Traits at Work: the design of a new trait-based stream library

Published in *Computer Languages, Systems and Structures*
September 2008 ⋆,⋆⋆

Damien Cassou [a] Stéphane Ducasse [b] Roel Wuyts [c]

[a] *INRIA-Futurs Bordeaux*

[b] *INRIA-Lille Nord Europe, Adam Team, CNRS 8022 - LIFL/UTSL*

[c] *IMEC, Kapeldreef 75, B-3001 Leuven and KU Leuven*

Abstract

Recent years saw the development of a composition mechanism called Traits. Traits are pure units of behavior that can be composed to form classes or other traits. The trait composition mechanism is an alternative to multiple or mixin inheritance in which the composer has full control over the trait composition.

To evaluate the expressiveness of traits, some hierarchies were *refactored*, showing code reuse. However, such large refactorings, while valuable, may not exhibit all possible composition problems, since the hierarchies were previously expressed using single inheritance and following certain patterns.

This paper presents our work on designing and implementing a new trait-based stream library named Nile. It evaluates how far traits enable reuse, what problems can be encountered when building a library using traits *from scratch* and compares the traits solution to alternative composition mechanisms. Nile's core allows the definition of compact collection and file streaming libraries as well as the implementation of a backward-compatible new stream library. Nile method size shows a reduction of 40% compared to the Squeak equivalent. The possibility to reuse the same set of traits to implement two distinct libraries is a concrete illustration of trait reuse capability.

Keywords. Object-Oriented Programming, Inheritance, Refactoring, Traits, Code Reuse, Smalltalk

⋆ We gratefully acknowledge the financial support of the Cook ANR French project.
⋆⋆This work is based on an earlier work: "Redesigning with Traits: the Nile Stream trait-based Library", in Proceedings of the 2007 International Conference on Dynamic Languages (ESUG/ICDL 2007) http://doi.acm.org/10.1145/1352678.1352682 © ACM, 2007

Email addresses: damien.cassou@inria.fr (Damien Cassou), stephane.ducasse@inria.fr (Stéphane Ducasse), Roel.Wuyts@imec.be (Roel Wuyts).

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 and the composer is empowered 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 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*. The reader unfamiliar with traits may read the appendix A for a rapid introduction to stateless traits.

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 taking *existing* single inheritance systems as input. Usability and reuse of traits when developing a *new* system has not been assessed. Implementing a non-trivial 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 new 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 answer the following questions that quickly arise when using traits in practice:

- What is a good granularity for traits that maximizes their reusability and composition? *(See Section 7.1)*
- Can we identify guidelines to assess when trait composition should be preferred over inheritance? *(See Section 7.2)*
- Are traits better reusable than classes? *(See Section 7.3)*
- Can an application still be made efficient when using traits composition? *(See Section 7.4)*

- What trait limits and problems do we encounter? *(See Sections 7.5 and 7.6)*

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 in terms of implemented protocols with the current Smalltalk implementations. The choice for a stream library was motivated by a number of reasons:

- streams are naturally modeled using multiple inheritance. When implemented in a single inheritance language developers need to resort to duplicating code and canceling inherited methods;
- N. Schärli [5] and A. Lienhard [6] already refactored the Stream library using traits so we can directly compare their refactoring results with a design based from the start on traits;
- streams are an important abstraction of computer language libraries;
- being a real-world implementation, several practical constraints are imposed on the stream library: ANSI Smalltalk standard compatibility is required and backward-compatibility with existing Smalltalk dialects is needed.

Nile is structured around just six 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 any needed functionality by composing with the relevant part of the core. Moreover, Nile has neither canceled methods nor methods implemented too high in the hierarchy. There are only three method overridden compared to the fourteen of Squeak [1] . In addition the exact same library core supports the building of backward-compatible libraries while at the same time supporting more compact ones. This design shows that traits are good unit of code reuse. In addition, Nile's compact design has 40% less methods and 39% less bytecodes than the existing single-inheritance Squeak collection-based stream library.

This article is an extension of our previous work [8]. The differences between the current paper and our previous work are (1) a large rethought of Nile's core, (2) the possibility to use the new core to express compact collection-based and file-based stream libraries as well as the implementation of a backward-compatible new stream library, and (3) a new analysis based on the new core.

The contributions of the paper are: (1) the design description 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 traits.

[1] Overridden methods are methods implemented in class A and reimplemented in B (subclass of A) without invoking the method in A.

We start by presenting the limits of the existing Squeak Stream hierarchy and the ANSI Smalltalk standard protocols (Section 2). Section 3 presents an overview of Nile and the library core with its most important traits. Section 4 details the implementation of the collection-based and file-based stream libraries. Two other libraries are presented in Section 5. Section 6 presents a backward-compatible stream library based on the exact same core, illustrating the trait reuse power. Section 7 answers the questions regarding traits usage and limits. Section 8 compares our approach with the one of N. Schärli [5] and with other related work. Section 9 concludes the paper.

2 Current stream library analyses

Streams are a well-known data structure, present in most programming language libraries. They provide operations to stream over data (typically with an operation to get the next element in the stream). One of the primary usages of streams is in IO, where they are used as an abstraction to read or write data to various outputs, such as the console or files. Streams are by nature a sequential access data structure, but some are random access data structures (file streams can be positionable, for example, in order to read or write data in a specific position in the file). Stream libraries in Smalltalk support random access streams through the SequencedStream abstraction.

In this section, we analyze the existing stream hierarchy of the open-source Smalltalk Squeak [9]. We highlight the key problems and present the ANSI Smalltalk standard.

2.1 Analysis of the Squeak stream hierarchy

Squeak [9], 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 a similar design with slightly different implementations: 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.

The existing single-inheritance implementation has different problems (methods implemented too high, unused superclass state, simulating multiple inheritance and duplicated code) that we detail in the next sections.

Methods implemented too high in the hierarchy. A common technique

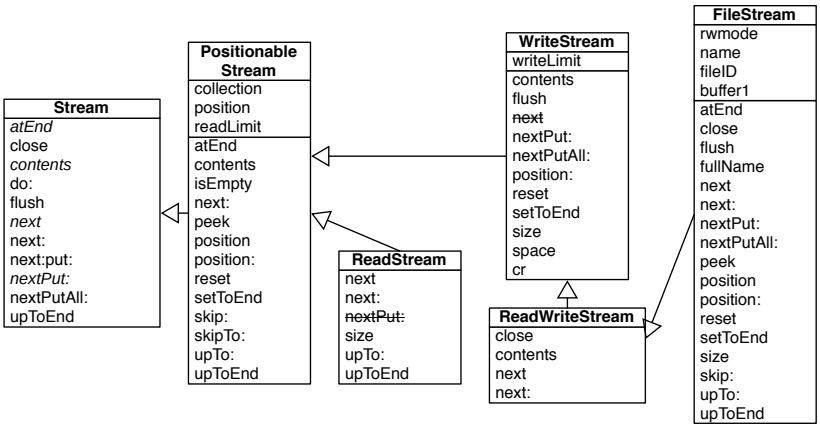

Fig. 1. The Squeak core Stream hierarchy. Only the most important methods are shown.

to avoid duplicating code consists in implementing a method in the topmost common superclass of all classes which need this method. This technique pollutes the interface of classes which do not want this method but inherit it along with methods they need. 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. The problem is that some subclasses are used for read-only purposes, like ReadStream. Those classes must cancel explicitly the methods they don't want [2] . This approach, even if it was probably the best available solution at the time of the first implementation, has several 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

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

to stream over collections (see Figure 1). It inherits three instance variables (collection, position and readLimit) from PositionableStream and one (writeLimit) from WriteStream. None of these four variables is used in FileStream or in any of its subclasses.

Simulating multiple inheritance by copying. ReadWriteStream is conceptually both a ReadStream and a WriteStream. However, Smalltalk is a single inheritance language, so ReadWriteStream can only subclass one of these. The other behavior 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 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. Figure 1 shows 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

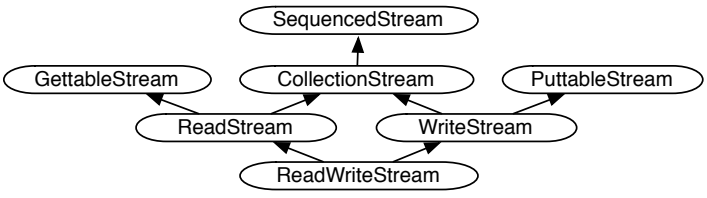

Fig. 2. The ANSI Smalltalk standard stream protocol hierarchy.

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: SequencedStream, PuttableStream and GettableStream. Table 1, Table 2 and Table 3 summarize the protocol contents.

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.

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

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 2
The 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 3
GettableStream protocol defined by the ANSI Smalltalk standard.

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, with the exception of the peekFor: method.

About the method peekFor: in GettableStream. The standard proposes

a definition of peekFor: that most Smalltalk implementations do not follow. In the ANSI Smalltalk standard, peekFor: is equivalent to an equality test between the peeked object and the parameter:

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

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 six traits (See Figure 3) 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. The other libraries we present in the paper are support for decoders, streams that can be chained and history stream. In addition, the exact same core is used to implement a backward-compatible version of the stream library presented in Section 6. By backward-compatible we mean that

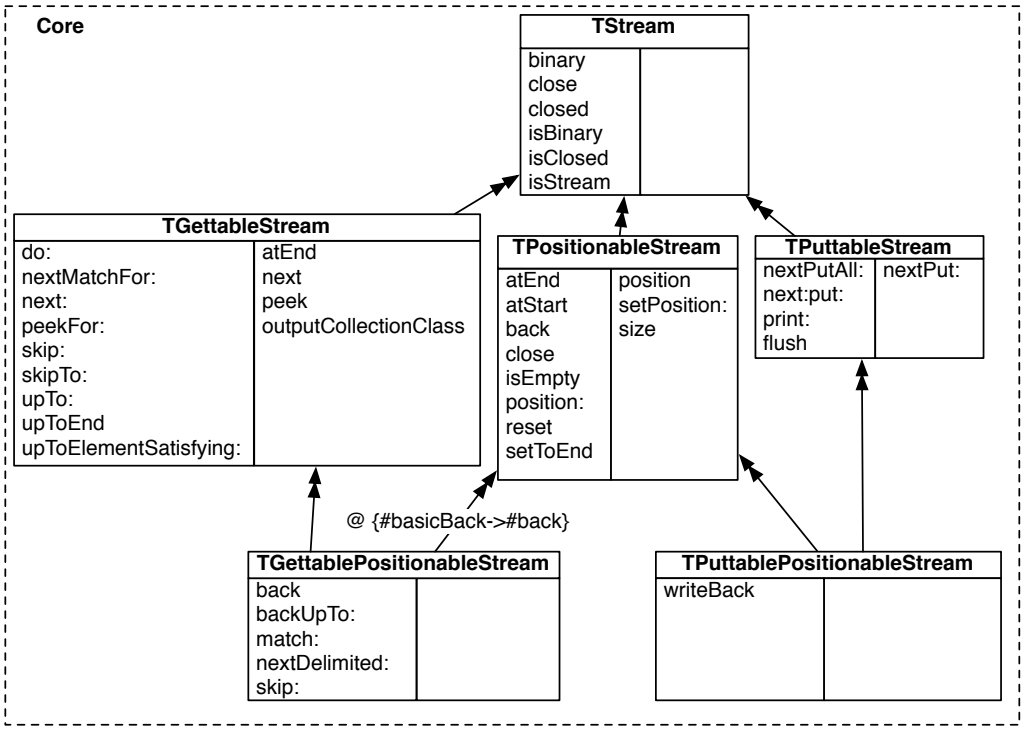

Fig. 3. Nile core composed of six traits. — We use a UML-based notation to represent traits: methods on the left are *provided* and methods on the right are *required*.

the solution follows the original Smalltalk class level decomposition described in the Stream class hierarchy [10,7]. Such hierarchy was mainly composed of ReadStream, ReadWriteStream and WriteStream.

We designed Nile around three traits reflecting the ANSI Smalltalk standard: TPositionableStream, TGettableStream and TPuttableStream. To increase reuse we added three other traits: TStream, TGettablePositionableStream and TPuttablePositionableStream. The two traits TGettablePositionableStream and TPuttablePositionableStream have been implemented to group behavior where both positioning and reading (or writing) was available. Also, a trait named TStream is implemented as a placeholder to provide behavior common to all streams. A description of each trait follows.

TStream. The trait TStream implements methods like isStream and isClosed which must be available for all traits.

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 outputCollectionClass in TGettableStream is used to determine the type of collection which is used when returning collections of

elements, and is for example used in methods next: and upTo:.

TPositionableStream. The trait TPositionableStream allows for the creation of streams that are positioned in an absolute manner. It corresponds to the ANSI Smalltalk standard SequencedStream 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 bounds 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 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 provide their own implementations.

TGettablePositionableStream. This trait allows streams to be readable and positionable. It uses the two traits TGettableStream and TPositionableStream. It implements different methods available when both of these features are available. The methods back and skip: override methods in TPositionableStream and TGettableStream respectively. The method back now returns the element before the current position. This is not possible in TPositionableStream because this trait is not able to read elements. The implementation of this method uses the basicBack: alias to call the previous implementation and then returns the result of sending peek. The method skip: which was naively implemented as a succession of nexts in TGettableStream, is now implemented efficiently using position:.

TPuttablePositionableStream. This trait implements behaviour that needs writing and positioning. The method writeBack: is equivalent to a call to back followed by a call to nextPut:.

4 Collection-based and file-based streams

This section presents two libraries based on the core described earlier: the collection-based and file-based streams. The resulting implementation is shown in Figure 4.

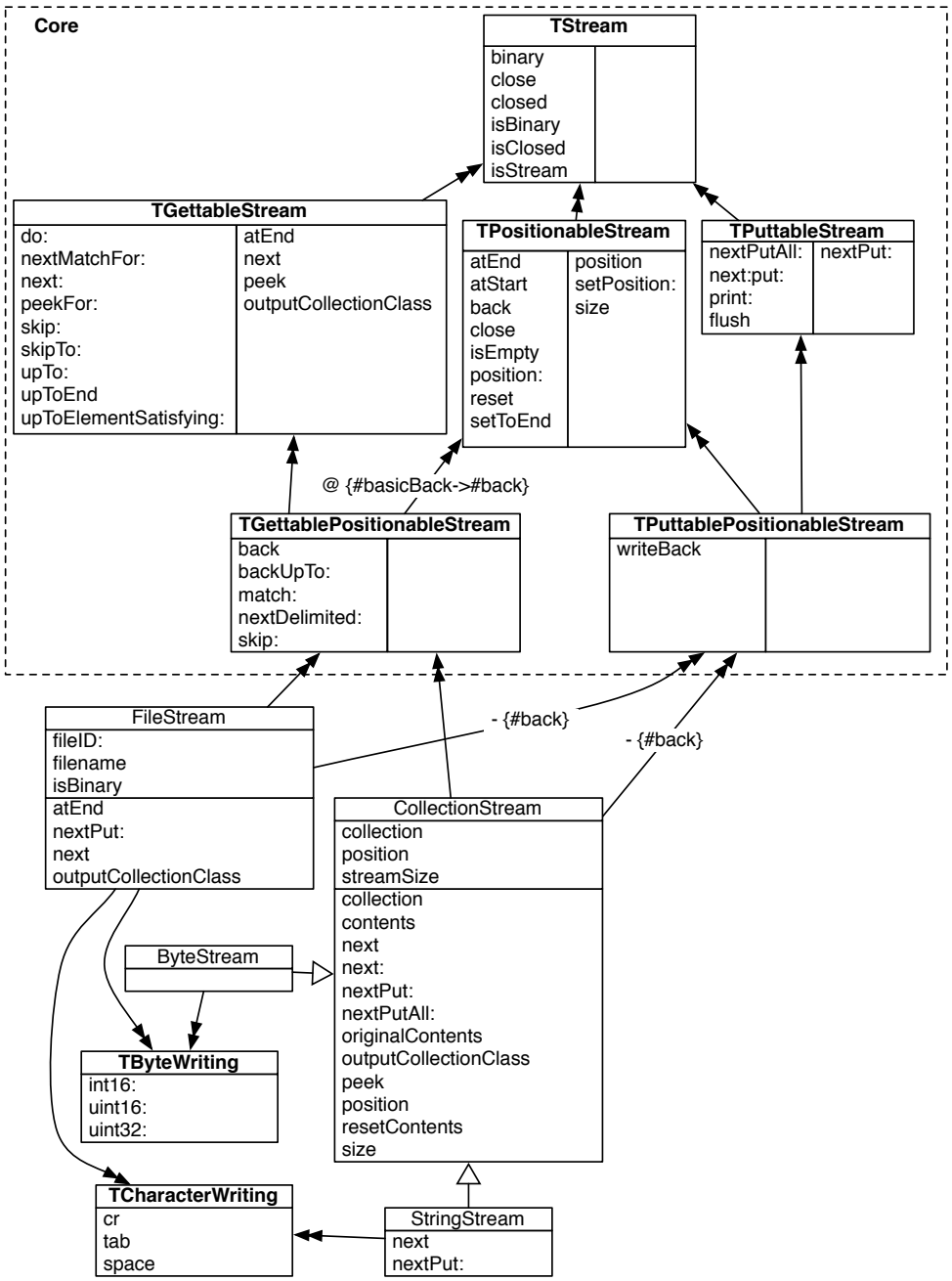

Fig. 4. Nile's core with two libraries: CollectionStream and FileStream.

4.1 Collection-based streams

With the core traits in place, the implementation of the collection-based stream library becomes straightforward. The library implements all collection-oriented methods in a single class: CollectionStream. This class replaces the three classes ReadStream, ReadWriteStream and WriteStream of the traditional Smalltalk stream hierarchy (See Figure 1). The reasons of this design decision

are explained in Section 7. The class declares three instance variables (namely collection, position and streamSize) and their accessors, and implements all required methods (like next, outputCollectionClass and peek). The variable streamSize allows the internal collection to be larger than the number of elements in the stream. This is a common technique used to avoid the creation of a new internal collection object for each single object written to the stream. The method size returns the value of this instance variable.

There is a method conflict in CollectionStream because of two implementations of the back method: one coming from TGettablePositionableStream and the other one coming from TPositionableStream through TPuttablePositionableStream. The one we want is the one from TGettablePositionableStream which is an improved version, returning the element before the current position (see Section 3). That is why the class CollectionStream removes the method back when it declares the use of trait TPuttablePositionableStream.

Streams of Strings are used much more than any other collection-based stream. For example, the Smalltalk systems uses them a lot internally to parse and concatenate strings. We decided to implement a specific stream, named StringStream for Strings as shown in Figure 4. There are two reasons for this choice: (1) to be able to make specific speed improvements and (2) to add character-related methods that are only valid for string-based streams and that should not pollute general collection streams. The character-related methods (like tab and cr) put non-printable characters in the streams. They are implemented in a reusable trait named TCharacterWriting. We chose a trait over a direct implementation in the class to be able to reuse them in file-based streams also.

For the same reasons, we implemented the trait TByteWriting to manipulate streams of bytes. The class ByteStream uses this trait to stream over collection of bytes.

Note that, in contrast to the default Squeak implementation [11] and similar to VisualWorks, our implementation actually works with any sequenceable collection, not just Arrays and Strings, which is a real limitation of the Squeak implementation.

4.2 File-based streams

Nile includes a file-based stream library, shown in Figure 4. 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).

The implementation of this part of the library requires only one class: FileStream.

This class is responsible for reading and writing in both text and binary files. We chose to implement all features in just one class to ease use of this library. This choice is the same as the one made by Squeak.

FileStream uses the traits TByteWriting and TCharacterWriting to ease writing in binary and text files respectively.

5 Other libraries

In this section we show how traits support reuse by presenting two small libraries. We first present a *history stream* which encapsulates undo/redo logic. Then we describe the trait TDecoder that implements stream composition. Note that Nile offers several other libraries which are summarized later in Table 4.

5.1 History

It is often tedious to handle properly a history mechanism with back and forward actions, like in internet browsers for example. This is because, as a developer, you have to take care of the underlying collection, your position in the collection and the modifications to this collection. Nile provides a history stream that encapsulates such a behavior. Such a stream can then manipulate any kind of objects such as web pages or Commands (design pattern [12]) to easily implement undo/redo behaviour. The following operations are possible in a history:

- positioning operations to go forward and backward by one element.
- an operation to insert an element at the current position. Existing elements after the current position are then removed (the inserted element replaces them).

This behavior is illustrated in Figures 5–6.

Fig. 5. A new history is empty. The user opens to page 1. — The user clicks on a link to page 2. — The user clicks on a link to page 3.

A history is a gettable and puttable stream which is not positionable, *i.e.,* we want to go forward and backward, and we want to be able to add elements, but we do not want to let the user go to a specific position directly. The History

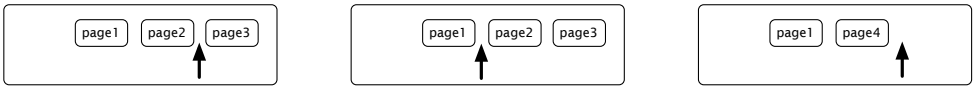

Fig. 6. The user clicks on the back button. He is now viewing page 2 again. — The user clicks again the back button. Page 1 is now displayed. — From page 1, the user clicks on a link to page 4. The history forgets pages 2 and 3.

class benefits from the traits TGettableStream and TPuttableStream but could not be a subclass of CollectionStream because we do not want methods that offer direct positioning behaviour to pollute the interface. Internally, however, the history is implemented using a collection-based internal stream. Figure 7 presents a class diagram of our implementation.

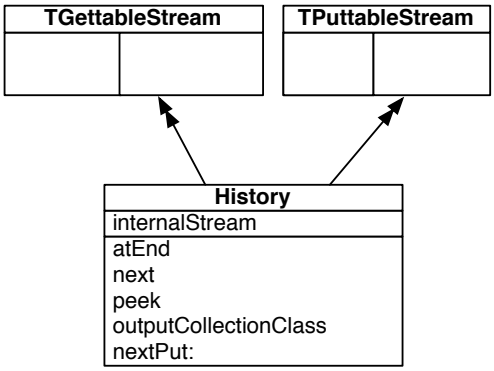

Fig. 7. The class History

The implementation of the class History reuses the traits TGettableStream and TPuttableStream. The class declares an instance variable internalStream which will contain a CollectionStream.

The implementation of the four required methods from the trait TGettableStream are straightforward and simply delegate to the internal stream. For example, here is the implementation of next:

```
History>>next
  ^ internalStream next
```

The implementation of method nextPut: is a little bit more difficult. Imagine a user goes backward a few times before clicking on a link to go to another page. It is then no longer possible to go forward again. This mechanism is implemented in the method nextPut::

```
History>>nextPut: anObject
  | result |
  result := internalStream nextPut: anObject.
  internalStream streamSize: internalStream position.
```

```
^ result
```

Setting the stream size to the current position prevents the user from going forward after this call.

5.2 Decoders

Developers often want to chain several streams to use them like connected pipes. 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 a clean implementation using traits.

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, or it can compress or decompress the input.

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 the class 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
```

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

The trait TDecoder. Figure 9 shows the decoder hierarchy. A decoder is a GettableStream and therefore class TDecoder uses the trait TGettableStream. We chose to implement the decoding methods in a trait to let developers

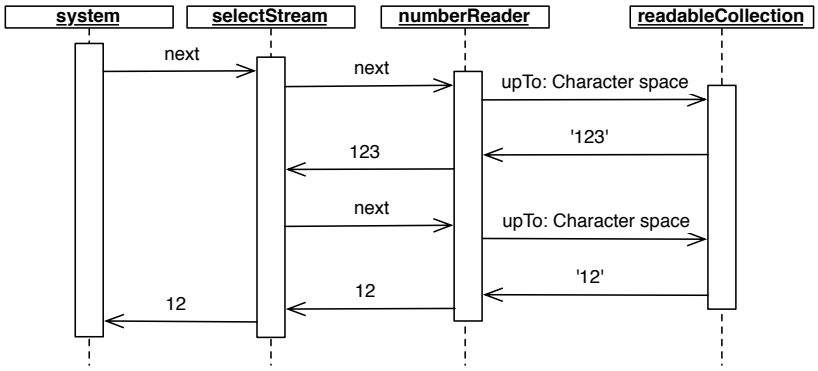

Fig. 8. Chaining streams

incorporate its functionalities into their own hierarchies.

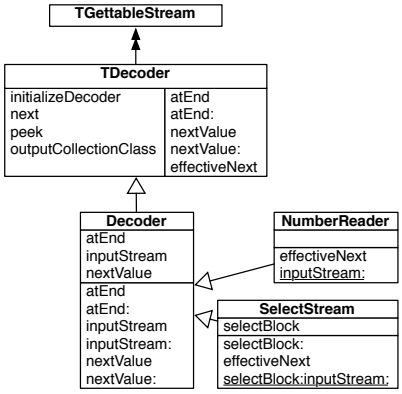

Fig. 9. The decoder and two possible clients (underlined methods are instance-creation methods).

TDecoder implements all required methods of TGettableStream with the exception of atEnd and it requires four accessors (including atEnd) and the method effectiveNext. This last method is where all of the work happens. It reads from its input stream and returns a new element. The method next in TDecoder calls effectiveNext and catches StreamAtEndErrors for setting the atEnd variable.

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

6 Backward-compatible collection-based streams

Traits have been proposed as a way to ease reuse. To verify that claim, we wanted to test if our traits could be used to implement a backward-compatible version of the original stream library as present in Squeak and the original Smalltalk design [10].

The original Squeak implementation of the library is represented in Figure 1. In this implementation different classes were available to read (ReadStream), write (WriteStream) or read/write (ReadWriteStream) elements in collections. We mimic this implementation by creating the classes ReadableCollectionStream, WriteableCollectionStream and ReadWriteCollectionStream. These classes selectively reuse core traits.

The integration between this backward-compatible implementation and the core is shown in Figure 10 while Figure 11 presents a more detailed view.

In the rest of the section, we detail the different entities of this implementation.

AbstractCollectionStream. The class AbstractCollectionStream is the common superclass to iterate over collections. It declares the two instance variables collection and position and their accessors.

WriteableCollectionStream. This class subclasses AbstractCollectionStream to benefit from the variable and accessor declarations and uses TPuttablePositionStream. It implements the method nextPut: which is indirectly required by TPuttablePositionableStream. The class also reimplements nextPutAll: for efficiency reasons. The method size returns the value of the instance variable streamSize.

A careful reader may have noticed there is no reason for the implementation of these methods to be different from the ones in CollectionStream (see Figure 10). In fact, the methods are duplicated between the new library and the backward-compatible one. We chose this duplication over the introduction of another trait that would merely group these methods. We tried to use traits to factor common behavior representing an abstraction and not just a couple of shared methods. Moreover, it's not worth complicating the new design only to keep backward compatibility. Another, equivalent, duplication occurs for the methods used to read data from streams.

TReadableCollectionStream. This trait factors out methods to read data from collection-based streams. It is used by both ReadableCollectionStream and ReadWriteCollectionStream. We implemented that trait to avoid duplication of code between these two classes and because it represents a coherent abstraction.

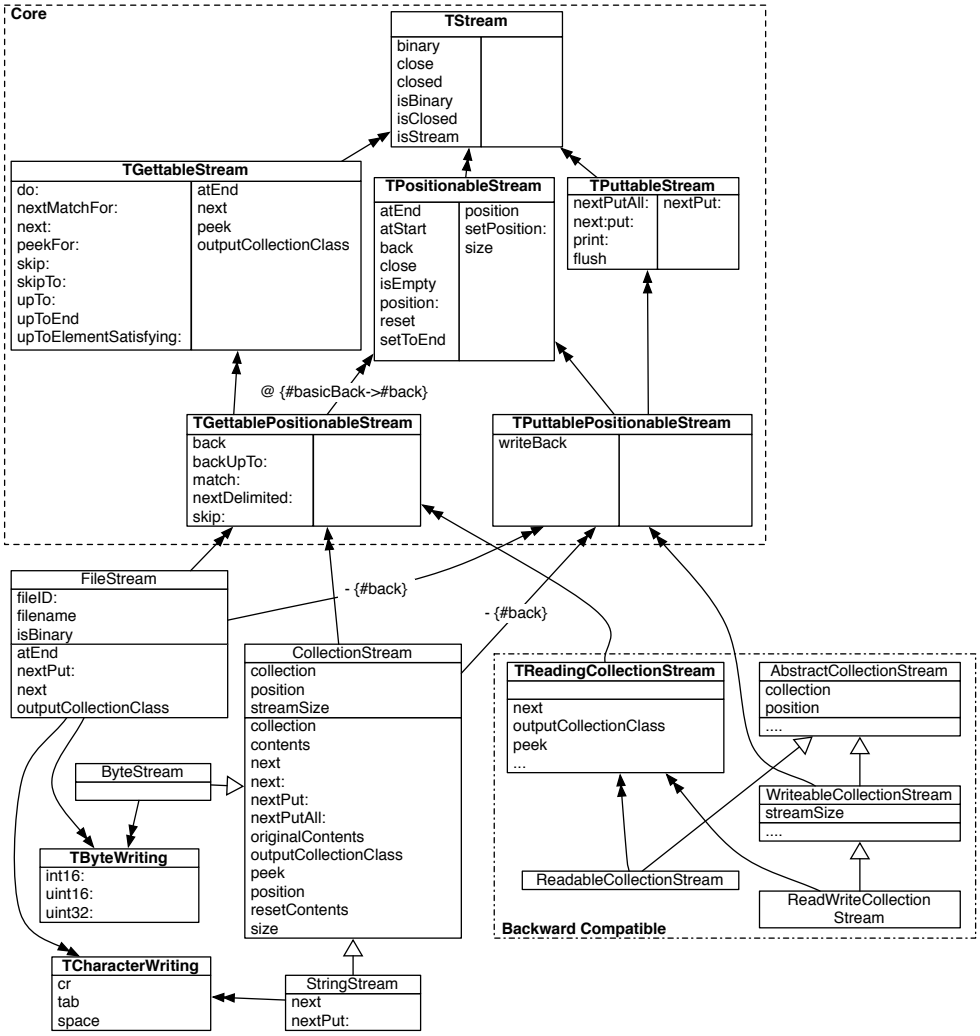

Fig. 10. Nile's core with three libraries: CollectionStream, FileStream and the backward-compatible collection-based streams.

ReadableCollectionStream. This class uses the traits TReadableCollectionStream and inherits from AbstractCollectionStream. The only method it implements is the method size which returns the size of the associated collection.

ReadWriteCollectionStream. This class allows for both reading and writing in collections. It inherits its instance variables, accessors and writing behavior from WriteableCollectionStream and its reading behavior from TReadableCollectionStream.

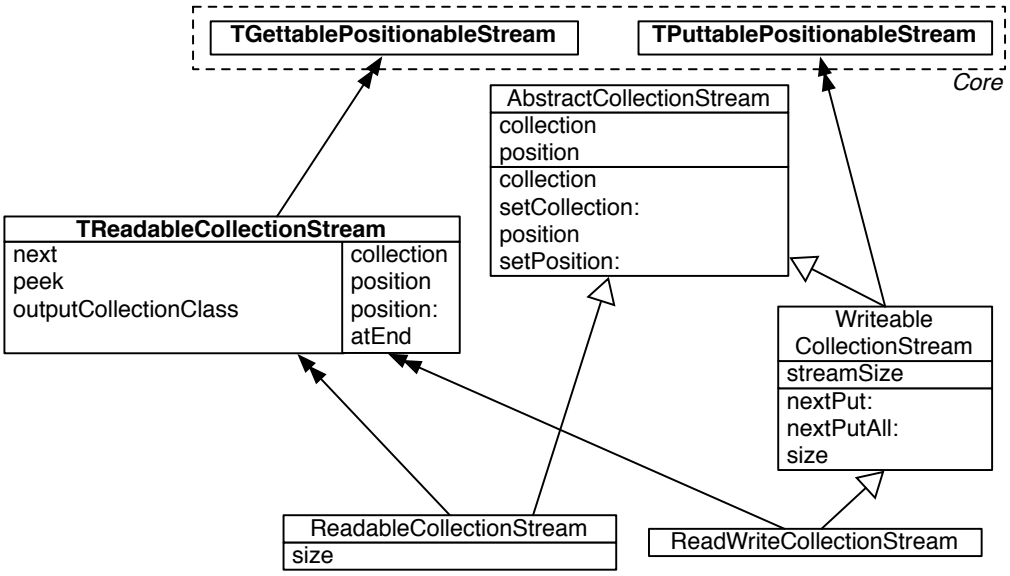

Fig. 11. The collection-based stream library.

7 Nile Analysis: Revisiting the Questions

Section 1 raised a number of questions regarding the traits model. This section revisits these questions one by one and provides answers based on our experience with developing Nile.

7.1 Trait granularity

When developing with traits, it is very tempting to develop traits that are too fine-grained. Similarly to the idea that inheritance is not only a reuse mechanism but can have a stronger semantics when used with subtyping in mind, we favored traits to represent coherent abstractions over a mere reuse mechanism. We avoided to create traits just to group and reuse a couple of methods; instead we developed our traits to represent abstractions. Our focus was to build a set of core abstractions that can be recombined in the context of collection and file-based streams.

While developing Nile, we first tried to mimic ANSI Smalltalk standard and Squeak by developing one class for each access mode on collections: reading, writing and reading-writing. We even went further, developing twelve entities (traits, concrete and abstract classes) only to manage file streams [8,13]. We had six concrete classes which represented the Cartesian product between binary and text files on one side and the three access modes on the other sides. These classes were linked with three abstract classes and three traits. This

implementation lets the users choose which kind of file streams they want and only get the necessary methods. On the downside we realized that this was far too complex and reimplemented everything in just one class. This solution is much simpler to use and to maintain. However, the user now has access to methods she doesn't really want (*i.e.,* she can send the method nextPut: even if her stream is read-only). In general, developing is making a lot of trade-offs. Traits introduce another dimension when designing a system and as such open the trade-off space.

From our experience developing Nile we conclude that traits support well the choice and the granularity of the abstractions that a programmer can define and compose.

7.2 Deciding between traits and classes

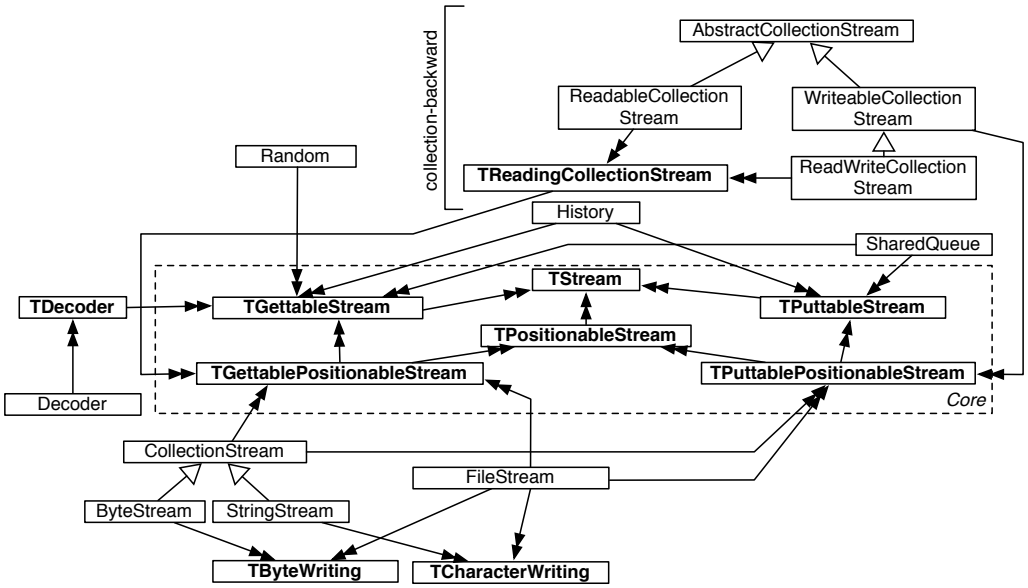

Fig. 12. An overview of some of the Nile's libraries

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 inherited but not used or inherited behavior should be canceled. There are clear design signs showing that the solution is suboptimal when the programmer could use instead traits, since traits can solve such problems.

Now defining one or more traits instead of a class is not a simple decision to take. 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 addition as we mentioned earlier traits can be used as a reuse mechanism but we believe that they are better when representing (composable) abstractions.

7.3 Reusability of traits at work

Figure 12 offers an overview of the core libraries of Nile as well as some other ones.

library name	superclass and trait used	met.	description
TDecoder	TGettableStream	3	allows stream chaining (see Section 5)
Decoder	TDecoder	1	allows stream chaining (see Section 5)
Random	TGettableStream	13	generate random numbers.
LinkedListStream	TGettableStream TPuttableStream	10	stream over linked elements.
History	TGettableStream TPuttableStream	18	manage an history of elements and allows going back(see Section 5).
SharedQueue	TGettableStream TPuttableStream	11	concurrent access on a queue.
CompositionStream	Decoder	2	multiplexer for input streams.
Tee	Decoder	3	fork the input stream (like the Unix *tee* command).
Buffer	Decoder	3	add a buffer to any kind of input stream.
NumberReader	Decoder	2	read numbers from a character based input stream.
SelectStream	Decoder	2	select elements from an input stream.
PipeEntry	TGettableStream TPuttableStream	10	allow data to be manually put into a pipe.

Table 4
Nile libraries

Nile comes with reusable traits that 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 4 presents the current libraries we implemented in Nile using traits as well as the number of implemented methods to get the desired behavior (without the accessors).

Table 5 presents how much the core traits are reused. It presents for each trait the number of libraries, 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.

Table 6 presents some metrics which compare the same functionalities in the

Trait	client classes	required m.	provided m.	$\frac{provided}{required}$
TGettableStream	17	5	18	360%
TPositionableStream	10	4	14	350%
TPuttableStream	8	2	12	600%
TGettablePositionableStream	5	7	30	428%
TPuttablePositionableStream	5	5	20	400%
TCharacterReading	4	5	2	40%
TCharacterWriting	4	1	8	800%
TByteReading	4	3	14	466%
TByteWriting	4	3	13	433%
TDecoder	7	7	21	300%

Table 5
Nile-trait reusability.

	Squeak	Nile	$\frac{Squeak-Nile}{Squeak}$
Number of Classes And Traits	5	6	-20%
Number of Methods	55	33	40%
Number of Bytes	1769	1064	39%
Number of Cancelled Methods	2	0	100%
Number of Reimplemented Methods	14	3	78%
Number of Methods Implemented Too High	10	0	100%

Table 6
Some metrics for the collection-based streams

Squeak implementation and in Nile for the collection-based streams. The first metric indicates that Nile has only one more entity (class/trait) than the Squeak implementation. The next two (number of methods and number of bytes) are also interesting and show that the amount of code is smaller in Nile than in Squeak. Nile has 40% less methods and 39% less bytecodes than the corresponding Squeak collection-based stream library. This means we avoided reimplementation of a lot of methods by putting them in the right traits. Finally, we can deduce from the last metrics that the design of Nile is better: there is neither cancelled methods nor methods implemented too high and there are only three methods reimplemented for speed reasons 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 trait. 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 TGettablePositionableStream doesn't have any work to get the implementation of atEnd. Such a situation is rare and based on the decomposition of traits using a compatible behavior and vocabulary.

On the other hand, it is sometimes better or even necessary to override a method coming from a trait. Similar to specializing an inherited method, the new implementation has more knowledge than the overridden one and thus can do a better job. For example, the method skip: in TGettablePositionableStream overrides the one in TGettableStream. The new method is more efficient because the stream is positioned directly, needing only a small bound computation:

```
TGettablePositionableStream>>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 the amount given is negative, which is not possible for the general implementation of TGettableStream.

7.4 Performance optimization

One of the key challenges of Nile's design in terms of performance is to be able to iterate over any kind of collection while at the same time being as efficient as the Squeak implementation that only accepts (and is optimised for) Arrays and Strings. We present our solution to this challenge.

Contrary to the Squeak class WriteStream which can only iterate over Arrays and Strings, Nile's CollectionStream is able to iterate over any kind of SequenceableCollection. In Squeak the method nextPutAll: in WriteStream directly manipulates its internal collection using a call to the primitive replaceFrom:to:with:startingAt: implemented in String and Array [3]. Nile has more work to do.

The idea is to propose a dedicated set of classes working specifically on Array and String. We first reimplemented the method nextPutAll: in CollectionStream 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

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

time was lost into calling methods. We have then implemented an optimized version (*i.e.,* using the primitive method replaceFrom:to:with:startingAt:) of nextPutAll: directly into the class StringStream in which we are sure that the underlying collection is a String (the problem and the solution are the same to handle Arrays).

Table 7 shows that we managed to make Nile even more efficient than Squeak. We tested the methods next, next:, nextPut: and nextPutAll: on string-based streams. The benchmarks have been executed on a GNU/Linux 2.6.22 kernel with an Intel Pentium M 2.13Ghz processor. This results are important because they show that traits allows better design while not preventing fine optimizations where necessary.

	Squeak	Nile	$\frac{Nile-Squeak}{Nile}$
next	72.2	72.9	1%
next:	144.4	202.8	29%
nextPut	64.2	105.5	39%
nextPutAll	177.8	185.4	4%

Table 7
Benchmarks comparing Squeak and Nile for string-based streams. The first two results are in number of executions per second. The third one shows the difference.

7.5 Traits Interface pollution problem

In this section we present some problems of traits 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 variables. This is a problem because (1) the accessors are then part of the interface of the classes and (2) this adds a burden to the class developers. (1) could be solved if Smalltalk would have method access control, such as a "protected" modifier, because the accessor methods could then be hidden instead of being public. A solution to solve both problems would be to use stateful traits [3], where traits can contain private state. For example, if we had used stateful traits the methods setPosition: in TPositionableStream would not have been required.

Lazily initialized variables. There are basically three ways of initializing an instance variable with an initial first value: initializing the variable lazily 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 the initial value of false and then returns the value:

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

Imagine a situation where a trait needs a variable and wants to give an initial value for that variable. Since traits cannot contain state, the variable cannot be declared directly in the trait. Accessors for that variable are made required methods instead. The question then is where we need to lazy initialize the variable. Two solutions are possible: you can either force users of the trait to initialize the variable or you can initialize the variable in the trait and use another method for accessing the variable. Here is an example of the latter 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. Note that method access control such as "protected" would solve that problem. 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 position: in TPositionableStream 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. Two methods are then part of the interface where only one was really necessary.

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 code sends initialize to the superclass and then sets a default value for the color instance variable. This can be done in a trait as well, provided that the developer uses an accessor instead of a direct reference to the variable. However, a class or a trait can use multiple traits, each defining their own initialize method. In this case, there will be conflicts between the initialize methods obtained by these traits. This is not a problem by itself: when composing the initialize methods can be removed and the different initialize methods can be aliased so that they can still be called in the composition. While not problematic, this brings a lot of pollution in the class interface (all the aliased initialization methods) and requires work.

Another solution would be to use a specific name for each initialize method. For example, if 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. This still requires too much work from the developer and does not solve the class interface pollution problem.

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

```
CollectionStream>>on: aCollection
  ^ self basicNew
      initialize;
      setCollection: aCollection;
      streamSize: 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.

7.6 Challenges for supporting development with traits

While traits are a powerful reuse mechanism, they introduce another axe of freedom in the already difficult task of developing. The following paragraphs detail a number of practical problems a developer faces when he wants to apply traits in practice.

Orthogonal relation representations. Representations of single inheritance class hierarchies have been done for a long time now and are working well in IDEs. Traits offer a new orthogonal relation between classes and traits. This makes IDE views harder to represent and understand. The more entities, the more difficult it is to represent the relations in a clean way.

Lack of code browsing support. While alternative browsers have been prototyped [14], there is currently no good tool to browse the orthogonal representations of class and trait hierarchies in a unified view. This makes things difficult when the developer wants to add features or refactor code. When the developer browses a class, she has to know where methods are really implemented (in this class, in a super class or in a trait) and where to put new methods.

Understanding two orthogonal hierarchies. In a class-only hierarchy, it is sometimes already difficult to know in which class to put a new feature. With traits, it is even more complicated and only a global understanding of the classes and traits used will allow adding a new feature in the correct place. Traits granularity and composition make certain changes possible as shown in the design of Nile: we can change the traits and their relationships easily and still get the same behavior for the streams.

8 Related work

8.1 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.

Figure 13 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 backward-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, for example, with files. This problem comes from the

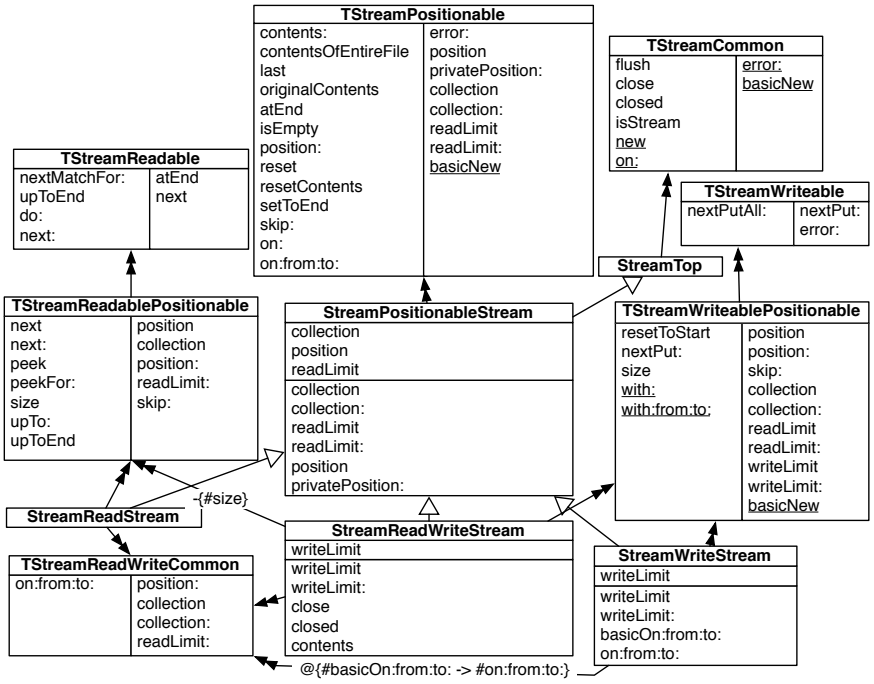

Fig. 13. Schärli's refactored stream hierarchy.

implementation in the Squeak hierarchy, where the class PositionableStream introduces both positioning and collections in the same time.

- The methods peek and upToEnd in TStreamReadablePositionable depends on the existence of methods collection and position, which should not be the case.
- The implementation only takes care of array-based and string-based streams.

8.2 Non-trait related work

We already compared our approach with the research on 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 [15] 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.

Snelting and Tip analyze a class hierarchy making the relationship between class members and variables explicit [16]. 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 [17]. 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 [18]. 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 [19] 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 [20]. 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 [21].

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. 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.

9 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 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 40% less code than the corresponding Squeak implementation. Core traits are reused by numerous libraries. 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 enhancements.

This experience shows that well-defined traits can naturally fit into lots of different libraries which can benefit from methods offered by the trait for a relatively low cost. In the future we plan to further develop the new library core and its extensions since it supports both backward-compatible extensions and more compact ones.

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). We thank Alexandre Bergel and Mathieu Suen for their reviews.

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 (July 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**(2) (March 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 (August 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 (October 2007) 171–190

[5] Black, A.P., Schärli, N., Ducasse, S.: 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. (October 2003) 47–64

[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 (November 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] Cassou, D., Ducasse, S., Wuyts, R.: Redesigning with traits: the nile stream trait-based library. In: Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007), ACM Digital Library (2007) 50–75

[9] 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 (November 1997) 318–326

[10] Goldberg, A., Robson, D.: Smalltalk-80: The Language. Addison Wesley (1989)

[11] Black, A., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., Denker, M.: Squeak by Example. Square Bracket Associates (2007) http://SqueakByExample.org/.

[12] Alpert, S.R., Brown, K., Woolf, B.: The Design Patterns Smalltalk Companion. Addison Wesley (1998)

[13] Cassou, D.: Remodularisation à base de traits. Master's thesis, Université Bordeaux I (2007)

[14] Black, A.P., Schärli, N.: Traits: Tools and methodology. In: Proceedings ICSE 2004. (May 2004) 676–686

[15] Godin, R., Mili, H., Mineau, G.W., Missaoui, R., Arfi, A., Chau, T.T.: Design of Class Hierarchies based on Concept (Galois) Lattices. Theory and Application of Object Systems **4**(2) (1998) 117–134

[16] Snelting, G., Tip, F.: Reengineering Class Hierarchies using Concept Analysis. In: ACM Trans. Programming Languages and Systems. (1998)

[17] 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

[18] 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 (October 1992) 1–15

[19] 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

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

[21] 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

A Appendix: Traits in a Nushell

To ease the understanding of this paper we added this section which presents traits in a nutshell. This part is taken from [4] 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 [2]. 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 A.1 shows a class SyncStream that uses two traits, TSyncReadWrite and TStream. The trait TSyncReadWrite provides the methods syncRead, syncWrite

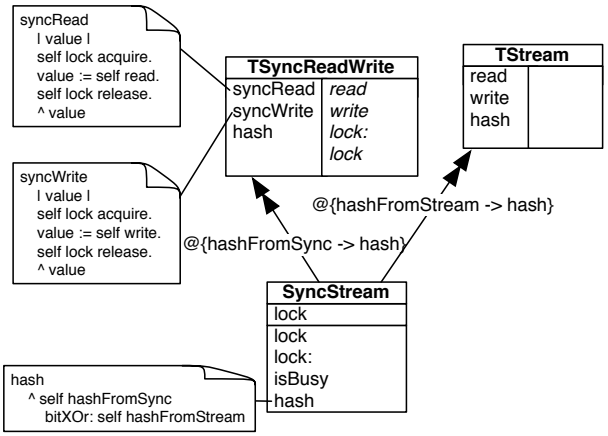

Fig. A.1. The class SyncStream is composed of the two traits TSyncReadWrite and TStream.

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 [2].

In Figure A.1, the class SyncStream is composed from TSyncReadWrite and

TStream. The trait composition associated to SyncStream is:

```
  TSyncReadWrite alias hashFromSync → hash
+ TStream alias hashFromStream → 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 (▷), exclusion (−) and aliasing (alias →) [2].

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 (▷) 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 → 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* → *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.

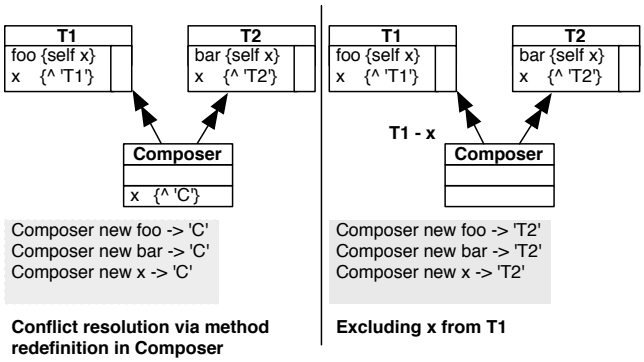

Fig. A.2. Trait conflict resolution strategies: either via method redefinition or via method exclusion.