Université Bordeaux I
Laboratoire Listic, Université de Savoie
Lieu Annecy, France
Maître de stage Stéphane Ducasse
Tuteur Marc Zeitoun

Mémoire de recherche

Remodularisation à base de traits

Damien Cassou

Février à Juin 2007

Keywords. Object-oriented programming, Refactoring, Restructuration, Traits, Inheritance.

Résumé

Récemment, les traits ont proposé un solution compatible avec l'héritage simple dans lequel l'entité qui compose a le contrôle sur la composition. Les traits sont des éléments à granularité fine qui permettent la composition de classes, mais qui évite la plupart des problèmes posés par l'héritage multiple et les approches basées sur les mixins.

Pour évaluer l'efficacité des traits, des bibliothèques ont été refactorisées, montrant une réutilisation importante du code. Cependant, bien que ces travaux soient intéressants, ils ne permettent pas de rencontrer tous les problèmes d'utilisation des traits ; ceci parce que les bibliothèques d'origines étaient réalisées et pensées avec les contraintes de l'héritage simple. Nous souhaitons évaluer l'expressivité des traits lors de la réalisation d'un projet complet, en se servant des traits comme unité de réutilisation de comportement. Ce document présente le design d'une nouvelle bibliothèque de streams appelée Nile. Nous présentons les traits que nous avons définis et leur réutilisabilité ainsi que les problèmes auxquels nous avons fait face.

Table des matières

1 Introduction **5**

2 Les traits, une unité de comportement **8**

3 Remodularisation à base de traits **12**
3.1 Introduction . . . 13
3.2 Working hypotheses and analyses . . . 14
3.2.1 Working hypotheses . . . 14
3.2.2 Analysis of the Squeak stream hierarchy . . . 15
3.2.3 The ANSI specification . . . 17
3.3 Nile overview and core . . . 19
3.4 Collection-based streams . . . 20
3.4.1 The traits . . . 20
3.4.2 Trait factories . . . 22
3.4.3 Classes . . . 22
3.5 File-based streams . . . 23
3.6 Other libraries . . . 24
3.6.1 Writing characters . . . 24
3.6.2 Decoders . . . 25
3.7 Discussions . . . 27
3.7.1 Comparison with previous work . . . 27
3.7.2 Nile Analysis . . . 28
3.7.3 Performance optimization . . . 31
3.8 Problems with traits . . . 33
3.8.1 A Squeak 3.9 trait implementation bug . . . 33
3.8.2 Interface pollution . . . 33
3.8.3 Methods silently ignored . . . 36
3.9 Related work . . . 36
3.10 Conclusion . . . 38

4 Résultats **39**

5 Conclusion **41**

A La syntaxe Smalltalk **44**

1 Introduction

Ce mémoire de recherche est destiné à présenter le travail que j'ai réalisé durant trois mois au Listic, laboratoire de l'université de Savoie.

Mon maître de stage et moi avons décidé, avec l'accord des tuteurs, de consacrer notre temps de rédaction à un article pour une conférence internationale plutôt qu'à un mémoire de recherche. L'essentiel de ce mémoire de recherche se trouve donc au chapitre 3 qui contient l'article original.

Contexte. L'héritage multiple a suscité beaucoup de travaux et d'efforts de recherche[10, 8, 16]. Récemment, les traits [12, 5] ont proposé une solution dans laquelle l'entité qui compose a le contrôle de la composition. Les traits sont des entités à granularité fine qui permettent de factoriser du comportement pour les classes. Nous les présentons à la section 2.

Problématique. Des recherches précédentes ont permis de valider l'utilité des traits en refactorisant des bibliothèques existantes de collections et de streams. Bien qu'utiles, ces refactorings ne permettent pas d'évaluer les traits dans le contexte d'un projet qui serait basé dessus depuis sa création.

Buts. Le but de ce travail est de vérifier expérimentalement la faisabilité de la réalisation d'un projet basé dès l'origine sur les traits. Plus spécifiquement, nous souhaitons répondre aux questions suivantes :

- Quelle est la granularité idéale des traits qui favorise la réutilisation ?
- Quelle proportion de code peut être réutilisée grâce aux traits ?
- Est-ce que les traits peuvent servir de blocs composables ?
- Comment choisir entre l'utilisation de trait et l'héritage de classe ?
- Jusqu'à quel point pouvons nous corriger les problèmes que nous identifierons dans la bibliothèque de streams existante ?
- Quelles sont les limites et les contraintes que nous rencontrerons lors de l'utilisation des traits ?

Solution. Nous avons décidé de recréer une bibliothèque de stream, appelée Nile. Le choix des streams s'est fait pour plusieurs raisons : (1) ils posent des problèmes lors de leur implémentation avec héritage simple ; (2) d'autres travaux impliquant les streams et les traits ont déjà été réalisés ce qui nous permettra une comparaison ; (3) les streams sont une bibliothèque importante des langages de programmation ; (4) des contraintes sont imposées par les spécifications ANSI Smalltalk et par les implémentations existantes.

Contributions. Les contributions de ce travail sont :

1. l'identification des problèmes des bibliothèques de streams existantes ;
2. l'analyse de la définition des streams par le standard Smalltalk ANSI ;
3. le design de Nile, une nouvelle bibliothèque de streams composée d'éléments réutilisables ;
4. l'évaluation des traits en tant qu'éléments réutilisables pour définir des bibliothèques et la vérification de l'obtention d'un design plus clair et plus modulaire ;
5. une identification des problèmes qui surviennent lors de l'utilisation des traits.

Réalisation. J'ai réalisé l'ensemble de la bibliothèque Nile seul. De nombreuses discussions avec mes collègues ou la communauté Smalltalk m'ont cependant aidé dans mes choix.

Publication. Le travail décrit ici a mené à la soumission d'un article, dont je suis le premier auteur, dans la conférence International Smalltalk Conference. Il est publié dans la série LNCS (Lecture Notes in Computer Science).

Les streams. Une bibliothèque de streams permet la lecture et l'écriture séquentielle d'objets dans des collections d'éléments, des fichiers, des flux réseaux...

Smalltalk. Le Smalltalk est un langage de programmation orienté objet de très haut niveau. Sa syntaxe est simple et son design très cohérent. J'ai écris quelques pages sur la syntaxe de Smalltalk, vous les trouverez en annexe A. Le groupe "Langage and Software Evolution" travaille presque exclusivement dans ce langage.

Cadre. Le Listic est un laboratoire universitaire rattaché à l'Université de Savoie. Il travaille particulièrement dans le domaine de la fusion d'informations et vise à développer des outils méthodologiques et logiciels pour la maîtrise de la chaîne d'informations.

Équipe. L'équipe "Logiciels et Systèmes", à laquelle je suis rattaché, travaille sur la conception, la réalisation et le déploiement des systèmes de fusion ainsi que sur l'évolutivité des logiciels. Le groupe "Langage and Software Evolution" travaille sur les langages et l'évolution des logiciels : architecture, conception et implantation de langages de programmation, réingénierie et évolution des logiciels... Le groupe est mené par le professeur Stéphane Ducasse, mon maître de stage.

Plan Ce mémoire de recherche est basé sur l'article "Redesigning with Traits : the Nile Stream trait-based Library" que vous trouverez en section 3. Avant cela, la section 2 présentera brièvement le modèle des traits. Enfin, la section 4 résumera les résultats que nous avons obtenu par notre travail.

2 Les traits, une unité de comportement

Les traits [5] sont à la base du travail que j'ai réalisé pendant ce stage. Une bonne compréhension de leurs mécanismes est donc nécessaire à la compréhension du reste de ce mémoire de recherche.

Des groupes réutilisables de méthodes. Les traits sont des ensembles de méthodes qui servent de blocs de réutilisations pour les classes. Une classe déclare utiliser un ensemble de trait et ces traits lui apportent du comportement supplémentaire. En plus de définir du comportement, les traits nécessitent des méthodes, que l'on appellera *required methods* ou méthodes requises. Ces méthodes sont nécessaires à l'utilisation du trait par une classe. Les traits ne définissent pas d'état, à la place, ils peuvent nécessiter des accesseurs.

ddd

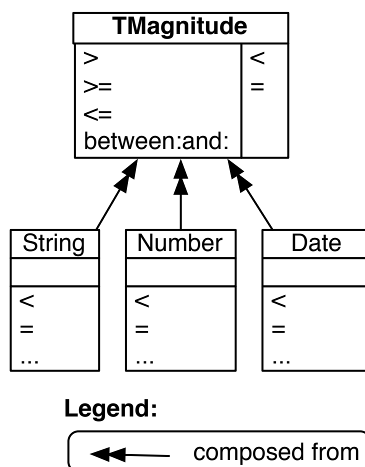

Fig. 2.1: Le trait TMagnitude et trois clients possibles.

La figure 2.1 présente un premier exemple d'utilisation de trait. Une extension de UML est utilisée pour représenter les traits : à gauche se trouvent les méthodes offertes et à droite les méthodes requises. Le trait TMagnitude offre quatre méthodes ($<=$, $>$, $>=$ et between:and:) à toutes les classes qui fournissent $<$ et $=$ (qui sont les deux méthodes requises du trait). Ici, les classes String, Number et

Date déclarent l'utilisation du trait TMagnitude et définissent les deux méthodes requises par le trait : $<$ et $=$.

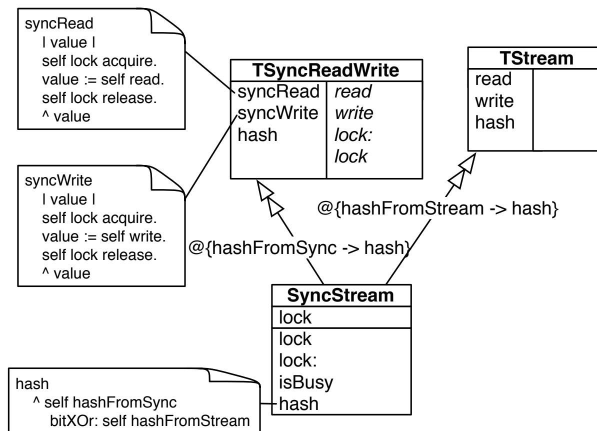

FIG. 2.2: La classe SyncStream est composée des deux traits TSyncReadWrite et TStream.

La figure 2.2 montre une classe SyncStream qui utilise deux traits, TSyncReadWrite et TStream. Le trait TSyncReadWrite offre les méthodes syncRead, syncWrite et hash. Il requière les méthodes read et write, et deux accesseurs lock et lock: ; il n'y a pas de différence sémantique entre un accesseur et une méthode quelconque.

Composition explicite. Une classe possède une référence à sa super classe, utilise un ensemble de traits, possède un état au moyen de variables d'instances et définit des méthodes pour brancher les traits ensembles et résoudre les possibles conflits.

La composition de traits respecte les trois règles suivantes :

- Les méthodes définies dans une classe sont prioritaires sur les méthodes venant des traits. Cela permet aux méthodes de la classe de redéfinir les méthodes des traits.
- Propriété de *mise à plat*. Dans chaque classe qui utilise des traits, les traits peuvent être fusionnés dans la classe pour donner une classe au comportement strictement équivalent mais sans les traits.
- L'ordre de composition des traits ne rentre pas en compte. Tous les traits ont la même priorité, et donc les conflits doivent être résolus explicitement.

Résolution de conflit. Lors de la composition des traits, des conflits de méthodes peuvent survenir. Un conflit arrive si deux ou plusieurs traits composés définissent des méthodes avec le même nom mais qui ne prennent pas leur source du même trait. Il y a deux stratégies pour résoudre un conflit : en implémentant une méthode au niveau de la classe qui redéfinie les méthodes en conflit ou en excluant la méthode en conflit de tous les traits sauf un. Les traits permettent la définition

d'alias qui donnent une deuxième nom à une méthode implémentée par un trait. Le nouveau nom est utilisé pour avoir accès à l'ancienne implémentation de la méthode qui a été redéfinie.

Dans la figure 2.2, la classe SyncStream est composée de TSyncReadWrite et TStream. La composition associée à SyncStream est :

TSyncReadWrite alias *hashFromSync* → *hash*
+ TStream alias *hashFromStream* → *hash*

La classe SyncStream est composée de (1) le trait TSyncReadWrite pour lequel la méthode hash possède un second nom hashFromSync et (2) le trait TStream pour qui la méthode hash possède un deuxième nom hashFromStream. La classe TStream redéfinie la méthode hash localement. La nouvelle définition utilise les deux alias précédemment définis.

```
TStream>>hash
  ^ (self hashFromSync) bitXOr: (self hashFromStream)
```

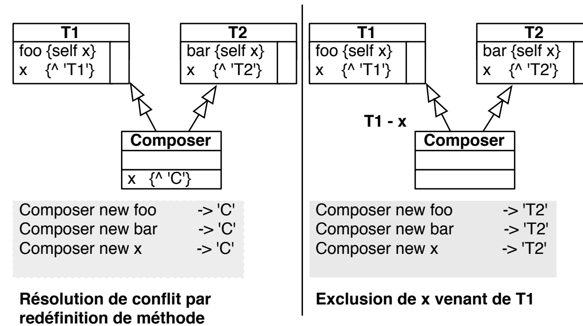

Fig. 2.3: Les stratégies de résolution de conflit : par redéfinition ou par exclusion de méthode.

La figure 2.3 présente les deux stratégies de résolution de conflit. À gauche, la classe Composer redéfinie la méthode x pour qu'elle retourne la chaîne de caractère 'C'. Après cette redéfinition, toutes les méthodes x, foo et bar retournent 'C'. Dans le cas de droite, la classe Composer n'importe pas la méthode x du trait T1. Il n'y a donc plus de conflits, et les méthodes x, foo et bar retournent toutes la chaîne de caractère 'T2'.

Opérateurs de composition. La sémantiques de la composition des traits est basée sur quatre opérateurs : sum (+), override (▷), exclusion (−) et aliasing (alias →).

Le trait TSyncReadWrite + TStream contient toutes les méthodes de TSyncReadWrite et TStream qui ne sont pas en conflit. S'il y a un conflit, c'est à dire si TSyncReadWrite et TStream définissent une méthode avec le même nom, alors

dans TSyncReadWrite + TStream, le nom de la méthode est lié à une méthode en conflit. L'opérateur + est associatif et commutatif.

L'opérateur de *override* (▷) construit un nouveau trait qui étend une composition de traits existant avec des définitions locales. Par exemple, SyncStream redéfinie la méthode hash obtenue à partir de la composition.

Un trait peut exclure des méthodes d'un trait existant en utilisant l'opérateur d'exclusion −. Par exemple, le trait TStream − {read, write} possède une seule méthode hash. L'exclusion est utilisée pour éviter les conflits, ou si l'utilisateur a besoin de réutiliser un trait trop gros pour ses besoins.

L'opérateur d'aliasing alias → créé un nouveau trait qui propose un second nom pour une méthode existante. Par exemple, si le trait TStream est un trait qui définit read, write et hash, alors le trait TStream alias *hashFromStream* → *hash* définit les méthodes read, write, hash et hashFromStream. La méthode additionnelle hashFromStream a le même corps que la méthode hash. Les alias rendent disponibles les méthodes conflictuelles sous un nouveau nom, permettent de satisfaire les besoins d'un autre trait ou pour éviter la redéfinition.

3 Remodularisation à base de traits

Ce chapitre est composé de l'article "Redesigning with Traits : the Nile Stream trait-based Library" dont je suis le premier auteur et le rédacteur principal. Deux autres auteurs ont participé à la rédaction de cet article : mon maître de stage tout d'abord et Roel Wuyts, professeur à l'Université Libre de Bruxelles. Cet article a été soumis à la International Smalltalk Conference.

Dans cet article, nous présenterons tout d'abord le contexte de ce projet : pourquoi est-il important dans le cadre de la recherche sur les traits. Ensuite, nous détaillerons l'implémentation de la bibliothèque de streams que j'ai développée et la comparerons avec l'existant. Enfin, nous aurons une discussion sur les avantages de Nile, notre bibliothèque, par rapport à d'autres approches et nous détaillerons aussi certains problèmes que nous avons rencontré avec l'utilisation des traits.

Redesigning with Traits: the Nile Stream trait-based Library

Abstract

Recently, traits have been proposed as a single inheritance backward compatible solution in which the composing entity has the control over the composition. Traits are fine-grained components 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. Adaptive Object Model, Meta-Data, Meta-Modeling, Business Application Development, Smalltalk

3.1 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 [12, 5]. Traits are fine-grained components 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 control to the composer: they are meant for single inheritance languages, automatically detect composition conflicts but let the composer resolve these conflicts explicitly, and at all times leave control to the composer. With this solution, traits claim to avoid many of the problems of multiple inheritance and mixin-based approaches that mainly favor linearization so that conflicts never arise explicitly and are solved implicitly by ordering.

Previous research evaluated the usefulness of traits by refactoring the Smalltalk collection and stream libraries, which showed up to 12% of gain in terms of code reuse [13]. Other research tried to semi-automatically identify traits in existing libraries [9]. While these are valuable results, they are all refactoring scenarios that investigated the applicability of traits using *existing* systems as input. Usability of traits when developing a *new* system has not been assessed.

The goal of this paper is to experimentally verify the original claims of simplicity 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: *what is the granularity of traits?*, *is the composition mechanism good enough to deal with common composition scenarios?*, *what do we gain from using traits?*, and *when is it better to define a trait versus a class?*

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 specifications 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 [13] and A. Lienhard [9] 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 specifications 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 elements, (2) the assessment that traits are good building elements 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 working hypotheses that drive this work. We highlight the existing Squeak Stream hierarchy limits and the ANSI Smalltalk standard protocols (Section 3.2). Section 3.3 presents an overview of Nile and the core of the library around its three most important traits. Section 3.4 and Section 3.5 detail the implementation of the collection-based and file-based stream libraries, respectively. Two other libraries will be presented in Section 3.6. Section 3.7 compares our approach with the one of N. Schärli. It analyses the reuse offered by traits as well as performance issues and optimization solutions. Finally, Section 3.8 presents the problems we identify due to the use of traits.

3.2 Working hypotheses and analyses

In this section, we present the hypotheses that motivated us to write a new stream library. Then we analyze the existing stream hierarchy of Squeak the open-source Smalltalk [7]. We highlight the key problems and present the ANSI standard.

3.2.1 Working hypotheses

Implementing a stream library from scratch is an important experience to test the expressiveness of traits. Indeed, contrary to previous validations of traits that consist in refactoring existing class hierarchies, we start from scratch and free ourselves from the constraints imposed by an existing system. By doing so we may face problems that may have been hidden in previous experiences and also face a large scheme of composition problems.

Here is a list of the questions we want to answer with this experience:

- Trait granularity. We want to assess the granularity of traits that maximize the reusability.
- Trait reusability. We want to understand how much code can be reused.
- Can we define traits as composable building blocks?
- Can we identify guideline to assess when trait composition should be preferred over inheritance?

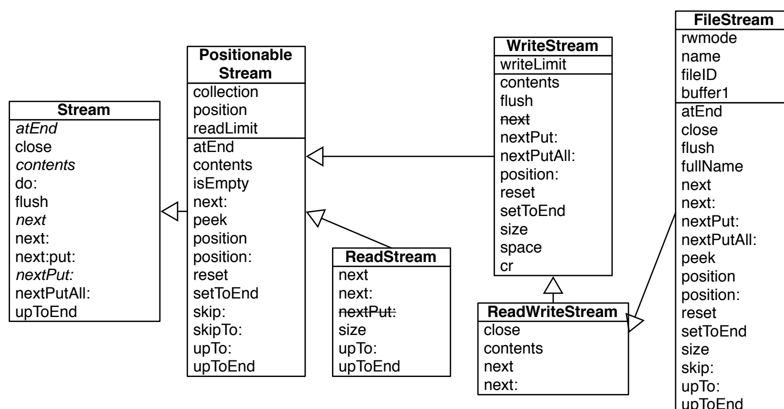

Figure 3.1: The Squeak core Stream hierarchy. Only the most important methods are shown.

- Until what extent can we fix the problems identified in the current stream hierarchy?
- What trait limits and problems do we encounter?

3.2.2 Analysis of the Squeak stream hierarchy

Squeak [7], like all Smalltalk environments, has its own implementation of the stream hierarchy. Figure 3.1 presents the core of this implementation, which is solely based on single inheritance and does not use traits. It can be compared to stream implementations of other Smalltalk dialects such as VisualWorks.

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.[1] 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 3.1). 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 then typically 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. In Figure 3.1, 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 is clearly a code smell that points to inheritance hierarchy anomalies.

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

3.2.3 The ANSI specification

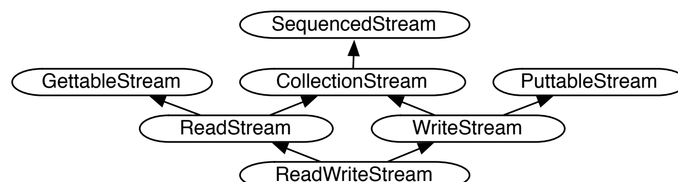

Figure 3.2: The ANSI Standard stream protocol hierarchy.

Figure 3.2 shows that even if Smalltalk is a single inheritance language, the ANSI standard defines the different protocols using multiple inheritance. In the standard, streams are based on the notion of sequence values. Each stream has got past and future sequence values. The ANSI Smalltalk defines a decomposition of stream behavior around three main protocols: GettableStream, SequencedStream and PuttableStream. Table 3.1 and Table 3.2 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.

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 3.1: The SequencedStream and PuttableStream protocols defined by ANSI Smalltalk.

The ANSI 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 definition is equivalent to an equality test between the peeked up object and the parameter:

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

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 3.2: GettableStream protocol defined by ANSI Smalltalk.

Most Smalltalk implementations (amongst which 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.

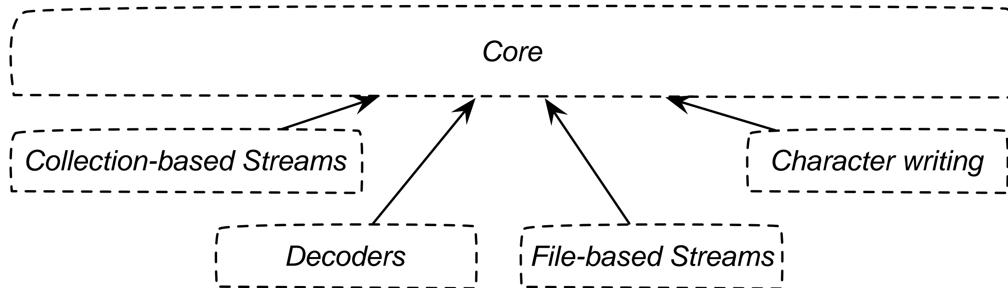

Figure 3.3: Overview of Nile: the core and its different libraries.

TGettableStream	
do: nextMatchFor: next: peekFor: skip: skipTo: upTo: upToEnd upToElementSatisfying:	atEnd next peek outputCollectionClass

TPositionableStream	
atEnd atStart close isEmpty position: reset setToEnd	position setPosition: size

TPuttableStream	
nextPutAll: next:put: print: flush	nextPut:

Figure 3.4: The Nile core traits.

3.3 Nile overview and core

Nile is designed around a core of traits offering base functionality reflecting the ANSI 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 3.3 presents an overview of Nile.

We designed Nile around three independent traits, reflecting the ANSI standard: TPositionableStream, TGettableStream and TPuttableStream. They are shown in Figure 3.4.

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:.

TPositionableStream. The trait TPositionableStream allows for the creation of streams that are positioned in absolute manner. It corresponds to the ANSI SequencedProtocol; 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. Due to the use of stateless traits [1], 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.

3.4 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 VisualWorks, our implementation actually works with any sequenceable collection, not just Arrays and Strings.

3.4.1 The traits

The traits TCollectionStream, TReadableCollectionStream and TWriteableCollectionStream implement the collection-based functionalities (as shown in Figure 3.5 — 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.

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 traits is inspired by the ANSI 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.

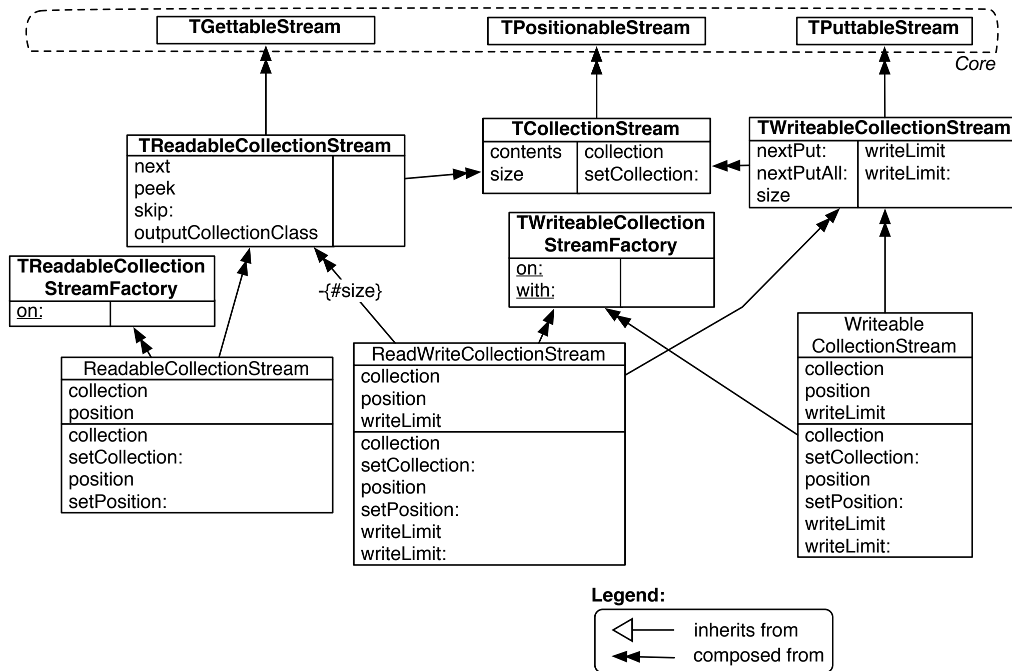

Figure 3.5: The collection-based stream library. We use a UML-based notation to represent traits: methods on the right are *required* and methods on the left are *provided*.

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.

3.4.2 Trait factories

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

We chose a third solution: implementing the constructors 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 Standard does not define on: for writeable streams, we decided to implement it following the Squeak and VisualWorks implementations.

3.4.3 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, inherited 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 3.5 on the arrow going from ReadWriteCollectionStream to TReadableCollectionStream[2].

3.5 File-based streams

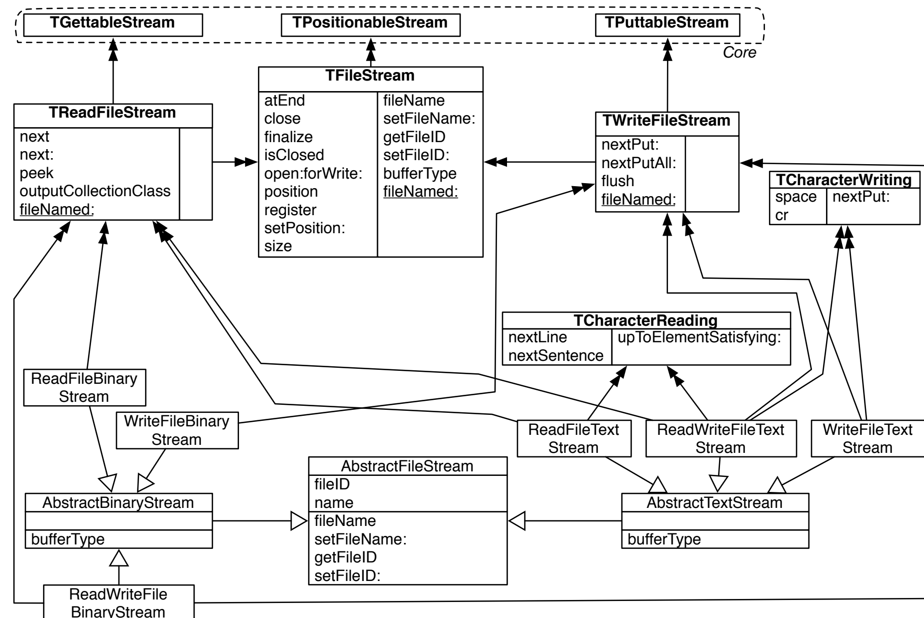

Figure 3.6: FileStream implementation.

Nile includes a file-based stream library, shown in Figure 3.6. 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 a constructor fileNamed. The private method bufferType is used to differentiate binary from text files.

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

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

At the very bottom of Figure 3.6, 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.

3.6 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 3.3.

3.6.1 Writing characters

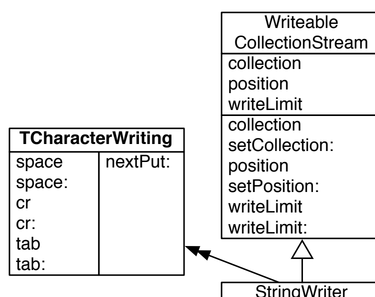

Figure 3.7: Writing characters to a string.

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 3.7 (a collection-based write stream which is writing characters) or WriteFileTextStream and ReadWriteFileTextStream in Figure 3.6.

3.6.2 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 an other 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
```

Figure 3.8 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.

The trait TDecoder. Figure 3.9 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 implements all methods of TGettableStream and it requires four accessors and the method effectiveNext. This method effectiveNext is where all

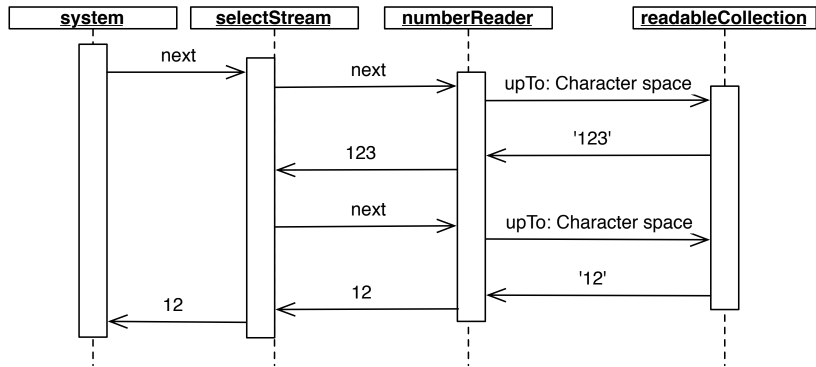

Figure 3.8: Chaining streams

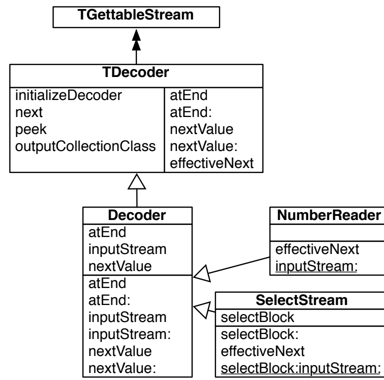

Figure 3.9: The decoder and two possible clients.

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 the original Squeak hierarchy, 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.

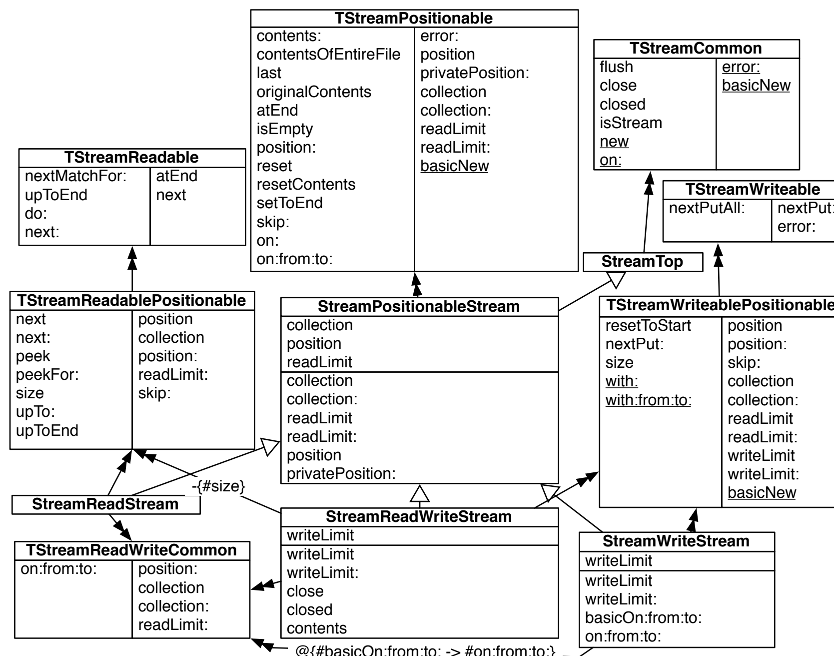

Figure 3.10: Schärli's refactored stream hierarchy.

3.7 Discussions

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

3.7.1 Comparison with previous work

There is no previous work building a library from scratch using traits. However, Schärli *et al.* [13] were the first to refactor the collection and stream hierarchies using traits.

Figure 3.10 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.

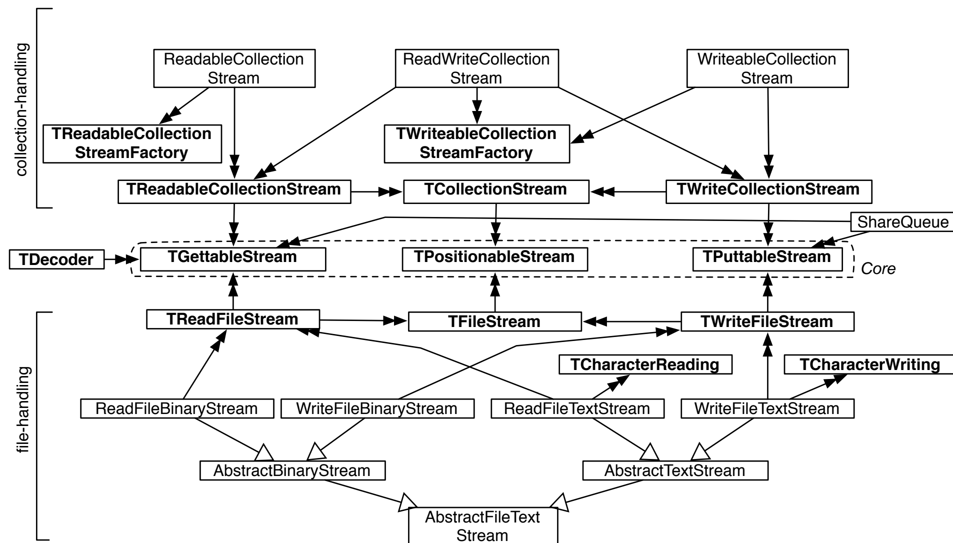

Figure 3.11: An overview of Nile first clients

3.7.2 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 constructors 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 constructors. 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 3.11).

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 3.2), 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 3.11 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.

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 3.3: Nile clients

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, 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 3.3 presents the current clients we implemented in Nile using traits as well as the number of implemented methods to get the desired behavior.

Table 3.4 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.

Trait	client classes	required methods	provided methods	$\frac{provided}{required}$
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 3.4: 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 3.5: Some metrics for the collection-based streams

Table 3.5 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 few classes. This is because Nile is designed to have fine grained and reusable components. 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 Nile is better: there is no canceled 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 composition of several traits 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 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.

3.7.3 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[3], 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 revealed to be really 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 3.12).

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

Accessor-use impact. Traits cannot define state, which is then accessed via accessor methods. As Squeak does not have a JIT compiler such as VisualWorks, using accessors instead of direct instance variable access has a cost. Table 3.6 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 3.12 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 3.6: Nile performances. Without accessors, Nile is faster than Squeak. But using them makes it slower.

The right of Figure 3.12 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.

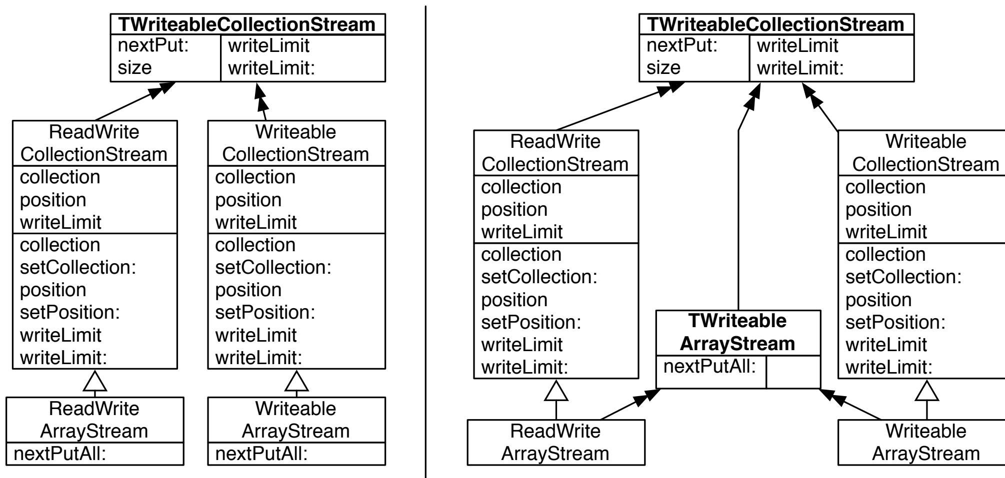

Figure 3.12: Two solutions to optimize nextPutAll:.

3.8 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 lack of visibility controls. In addition it should be noted that we did not encounter problem with alias in the context of recursive calls which is a known problem of traits.

3.8.1 A Squeak 3.9 trait implementation bug

Provided trait methods always takes precedence over superclass methods. When a class uses a trait, all methods in the trait are implemented as if they were implemented directly in the class, but provided methods take precedence over required methods (*i.e.,* they do not lead to a conflict). Now the Squeak 3.9 trait implementation has a bug because methods implemented in a superclass do not take precedence over required methods.

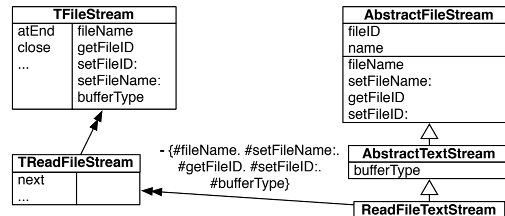

Figure 3.13: Simplified version of the file-based streams (complete version in Figure 3.6)

Figure 3.13, which presents a simplified version of the file-based streams, illustrates the problem. TFileStream requires four accessors for fileName and fileID plus one method named bufferType. These methods are all implemented in AbstractFileStream or AbstractTextStream. However, in class ReadFileTextStream, the traits take precedence and thus, the five methods are still a requirement. This is clearly a bug of the current implementation. The solution we chose in Nile is to explicitly remove all required methods from the imported trait (Figure 3.13).

3.8.2 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 over the class developers. However this would be solved if Smalltalk would have method access control. Stateful traits [1] solves this problem by allowing traits to contain private state. For example, if we had used the stateful trait model, methods TPositionableStream>>setPosition:, TWriteableCollectionStream>>writeLimit would not had been required.

However, developing Nile showed that stateful traits would not have been of great help. If we examine the trait TCollectionStream in Figure 3.5, we can see that implementing an instance variable collection here would had 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 constructor 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 still clunky and requires too much work from the developer.

Initializing in the constructor. Constructors can be used to initialize the variable too. 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 constructor which now needs more knowledge over the class it instantiates.

3.8.3 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 3.14.

Figure 3.14 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 TWriteableCollectionStreamTest>>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.

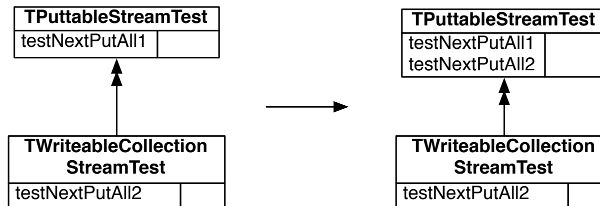

Figure 3.14: If the tester implements TPuttableStreamTest>>testNextPutAll2, the test will never be launched because TWriteableCollectionStreamTest>>testNextPutAll2 hides it.

3.9 Related work

We already compared our approach with the few work refactoring existing code using traits [13, 9]. We now present the approaches that automatic transform of existing libraries using FCA or other techniques. FCA was used in different ways.

Godin [6] 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 [14]. 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 [15]. 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 [9]. 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 [3]. 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 [11] proposes automatic refactorization 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 [2]. 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 [4].

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 composition, in the context of mixin-like language abstractions. Another important difference is that we do rely on algorithm. 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 benchmark reference point.

3.10 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 ground new implementation. We have started from a textual description, the ANSI Standard, and already existing implementations, VisualWorks and Squeak stream libraries. Our result is a completely new implementation, named Nile, of the stream hierarchy which does not share any line of 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% code less 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).

4 Résultats

Optimisation de la vitesse. Comme il est dit dans la section 3.7.3, une partie du travail a été d'optimiser Nile pour arriver à des performances équivalentes à celle de l'implémentation de base. Le premier test de performance a montré que Nile allait deux cents fois plus lentement lors de l'utilisation de la méthode nextPutAll:. Après une légère modification, Nile était un peu moins de deux fois plus lent.

Après deux jours d'optimisations plus fines, Nile est plus rapide sur les quatre tests de performance implémentés comme le montre le tableau 4.1. Les optimisations et les tests de performance se limitent aux streams de caractères qui sont de loin les plus utilisés.

Méthode testée	Squeak	Nile	$\frac{Nile-Squeak}{Nile}$
	(opérations/seconde)		
next	41.5	45	8%
next:	98.2	158.2	38%
nextPut:	24.9	42.4	41%
nextPutAll:	115.9	120.2	4%

Tab. 4.1: Tests de performance pour les streams de caractères.

Ces bons résultats permettent de nous assurer que les traits ne sont pas une limite inhérente aux performances d'une application. Les traits permettent un design clair, mais ils n'empêchent pas les modifications ciblées, comme l'optimisation des performances, sur une partie d'un projet.

Problèmes liés à l'utilisation des traits. La section 3.8 nous a montré plusieurs problèmes rencontrés lors du développement de Nile. Certains de ces problèmes étaient dus à l'implémentation des traits, d'autres au modèle des traits et d'autre encore à Smalltalk ou à l'approche orientée objet en général.

Le problème principal vient de la pollution d'interface dû au fait que les traits n'ont pas d'état mais nécessite des accesseurs. Au final, les classes possèdent des méthodes qui ne devraient pas faire partie de leur interface publique.

Deux autres modèles de traits ont été développés au cours des années précédentes pour parer à ses problèmes. Les traits *stateful* permettent de définir de l'état dans les traits et les *freezable* traits sont utiles lorsque le développeur d'un trait veut masquer des méthodes aux classes qui souhaitent utiliser ce trait. Après l'analyse de Nile, il apparaît que ces deux modèles n'auraient pas beaucoup participé à la

résolution de nos problèmes.

Réutilisabilité des traits. La section 3.7.2 a montré que les traits apportent une grande réutilisabilité grâce à leur granularité fine. J'ai pu implémenter un grand nombre de librairies réutilisant les traits du core ou les traits des streams sur les collections.

Si ces librairies étaient développées en utilisant la bibliothèque de Squeak alors il faudrait utiliser l'héritage. Dans ce cas, il faudrait sous classer directement ou indirectement Stream, qui contient déjà trop de méthodes : la classe Stream possède déjà toutes les méthodes de lecture et d'écriture. Les traits que j'ai définis dans Nile permettent aux différentes classes implémentées de n'avoir que les méthodes dont elles ont besoin : la lecture, l'écriture ou le positionnement. C'est un des avantages les plus importants des traits : ils ont une granularité fine qui bénéficie aux clients.

Choix d'une classe ou d'un trait. Une des questions importantes lorsque l'on développe avec les traits est de savoir quand utiliser une classe et quand utiliser un trait. Les traits ont beaucoup d'avantages, mais ils nécessitent plus de travail ; il faut en effet toujours définir une classe pour les utiliser et ils ne peuvent définir d'état.

Nous pensons que le choix par défaut doit être la classe. C'est seulement lorsque le besoin se présente, à cause de duplication de code, que la classe doit être refactorisée en utilisant les traits.

5 Conclusion

Problématique. Les traits permettent de factoriser du comportement pour les classes. Des travaux ont déjà montré l'efficacité des traits dans le contexte de la refactorisation de hiérarchies existantes. Cependant aucune recherche n'avait été menée sur leur utilisation dans un nouveau projet.

Solution. Le travail présenté dans ce mémoire de recherche est la première réalisation de projet pensé pour les traits. J'ai redéveloppé complètement une bibliothèque de streams, appelée Nile, en me basant sur une spécification plutôt que sur une implémentation existante. Cela m'a permis de m'affranchir des limitations du design d'origine dues à l'utilisation de l'héritage simple. La spécification apportée par ANSI se base sur des protocoles réutilisables qui se prêtent bien à une implémentation à base de traits.

Résultats. Le travail réalisé a ouvert sur la rédaction d'un article pour la conférence International Smalltalk Conference. Cet article, qui constitue le cœur de ce mémoire de recherche présente le design de Nile, des discussions sur l'utilisation des traits dans le cadre du développement d'un projet et aussi l'identification de problèmes liés aux traits.

Travaux futurs sur les traits. Au niveau des traits, de nombreux travaux continuent à être effectués. Un modèle pour les traits avec état a été défini ainsi qu'un modèle qui permettrait à un trait de cacher ses méthodes privées de façon à éviter les conflits. Cependant, il n'est pas encore clair que ces modèles soient intéressants à l'utilisation. Nile sera utilisé pour vérifier la validité et l'intérêt des modèles de traits futurs.

Travaux futurs sur Nile. Nile ne définit pas encore toutes les fonctionnalités de la bibliothèque de Squeak. Mon travail va consister à poursuivre son développement pour un possible remplacement de la bibliothèque d'origine.

Conclusion. En conclusion, Nile a permis de mieux évaluer les traits dans le cadre d'un projet de développement complet. Grâce au travail réalisé, nous avons pu nous rendre compte des problèmes soulevés par l'utilisation des traits et leurs réels avantages. Cette expérience va servir à mieux cibler les recherches futures dans le domaine.

Bibliographie

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

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

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

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

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

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

[7] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future : The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326. ACM Press, November 1997.

[8] Sonia E. Keene. *Object-Oriented Programming in Common-Lisp.* Addison Wesley, 1989.

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

[10] Bertrand Meyer. *Object-oriented Software Construction.* Prentice-Hall, 1988.

[11] Ivan Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *Proceedings of OOPSLA '96 (11th Annual Conference on*

Object-Oriented Programming Systems, Languages, and Applications), pages 235–250. ACM Press, 1996.

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

[13] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Composable encapsulation policies. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'04)*, LNCS 3086, pages 26–50. Springer Verlag, June 2004.

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

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

[16] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, Reading, Mass., 1986.

A La syntaxe Smalltalk

Le tableau A.1 montre les éléments de base de la syntaxe Smalltalk.

.	le point sépare les instructions.
:=	le symbole := sert à l'affectation.
^	L'accent circonflexe sert à retourner des valeurs à la méthode appelante.
'chaîne de caractères'	les simples quotes délimitent les chaînes de caractères.
[bloc]	les crochets délimitent les closures, appelées bloc en Smalltalk.
[:arg1 :arg2 | bloc]	les blocs peuvent prendre des arguments.
|tmpVar1 tmpVar2|	les pipes servent à déclarer des variables temporaires. Leur durée de vie et leur portée est limitée à la méthode.
"commentaire"	les commentaires sont entre guillemets.
$a	représente le caractère a, instance de la classe Character.
#symb	représente un symbole instance de ByteSymbol.
#(a b)	représente un tableau instance de la classe Array qui contient deux élément a et b.

Tab. A.1: Éléments de base de la syntaxe

En Smalltalk, il existe six pseudo variables. Elles sont données dans le table A.2.

true et false	ces constantes sont des instances des classes True et False.
nil	nil est la valeur par défaut de toute variable.
self	self représente l'objet courant. Il est équivalent au this de Java.
super	indique que le lookup doit se faire à partir de la super classe.
thisContext	thisContext représente la pile d'exécution en cours.

Tab. A.2: Les pseudo variables

Il n'y a pas de fonction en Smalltalk, seulement des méthodes que l'on exécute sur des objets. Il y a trois types de méthodes :

Les méthodes unaires. Les méthodes unaires n'ont pas d'argument.

```
"Appel de la méthode removeAll sur l'objet uneCollection :"

uneCollection removeAll.
```

Les méthodes binaires. Les méthodes binaires prennent un argument et leurs noms est composé de symboles ('.' ',' '+' '-'...).

```
"Appel de la méthode + sur l'objet 1 avec le paramètre 3 :"

1 + 3
```

Les méthodes à mots-clés. Enfin, les méthodes à mots-clés possèdent un nombre illimité de paramètres. Leurs noms sont composés de plusieurs groupes de caractères (autant qu'il y a d'arguments) se terminant chacun par le caractère ':'. Les arguments se placent après chaque caractère ':'.

```
 "Appel de la méthode replaceFrom:to:with: qui prend 3 arguments sur
 l'objet uneCollection :"

uneCollection replaceFrom: 1 to: 6 with: uneAutreCollection.

"L'équivalent java serait :"
uneCollection.replaceFromToWith(1, 6, uneAutreCollection);
```

Priorité des messages. Lors de la lecture d'une expression les messages sont évalués dans l'ordre suivant :

1. Méthodes unaires ;
2. Méthodes binaires ;
3. Méthodes à mots-clés.

Par exemple :

```
5 factorial + 5 gcd: 5
```

Doit être lu de la manière suivante :

```
((5 factorial) + 5) gcd: 5
```

Il n'y a donc pas de priorité pour les opérateurs mathématiques. Ainsi `3 + 4 * 3` vaut 21 et non pas 15. Il faut donc écrire `3 + (4 * 3)` si l'on veut que la multiplication prenne précédence sur l'addition.

Les cascades. Il est possible et très fréquent en Smalltalk d'envoyer plusieurs messages au même objet. Cela se fait grâce à l'opérateur ';'.

```
uneCollection
      add: unObjet;
      add: unAutreObject;
      add: unTroisiemeObjet.
```

La définition d'une nouvelle méthode se fait de la façon suivante :

```
String>>lineCount

  "Compte le nombre de lignes dans le receveur (une chaîne de
   caractères). Chaque carriage return (cr) compte pour une nouvelle
```

```
ligne."

| count |
count := 1.
self do:
   [:c | (c == Character cr)
            ifTrue: [count := count + 1]].
^ count
```

Le morceaux de code précédent doit s'interpréter de la façon suivante :

1. une nouvelle méthode appelée lineCount est définie dans la classe String.
2. un commentaire explique le but de cette méthode.
3. une variable temporaire count est déclarée puis initialisée à 1.
4. la méthode do: est exécutée sur l'objet en cours (self). Cette méthode prend un argument de type bloc, et le bloc doit avoir un argument. Le bloc sera évalué pour chaque caractère de la chaîne. La méthode do: est équivalent à une boucle foreach ou a une fonction mapcar en Common Lisp.
5. Le bloc définit un argument appelé c. Cet argument sera remplacé par chaque caractère de la chaîne, l'un après l'autre.
6. Le corps du bloc commence par comparer le caractère en cours avec le caractère de fin de ligne. Le caractère de fin de ligne s'obtient avec '\n' en C et en appelant la méthode cr sur la classe Character en Smalltalk.
7. Le résultat de la comparaison est un booléen : soit true soit false.
8. Sur ce booléen, la méthode ifTrue: est appelée. Cette méthode prend un bloc en paramètre et n'évalue ce bloc que si le receveur est true.
9. Si le booléen est true, le bloc est exécuté et on ajoute 1 à la variable count.
10. Quand la méthode do: a fini d'itérer sur tous les caractères, elle se termine puis la valeur de la variable count est retournée.