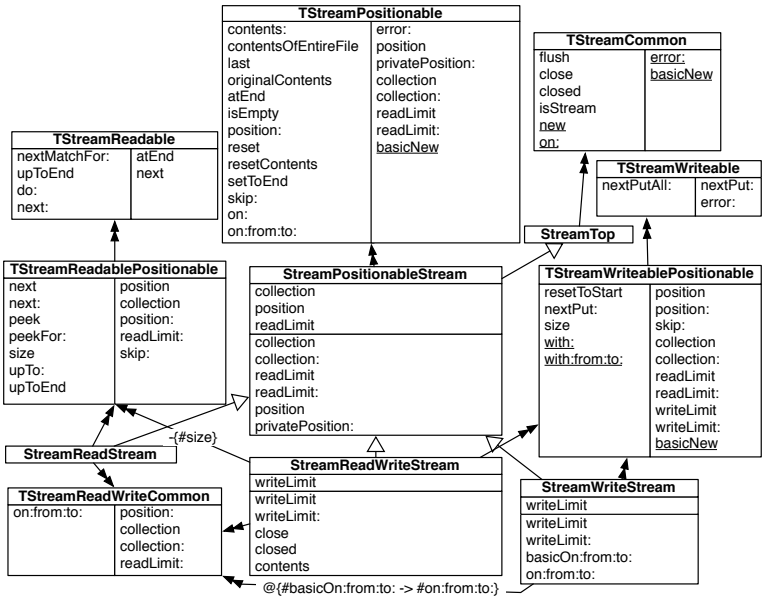

Fig. 10. Schärli's refactored stream hierarchy.

Figure 10 shows Schärli's stream hierarchy. Their work is a refactoring, where they took the original Squeak stream hierarchy and extracted the existing behavior into traits. This was a valuable experience that showed how a non-trivial implementation could be replaced with a cleaner implementation that was backwards compatible. While valuable, the backward compatibility constraint forces the result to be linked to the original implementation. Therefore it exhibits a number of problems:

- The positioning methods for a stream have to be based on collections because the methods position:, atEnd and setToEnd are all defined in the trait TStreamPositionable which depends on collection and collection:. Therefore it can not be used with files for example.
- The method TStreamReadablePositionable>>peek is dependent of the existence of methods collection and position but it shouldn't be.
- The granularity of the traits is big which hampers their reuse. For example, if we would like to have a skip: method, which is provided by TStreamPositionable,

we would get many more methods and, worse, we have to provide many collection-related methods.

7.2 Nile Analysis

The factories. Having implemented the factories in two separate traits complicates the hierarchy. Another solution would have been to define ReadWriteCollectionStream as a subclass of WriteableCollectionStream to inherit both instance creation methods directly. However we believe that having explicit traits is better, since they are potentially reusable.

Using an abstract superclass. Nile defines three concrete classes to stream over collections: ReadableCollectionStream, WriteableCollectionStream and ReadWriteCollectionStream. They all define the same instance variables and the same instance creation methods. To simplify the implementation of these classes, we could have implemented an abstract superclass for all of these classes with two common instance variables position and collection and their accessors. This is what we chose for the file-based streams (see Figure 11).

Classes vs. Traits. One of the key questions when building a system with traits is to decide when to use classes and when to use traits. In certain situations as illustrated by the Squeak stream hierarchy (see Section 2), defining a class or inheriting from a class does not make sense since some of its state is not used or its behavior should be canceled. This is a clear indication 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. In a lot of situations this means that traits are favored, since the price to pay to use traits is very low compared with the benefits one gets.

Reuse at Work. Figure 11 offers an overview of the core and some libraries of Nile. The fact that we based our implementation on traits rather than on inheritance and that we completely rethought the stream hierarchy leads to several advantages.

With Nile comes some really reusable traits which can be plugged in any other hierarchy. For example, implementing socket-based streams would only require socket manipulation work whereas utility methods like nextPutAll:, skip:, upToEnd are offered to the developer. Using the trait TGettableStream, a developer can easily implement a Random class which is basically a stream over random numbers. Table 3 presents the current clients we implemented in Nile using traits as well as the number of implemented methods to get the desired behavior.

Table 4 presents how much our core traits are reused. It presents for each traits the number of clients, the number of required methods and the number of methods that the trait provides. We see a good ratio provided/required for most of the traits. The ratio may still improve if additional behavior based on the core functionality is introduced.

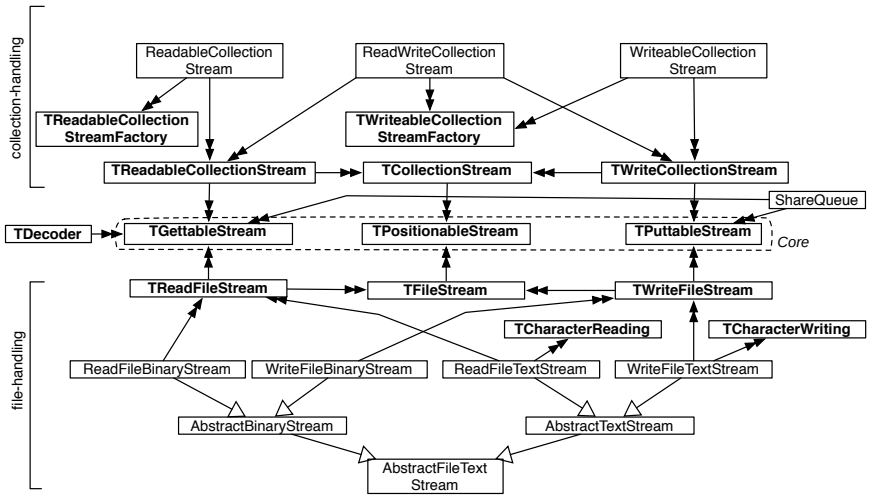

Fig. 11. An overview of Nile first clients

client name	superclass and trait used	met.	description
Random	TGettableStream	4	generate random numbers.
LinkedListStream	TGettableStream TPuttableStream	5	stream over linked elements.
History	TReadableCollectionStream TWriteableCollectionStream	7	manage do and undo of command objects.
SharedQueue	TGettableStream TPuttableStream	5	concurrent access on a queue.
StringReader	ReadableCollectionStream TCharacterReading	0	add character-based reading methods.
StringWriter	WriteableCollectionStream TCharacterWriting	0	add character-based writing methods.
CompositionStream	Decoder	1	multiplexer for input streams.
Tee	Decoder	1	fork the input stream (like the Unix *tee* command).
Buffer	Decoder	1	add a buffer to any kind of input stream.
NumberReader	Decoder	1	read numbers from a character based input stream.
SelectStream	Decoder	1	select elements from an input stream.
PipeEntry	TGettableStream TPuttableStream	7	allow data to be manually put into a pipe.

Table 3. Nile clients

Table 5 presents some metrics which compares the same functionalities in the Squeak implementation and in Nile for the collection-based streams. The first two metrics show that Nile uses a lot of traits and only a few classes. This is because Nile is designed to have fine grained and reusable units. The next two (number of methods and number of bytes) are more interesting and show that the amount of code is really smaller in Nile than in Squeak. Nile has 18% less methods and 15% less bytecodes than the corresponding Squeak collection-based stream library. Finally, we can deduce from the last metrics that the design of

Trait	client classes	required met.	provided met.	$\frac{provided}{required}$
TGettableStream	22	4	11	275%
TPositionableStream	20	3	9	300%
TPuttableStream	13	1	4	400%
TReadableCollectionStream	6	4	26	650%
TCollectionStream	12	4	11	275%
TWriteableCollectionStream	6	6	23	383%
TCharacterReading	3	2	1	50%
TCharacterWriting	3	1	8	800%
TByteReading	3	3	6	200%
TByteWriting	3	2	5	250%
TDecoder	7	6	14	233%

Table 4. Nile-trait reusability.

	Squeak	Nile	$\frac{Squeak-Nile}{Squeak}$
Number of Classes And Traits	5	13	-160%
Number of Classes	5	4	20%
Number of Methods	53	43	18%
Number of Bytes	1725	1459	15%
Number of Cancelled Methods	2	0	100%
Number of Reimplemented Methods	14	3	78%
Number of Methods Implemented Too High	10	0	100%

Table 5. Some metrics for the collection-based streams

Nile is better: there is no cancelled method nor method implemented too high and there are only four methods reimplemented for speed reason compared to the fourteen of the Squeak version.

About Trait Composition. During trait composition it is possible that required methods of a trait are fulfilled by the provided methods of another traits. When this happens the developer does not have to do any extra work and benefits from the composition result. We can see this at work for the method atEnd that is required in TGettableStream and provided by TPositionableStream. The trait TReadableCollectionStream doesn't have any work to get the implementation of atEnd. However, such a situation is rare and based on the decomposition of traits using a compatible behavior and vocabulary.

However, it is sometimes better or necessary to override a method coming from a trait. It is because the new implementation have more knowledge than the overridden one and thus can do a better job. For example, the method TReadableCollectionStream>>skip: overrides the method TGettableStream>>skip:. The new method is more efficient because the stream is positioned directly, needing only a small bound computation:

```
TReadableCollectionStream>>skip: amount
  "Moves relatively in the stream. Go forward amount elements. Go backward if amount is
   negative. Overrides TGettableStream>>skip: for efficiency and backward possibility."
```

```
self position: ((self position + amount) min: self size max: 0)
```

Moreover, skip: is now able to go backward if amount is negative, which was not the case in the implementation of TGettableStream.

7.3 Performance optimization

One of the key challenge of Nile in terms of performance is to be able to iterate over any kind of collection while at the same time be as efficient as the squeak implementation for Arrays and Strings. We present our solution to this challenge.

Contrary to the Squeak class WriteStream, Nile's TWriteableCollectionStream is able to iterate over any kind of SequenceableCollection. In Squeak the method WriteStream>>nextPutAll: directly manipulates its internal collection using a primitive call to replaceFrom:to:with:startingAt: implemented in String and Array[5], Nile has more work.

The idea is to propose a dedicated set of classes working specifically on Array and String. We first reimplemented the method nextPutAll: in TWriteableCollectionStream to take care of any kind of collection. This proved to be slow when iterating over Arrays and Strings compared to Squeak. Benchmarking shows that too much time was lost into calling methods. We have then implemented an optimized version (*i.e.,* using the primitive mentioned above) of nextPutAll: directly into the classes ReadWriteArrayStream and WriteableArrayStream in which we are sure that the underlying collection is an Array (as shown on the left side of Figure 12).

Accessor-use impact. Traits cannot define state, which must be accessed via accessor methods. As Squeak does not have a JIT compiler, using accessors instead of direct instance variable access has a cost. Table 6 shows that using accessors in the context of stream on strings and arrays is 41% times slower than direct instance variable access. To optimize our library as much as possible we used direct accesses, *i.e.,* as shown on the left of Figure 12 we redefined nextPutAll:. However this has as impact that we have to duplicate the optimized implementation of nextPutAll into WriteableArrayStream and ReadWriteArrayStream.

	execution (per second)	$\frac{nile}{squeak}$
Squeak implementation	126	
Nile with direct variable access	138	110%
Nile with accessors	81	64%

Table 6. Nile performances. Without accessors, Nile is faster than Squeak. But using them makes it slower.

[5] While replaceFrom:to:with:startingAt: is implemented for all kinds of SequenceableCollections, it does not work for OrderedCollection.

The right of Figure 12 presents another solution we implemented using an extra trait to share the optimized method for the two classes (*i.e.,* calling the primitive). However we discarded this solution since it is slower because traits forced us to use accessors.

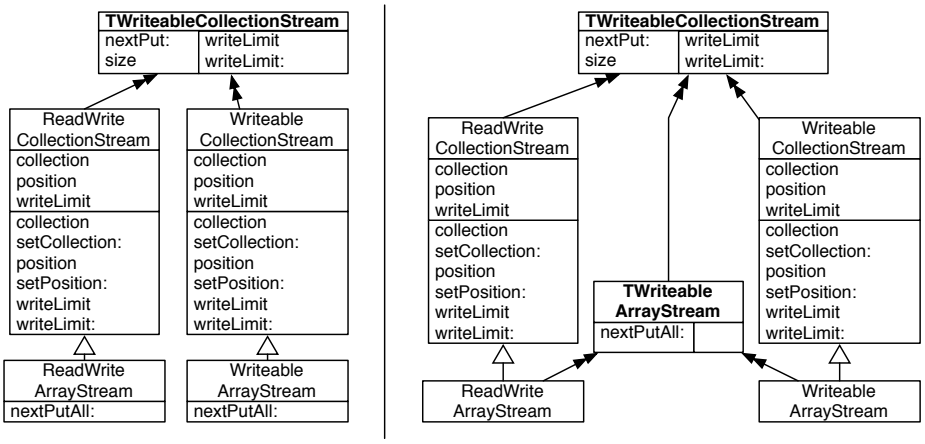

Fig. 12. Two solutions to optimize nextPutAll:.

8 Problems with traits

Since one of the original goal of the work presented in this paper is to identify potential problems with traits, we now report the problems we faced while developing Nile. Note that some problems are not trait specific but due to Smalltalk's lack of visibility controls. In addition it should be noted that we did not encounter problem with aliasing in the context of recursive calls which is a known problem of traits.

8.1 Interface pollution

In this section we present some problems due to class interface extension.

Required accessors. With stateless traits, it is not possible to add state, *i.e.,* instance variables, to traits. Instead, the developer must add required accessors to its trait and the classes will implement those required accessors and the instance variable. This is a problem because the accessors are then part of the interface of the classes and this adds a burden to the class developers. However this would be solved if Smalltalk would have method access control. Stateful traits

[3] solve this problem by allowing traits to contain private state. For example, if we had used stateful traits the methods TPositionableStream>>setPosition: and TWriteableCollectionStream>>writeLimit would not have been required.

However, developing Nile showed that stateful traits would not have been of great help. If we examine the trait TCollectionStream in Figure 5, we can see that implementing an instance variable collection here would have been interesting because classes would not have needed to define it. But, methods collection and setCollection: would still need to be in the interface because TReadableCollectionStream>>outputCollectionClass and TReadableCollectionStreamFactory>>on: need them.

We believe that stateful traits are not as interesting as what a first impression might tell.

Lazily initialized variable. There are basically three ways of initializing an instance variable giving it a first value: initializing lazily the variable in the accessor, using an initialize method, or initializing the variable in the instance creation method through an accessor.

Lazy initialization is a common programming pattern. Here is an example in Smalltalk which returns the value of the variable checked if it has been set or sets it to false and returns false:

```
checked
  ^ checked ifNil: [checked := false]
```

Now, imagine a trait needs a variable and a default value. Since traits can't contain state in their standard implementation, accessors must be required methods. But where do you lazily initialize the variable? Two solutions are possible: you can force users of the trait to initialize the variable or you can initialize in the trait and use another method for accessing the variable. Here is an example of the later possibility:

```
checked
  ^ self getChecked ifNil: [self checked: false. false].

checked: aBoolean
  self explicitRequirement.

getChecked
  self explicitRequirement.
```

This solution pollutes the trait interface with an unnecessary method getChecked. The other solution consists of letting the trait user initialize the variable. This solution does not pollute the interface but gives more responsibility to other developers and may produce code duplication or bugs.

The same problem appears when you want to do some checking before assigning to a variable as shown in TPositionableStream>>position: for example:

```
TPositionableStream>>position: newPosition
  "Sets the  number of elements before  the position to  be the parameter
   newPosition. 0  for the start  of the stream.  Throws an error  if the
   parameter is lesser than 0 or greater than the size."

   (self isInBounds: newPosition) ifFalse: [InvalidArgumentError signal].
   self setPosition: newPosition.
```

This setter needs an additional method setPosition: which really modifies the variable and which is a required method of the trait.

Initializing a trait. In a class, when a developer wants to initialize a newly created object, he can use an initialize method:

```
initialize
  super initialize.
  color := Color transparent.
```

This can be done in a trait too provided that the developer uses an accessor instead of a direct reference to the variable. Problems arise when a class or a trait uses multiple traits, each defining its own initialize method. In this case, there will be conflicts and those conflict can only be resolved by aliasing. This brings lots of pollution in the class interface and require a lot of work.

Another solution would be to use a specific name for each method initialize. For example, it the trait TPositionableStream needs an initialize method, the developer can name it initializePositionableStream. Each user of the trait now needs to define its own initialize method which calls initializePositionableStream. We believe this is still clunky and requires too much work from the developer.

Initializing in the instance creation method. Instance creation methods can be used to initialize variables. This is what we did for Nile:

```
TWriteableCollectionStreamFactory>>on: aCollection
  ^ self basicNew
      initialize;
      setCollection: aCollection;
      writeLimit: 0;
      reset;
      yourself
```

Smalltalk is made such that this requires that setters are available in the interface of the class. It also puts more responsibility on the instance creation method which now needs more knowledge over the class it instantiates.

8.2 Methods silently ignored

Sometimes, modifying a trait does not modify the users of this trait in the same way because of name overriding. Note that this problem is not trait specific but it is a problem of object-oriented programming as shown by Figure 13.

Figure 13 shows a part of our test hierarchy for Nile. The test hierarchy is very similar to the Nile hierarchy: for each model trait or class, there is a test trait or class. The method nextPutAll: is tested in two different places: in the methods TPuttableStreamTest>>testNextPutAll1 and TWriteableCollection-StreamTest>>testNextPutAll2. If a tester adds a new test named testNextPutAll2 in the trait TPuttableStreamTest, then the test is silently ignored and will never be launched.

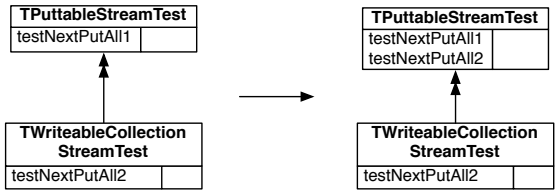

Fig. 13. If the tester implements TPuttableStreamTest>>testNextPutAll2, the test will never be launched because TWriteableCollectionStreamTest>>testNextPutAll2 hides it.

9 Related work

We already compared our approach with the few work refactoring existing code using traits [5, 6]. 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 [9] developed incremental FCA algorithms to infer implementation and interface hierarchies guaranteed to have no redundancy. To assess their solutions from a point of view of complexity and maintainability they propose a set of structural metrics. They analyze the Smalltalk Collection hierarchy. One important limitation is that they consider each method declaration as a different method and thus cannot identify code duplication. Moreover their approach serves rather as a help for program understanding than reengineering since the resulting hierarchies cannot be implemented in Smalltalk because of single inheritance.

In C, Snelting and Tip analyze a class hierarchy making the relationship between class members and variables explicit [10]. By analyzing the *usage* of the hierarchy by a set of client programs they are able to detect design anomalies such as class members that are redundant or that can be moved into a derived class. Taking into account a set of client programs, Streckenbach infer improved hierarchies in Java with FCA [11]. Their proposed refactoring can then be used for further manual refactoring. The tool proposes the reengineer to move methods up in the hierarchy to work around multiple inheritance situations generated by the generated lattice. The work of Streckenbach is based on the analysis of the

usage of the hierarchy by client programs. The resulting refactoring is behavior preserving (only) with respect to the analyzed client programs.

Lienhard *et al.* applied Formal Concept Analysis to semi-automatically identify traits [6]. We cannot really compare their resulting hierarchy with ours since the information about the respective traits is no longer available. However we can conclude that the resulting hierarchy was limited and resulted only from a refactoring effort and not from a new design.

Interfaces and specifications of the Smalltalk collection hierarchy are also analyzed by Cook [12]. He also takes method cancellation into account to detect protocols. By manual analysis and development of specifications of the Smalltalk collection hierarchy he proposes a better protocol hierarchy. Protocol hierarchies explicitly represent similarities between classes based on their provided methods. Thus, compared to our approach, protocol hierarchies present a *client* view of the library rather than one of the *implementor*.

Moore [13] proposes automatic refactoring of Self inheritance hierarchies. Moore focuses on factoring out common expressions in methods. In the resulting hierarchies none of the methods and none of the expressions that can be factored out are duplicated. 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. Moore's analysis finds some of the same problems with inheritance that we have described in this paper, and also notes that sometimes it is necessary to manually move a method higher in the hierarchy to obtain maximal reuse.

Casais uses an automatic structuring algorithm to reorganize Eiffel class hierarchies using decomposition and factorization [14]. 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 [15].

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 do rely on algorithms. This is important since we want to be able to use our result to compare it with the result of future approach extracting traits automatically, so the Nile library may serve as a reference point.

10 Conclusion

Traits are units of reuse that can be used to compose classes. This paper is an experience report. Even if other experiences have been made to test traits, they were always refactoring an existing hierarchy, moving methods from classes to traits. Our work however presents a brand new implementation. We started from the textual description from the ANSI Smalltalk standard and from existing implementations of stream libraries in Squeak and VisualWorks. Our result is a

completely new implementation, named Nile, of the stream hierarchy which does not share any code with previous implementations.

Our experience shows that traits are good building blocks which favor reuse across different hierarchies. In the present implementation of Nile we get up to 15% less code than the corresponding Squeak code. Core traits are reused by numerous clients. We also presented the problems we faced during the experience and believe that Nile can be used in the future as a reference point for comparing future trait enhancement.

This experience shows that well defined traits can naturally fit into lots of different clients which can benefit from methods offered by the trait for a relatively low cost.

Acknowledgment

We gratefully acknowledge the financial support of the Agence Nationale pour la Recherche (ANR) for the project "Cook: Rearchitecting object-oriented applications" (2005-2008).

References

1. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.: Traits: Composable units of behavior. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP'03). Volume 2743 of LNCS., Springer Verlag (2003) 248–274
2. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.: Traits: A mechanism for fine-grained reuse. ACM Transactions on Programming Languages and Systems (TOPLAS) **28** (2006) 331–388
3. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Stateful traits. In: Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006). Volume 4406 of LNCS., Springer (2007) 66–90
4. Ducasse, S., Wuyts, R., Bergel, A., Nierstrasz, O.: 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), New York, NY, USA, ACM Press (2007) To appear.
5. Schärli, N., Ducasse, S., Nierstrasz, O., Wuyts, R.: Composable encapsulation policies. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP'04). LNCS 3086, Springer Verlag (2004) 26–50
6. Lienhard, A., Ducasse, S., Arévalo, G.: Identifying traits with formal concept analysis. In: Proceedings of 20th Conference on Automated Software Engineering (ASE'05), IEEE Computer Society (2005) 66–75
7. 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.
8. 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: Proceedings OOPSLA '97, ACM SIGPLAN Notices, ACM Press (1997) 318–326
9. 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** (1998) 117–134

10. Snelting, G., Tip, F.: Reengineering Class Hierarchies using Concept Analysis. In: ACM Trans. Programming Languages and Systems. (1998)
11. Streckenbach, M., Snelting, G.: Refactoring class hierarchies with KABA. In: OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2004) 315–330
12. Cook, W.R.: 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., ACM Press (1992) 1–15
13. Moore, I.: Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In: Proceedings of OOPSLA '96 (11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications), ACM Press (1996) 235–250
14. Casais, E.: Automatic reorganization of object-oriented hierarchies: A case study. Object-Oriented Systems **1** (1994) 95–115
15. Dicky, H., Dony, C., Huchard, M., Libourel, T.: On Automatic Class Insertion with Overloading. In: Proceedings of OOPSLA '96 (11th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications), ACM Press (1996) 251–267