Université Lille 1

Laboratoire INRIA

Lieu Villeneuve d'ascq, France

Maître de stage Stéphane Ducasse

Tuteur Anne Françoise Le Meur

Rapport de stage

Library engineering with traits

Ingénieurie de bibliothèques à base de traits

Cyrille Delaunay

Mars à Septembre 2009

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

Table des matières

1. Introduction

Du 1 Mars au 4 Septembre, j'ai éffectué un stage au sein de l'équipe RMOD à l'Inria (Institut Nationale de Recherche en Informatique et Automatique) à Villeneuve d'asq . Celui-ci a été pour moi l'opportunité d'établir un premier contact avec le monde de la recherche, secteur qui m'intéresse et me motive particulièrement, d'acquerir de nouvelles connaissances et d'appronfondir de manière générale des notions en programmation orientée objet.
En effet, l'objectif de RMoD est d'aider à la remodularisation des applications orientées objets en s'interessant de manière complémentaire à la réingenieurie et à la définiton de nouveaux constructeurs pour la modularité. Dans ce dernier contexte, elle travaille à la validation du modèle de traits.

Les traits sont des unités de réutilisation constituées essentiellement de méthodes qui peuvent être appliquées à des classes indépendamment des contraintes d'héritage. Les traits peuvent être composés pour définir d'autres traits ou des classes . Les traits laissent le contrôle de la résolution de conflits au composeur de la classe/trait. Les traits ont reçu une certaine attention dans la communauté scientifique : Les traits ont été ajoutés à Perl 6, Squeak [IKM+97], Scala [sca], Slate [Sla] et Fortress de SUN Microsystems [for]. Plusieurs extensions ont été proposées [Den04, Qui04, BDNW07, DWBN07] ainsi que des systèmes de types [FR03, SD05, NDS06, LS07]. 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.

Le but de ce travail est de vérifier expérimentalement les possibles améliorations pouvant être apportées par les traits et d'évaluer leur degré d'utilisabilité . Plus spécifiquement, nous souhaitons répondre aux questions suivantes :

- Quelle est la granularité idéale des traits afin de favoriser leur 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 visibles dans les bibliothèques existantes ?
- Quelles sont les limites et les contraintes que nous rencontrerons lors de l'utilisation des traits ?

En vue de répondre à ces questions j'ai eu l'occasion d'utilser les traits dans deux differents contextes :

- en tant qu'outil dans une architecture basée sur l'héritage simple pour l'écriture de tests.
- en tant qu'unité de construction pour la redéfinition de librairies .

Afin de rendre compte au mieux des mois passés dans l'équipe, nous nous interesserons dans une première étape à l'environnement et au contexte dans lequel le stage s'est réalisé. Je présenterais l'équipe RMOD ainsi que le modèle de traits(2). Nous nous attarderons ensuite sur les projets sur lesquels j'ai pu travailler : La couverture de la librairie des Collections à l'aide de traits de tests (3) et la remodularisation de la librairie des Stream : Nile (4). Dans une dernière étape je décrirai la réalisation et l'organisation du travail éffectué durant le stage (5) avant d'établir un bilan (6).

2. Cadre du projet et objectifs

2.1. L'INRIA

Secteur d'activité. L'INRIA, institut national de recherche en informatique et en automatique, placé sous la double tutelle des ministères de la recherche et de l'industrie, a pour vocation d'entreprendre des recherches fondamentales et appliquées dans les domaines des sciences et technologies de l'information et de la communication (STIC). L'institut assure également un fort transfert de technologie en accordant une grande attention à la formation par la recherche, à la diffusion de l'information scientifique et technique, au développement, à l'expertise et à la participation à des programmes internationaux.

Composantes. L'INRIA accueille 3 800 personnes réparties dans ses 8 centres de recherche situés à Rocquencourt, Rennes, Sophia Antipolis, Grenoble, Nancy, Bordeaux, Lille et Saclay. 2 800 d'entre elles sont des scientifiques de l'INRIA et d'organismes partenaires (CNRS, universités, grandes écoles) qui travaillent dans plus de 160 équipes-projets de recherche communes. Un grand nombre de chercheurs de l'INRIA sont également enseignants, et leurs étudiants (environ 1 000) préparent leur thèse dans le cadre des équipes-projets de recherche de l'INRIA.

Le centre de recherche de Lille. Le centre de recherche INRIA Lille - Nord Europe, sous la direction de Max Dauchet, regroupe dès sa création 10 équipes de recherche installées dans un bâtiment de $4000m^2$ acquis grâce à l'aide des collectivités territoriales et des fonds européens. Il accueille plus de 220 personnes, dont près de la moitié est rémunérée par l'Institut. Ce centre INRIA est un atout déterminant pour la compétitivité du Nord - Pas de Calais en matière de recherche et d'innovation.

2.2. L'équipe RMoD

Secteur d'activité L'objectif de RMoD est d'aider à la remodularisation des applications orientées objets. Cet objectif est attaqué suivant deux axes complémentaires : la réingénierie et la définition de nouveaux constructeurs dans les langages de programmation. Dans le cadre de la réingénierie nous allons proposer de nouvelles analyses pour comprendre et restructurer de grandes applications (métriques spécialisées, visualisations adaptées). Dans le contexte des constructeurs pour la modularité

nous allons travailler à la validation du modèle de traits ainsi que de nouveaux systèmes de modules. Nous allons travailler à la définition d'un noyau sécurisé. Ces travaux seront validés dans Pharo un environnement pour le développement dynamique d'applications web.

Remodularisation d'applications existantes. L'évolution des applications est limitée par la présence de couplages forts entre les différentes parties. C'est pourquoi répondre aux questions suivantes est crucial : comment peut-on substituer une partie en limitant l'impact sur les autres ? Comment identifier des éléments réutilisables ? Comment modulariser des applications à objets en présence de liaison tardive ? Nous allons enrichir Moose, notre environnement de réingénierie, avec un ensemble d'analyses. Nous décomposons notre travail en trois approches se recouvrant partiellement :

- Outils pour la comprehension des grandes applications (packages/modules)
- Analyses pour la remodularisation
- Qualité du logiciel

Eléments Sémantiques pour la Modularité. Alors que l'axe précédent s'attache à la remodularisation de logiciels existants, ce second axe se concentre sur la définition de nouveaux éléments sémantiques des langages pour la construction de logiciels flexibles et reconfigurables. Nous allons continuer notre effort sur les traits et classboxes mais aussi travailler sur de nouvelles aires telles que la sécurité dans les langages dynamiques. Nous allons travailler sur

- La définition d'un langage à traits purs
- La réconciliation entre les langages réflexifs et la sécurité.

Smalltalk. L'équipe RMOD utilise le langage Smalltalk. Smalltalk est un langage de programmation orienté objet, réflexif et dynamiquement typé. Il fut l'un des premiers langages de programmation à disposer d'un environnement de développement intégré complètement graphique. Il a été créé en 1972. Il est inspiré par Lisp et Simula. Il a été conçu par Alan Kay, Dan Ingals, Ted Kaehler, Adele Goldberg au Palo Alto Research Center de Xerox. Le langage a été formalisé en tant que Smalltalk-80 et est depuis utilisé par un grand nombre de personnes. Smalltalk est toujours activement développé. Smalltalk a été d'une grande influence dans le développement de nombreux langages de programmation, dont : Objective-C, Actor, Java et Ruby. Un grand nombre des innovations de l'ingénierie logicielle des années 1990 viennent de la communauté des programmeurs Smalltalk, tels que les Design Patterns (appliquées au logiciel), l'Extreme Programming (XP) et le refactoring. Ward Cunningham,

l'inventeur du concept du Wiki, est également un programmeur Smalltalk. Pharo, le projet sur lequel l'équipe travaille et l'environnement dans lequel j'ai réalisé mes différents travaux, est une variante de Smalltalk basée sur Squeak. Vous trouverez une description de la syntaxe Smalltalk en annexe A.

2.3. Les traits, une unité de comportement

Les traits 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 rapport de stage.

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 traits 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.

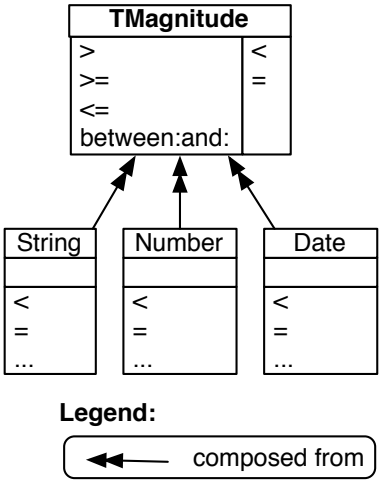

FIGURE 2.1.: Le trait TMagnitude et trois clients possibles.

La figure 2.1 présente un premier exemple d'utilisation de traits. 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 $=$.

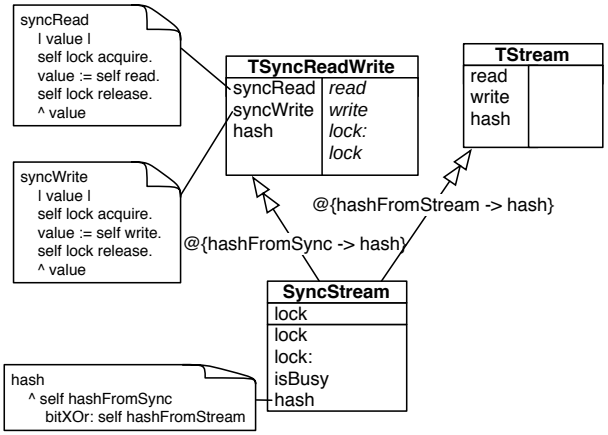

FIGURE 2.2.: La classe SyncStream est composée des deux traits TSyncReadWrite et TStream.

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)
```

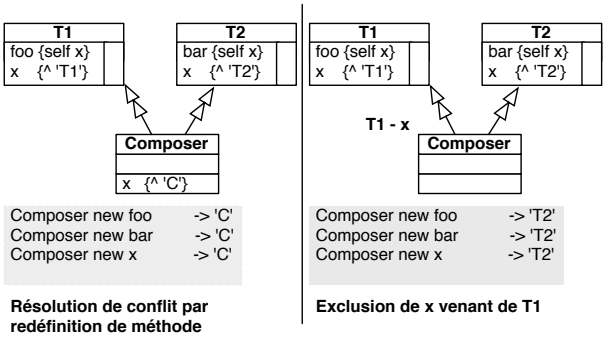

FIGURE 2.3.: Les stratégies de résolution de conflit : par redéfinition ou par exclusion de méthode.

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 sémantique 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.

2.4. Objectifs

Tests pour la bibliothèque des collections. Le premier travail qui m'a était attribué est la couverture de la librairie des collections à l'aide de traits de tests (la description de ce projet est détaillée au chapitre 3). Celui-ci s'éffectue dans l'optique dévaluer l'utilisation des traits pour l'écriture de tests. L'équipe a déjà commencé à écrire des traits ainsi que des classes de tests les utilisant. Cependant, certains points restent à compléter :

- Beaucoup de méthodes ne sont pas encore testées.
- Peu de classes sont testées á l'aide des traits de tests.
- Certains traits peuvent ne pas être bien définis (trop spécifiques).

Mon rôle est d'analyser, d'utiliser, et d'améliorer le travail déjà réalisé. Cet objectif a pu étŕe guidé á l'aide de plusieurs questions :

- Doit-on cŕeer de nouveaux traits ?
- Doit-on changer ceux qui sont trop spécifiques ?

- Quelle partie des classes n'est pas couverte ?
- Comment calculer le coverage statique ? (problème des méthodes trop hautes dans la hiérarchie qui ne font plus de sens pour les classes filles. Le coverage ne doit pas les prendre en compte).
- A t-on besoin de traits de tests pour couvrir les aspect spécifiques de certaines classes ?
- Quelle est une bonne granularité pour des traits de tests ?

Un point important est de couvrir au mieux les classes basiques (Array, Bag, OrderedCollection ...). Un second objectif est la réécriture de la librairie des Collection à l'aide de traits. Ainsi, les différentes classes de tests pourront être réutilisées lors de cette étape. Néanmoins, il sera important d'approuver les traits en testant un grand nombre de classes.

Compléter la librairie de streams Nile. Nile est un projet réalisé en majeur partie par Damien Cassou. Il consiste en la redéfinition de la librairie des Stream dans Pharo à l'aide de traits (la déscription de ce projet est détaillée au chapitre 4). Cette nouvelle librairie possède une structure déjà solide et assez complète. Cependant, un des objectifs final serait de remplacer totalement la hiérarchie actuelle dans Pharo par Nile. Cette étape nécessite certains ajustements :

- Les classes de Stream utilisées actuellement dans le système n'ont pas toutes un équivalent dans Nile.
- Certaines méthodes de la hiérarchie actuelle (utilisées par le système) ne sont pas présentes dans Nile.

Mon travail consiste ainsi à :

- Reconstruire les classes manquantes en me basant sur la structure de Nile. Un des objectifs des trait étant de donner plus de liberté dans la définition des classes, chaque classe doit être redéfinie en faisant abstraction de leur structure actuelle dans une hiérarchie limitée par l'héritage simple. Toutes doivent être repensées afin de leur donner un sens meilleur.
- Tester chaque nouvelle classe.
- Analyser les méthodes non présentes dans Nile. S'interroger sur leurs rôles, leurs utilités dans la hierarchie actuelle. Les insérer au bon endroit dans la hiérarchie de Nile (possibilité de créer de nouveaux traits).

3. Couverture de la libraire des collections à l'aide de traits de tests

3.1. Réutilisation de tests

Un des principes fondamental de l'ingenierie logicielle est de favoriser la réutilisation de code plutôt que sa duplication. Ainsi, une question qui se pose naturellement est de savoir comment appliquer cette réutilisation aux tests unitaires. Comme les traits sont des unités de comportement composables, il serait intéressant de les utiliser et de les évaluer lors de la construction de tests unitaires. C'est ce que j'ai pu faire en travaillant sur la couverture de la librairie des collections[1] de Squeak à l'aide de traits (Pharo est basé sur Squeak et reprend actuellement la hiérarchie des collections de Squeak).

3.2. xUnit

La famille des frameworks de tests xUnit est née avec son incarnation Smalltalk : SUnit. Un test unitaire est un petit bout de code stimulant des objets puis vérifiant des assertions concernant leur réaction. Les objets stimulés sont collectivement appelés fixture de test. Les tests unitaires executés dans le contexte d'une fixture donnée sont groupés ensemble à l'intérieur d'un cas de tests. Le framework automatise l'execution des tests unitaires, en assurant que chacun d'entre eux est lançé dans le contexte d'une fixture venant d'être initialisée.

Dans SUnit, un cas de tests est une sous-classe de TestCase définissant les tests unitaires en tant que méthodes et leur fixture partagée en tant qu'état. Dans l'exemple ci-dessous[2], une telle sous-classe est présentée, appelée SetTest, avec les variables d'instance full et empty. On définit ensuite la méthode setUp, laquelle initialise les deux variables d'instance pour obtenir différents sets comme fixture, puis la méthode de test unitaire testAdd, laquelle vérifie qu'ajouter un élément fonctionne sur le set vide :

[1] Ce projet a été l'objet du papier *"Reusing and Composing Tests with Traits"* réalisé par Stéphane Ducasse, Damien Pollet, Alexandre Bergel et Damien Cassou. Il a été présenté à la Technology of Object-Oriented Languages and Systems (TOOLS) International Conference 2009. Certaines parties de ce rapport s'inspirent de ce document.

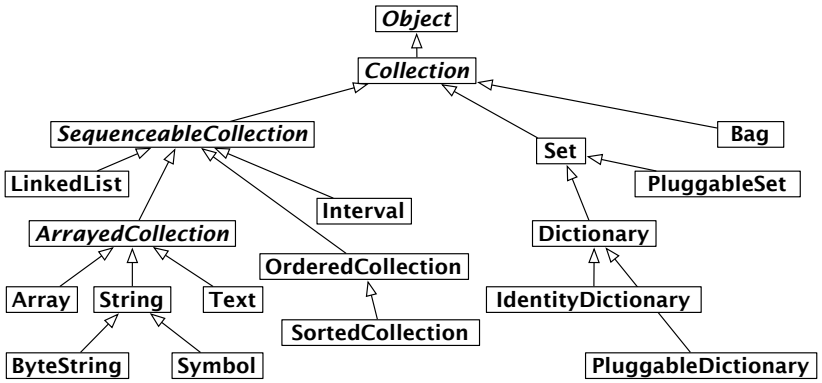

FIGURE 3.1.: Les pincipales collections dans Squeak.

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

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

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

3.3. La diversité de la librairie des Collections dans Smalltalk

Dans Smalltalk, quand quelqu'un parle d'une collection sans être plus spécifique à propos du type de celle-ci, il ou elle signifie un objet supportant les protocoles connus pour énumerer les élément et pour tester l'appartenance. Toutes les collections comprennent les messages de type *testing* comme includes :, isEmpty et occurrencesOf :. Toutes les collections comprennent les messages de type *enumeration* comme do :, select :, collect :, detect :ifNone :, inject :into : et beaucoup d'autres.

Le tableau 3.1 résume les protocoles standards supportés par la plupart des classes dans la hiérarchie des collections. Ces méthodes sont définies, redéfinies, optimisées et parfois même annulée dans les sous-classes de Collection [3]

Grand ensemble de comportements différents. Derrière cette uniformité basique, il existe beaucoup de collections différentes. Chaque collection comprend un groupe

[2] Un résumé de la syntaxe smalltalk est disponible en annexe A

[3] Notez qu'annuler des méthodes est un pattern qui existe à l'extérieur du monde Smalltalk. Java annule plusieurs méthodes dans son système de collections en levant UnsupportedOperationException.

Table 3.1.: Protocoles standards des collections

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

de protocoles particulier où chaque protocole est un groupe de méthodes sémentiquement cohésives. Le tableau 3.2 présente certaines des différentes facettes des collections ainsi que leur implémentation. Pour une compréhension générale, le point clé à avoir en tête est que les protocoles présentés dans le tableau 3.1 entrecoupe le groupe des différents comportements présentés dans le tableau 3.2. Ci-dessous l'énumeration des comportements clés que l'on peut trouver dans le système de collection de Smalltalk :

Sequenceable : Les instances de toutes les sous-classes de SequenceableCollection démarre à partir d'un premier élément et avancent dans un ordre bien défini jusqu'à un dernier élément. De l'autre coté, Set, Bag, Dictionary ne sont pas *Sequenceable*.

Sortable : Une SortedCollection maintient ses élément dans un ordre trié.

Indexable : La plupart des collections *sequenceable* sont également *indexable*. Cela signifie que les éléments de telles collections peuvent être accedés à l'aide d'un index numérique, en utilisant le message at:. Array est la structure *indexable* la plus familière. LinkedList and SkipList sont *sequenceable* mais pas *indexable*, ce qui signifie quelles comprennent les messages first et last mais pas at:.

Keyed : Les instances de Dictionary et ses sous-classes accédent à leurs éléments par un indice non numérique. Tout type d'objet peut être utilisé comme clé

Table 3.2.: Quelques-uns des principaux comportements de la librairie des collections.

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

pour référencer une association.

Mutable : La plupart des collection sont *mutable* exepté Interval et Symbol. Un Interval est une collection représentant une chaîne d'Integers. Il est *indexable* avec Interval >> at :, mais ne peut pas être modifié avec at:put:.

Growable : Les instances de Interval et Array ont toujours une taille fixe. Les autres sortes de collections peuvent grandir après création.

Accepts duplicates : Un Set filtre les doublons, à l'inverse d'un Bag. Cela signifie qu'un même élément peut apparaître plusieurs fois dans un Bag, mais pas dans un Set. Dictionary, Set et Bag utilisent la méthode = fournie par les éléments ; la variante Identity de ces classes utilise la méthode ==, qui teste si les arguments sont le même objet. La variante Pluggable utilise une relation d'équivalence arbitraire fournie par le créateur de la collection.

Contains specific elements : La plupart des collections accepte n'importe quel type déléments. Cependant, un String, CharacterArray ou Symbol n'accepte que des éléments de type Character. Un Array contiendra n'importe quelle sorte d'objet, mais un ByteArray ne contiendra que des Bytes. Une LinkedList est constrainte de contenir des éléments de type Link et un Interval contient seulement des éléments de type Integer.

3.4. Création et réutilisation des traits

L'approche utilisée ici pour rétuiliser les tests est basée sur la définition et la composition de traits. Celle-ci se caractérise par les étapes suivantes :

Identification et selection de protocoles. On selectionne les principaux protocoles des classes principales. Par exemple, les messages définis par la classe abstraite Collection ont été groupés à l'intérieur d'ensembles cohérents, influencés par les catégories de méthodes éxistantes et du standard ANSI.

Défintion des traits. Pour chaque protocole, on définit des traits. Comme je le montrerai plus tard, chaque trait définit un ensemble de méthodes de tests et un ensemble de méthodes d'accés faisant le lien avec la fixture. Notez qu'un protocole peut conduire à plusieurs traits car le comportement associé à un ensemble de messages peut être différent : par exemple, ajouter un élément à une OrderedCollection ou un Set est typiquement différent et devrait être testé différemment (on définit deux traits TAddTest, TAddForUniquenessTest pour l'ajout d'un simple élément). Maintenant ces traits peuvent être réutilisés indépendament selon les propriétés de la classe testée.

Composer les classes de tests avec les traits. Les classes de tests sont définies en composant les traits définis dans l'étape précédente et en spécifiant leur fixture.

3.4.1. Les traits de tests par l'exemple

L'approche est ici illustrée avec le protocole pour insérer et récupérer des éléments. Une méthode importante de ce protocole est at:put:. Quand elle est utilisée sur un Array, cette méthode Smalltalk est léquivalent java de a[i] = val : cela stocke une valeur à un index donné (numérique ou non). Cet exemple a été sélectionné pour sa simplicité et son ubiquité. Il est réutilisé pour tester le protocole d'insertion sur des classes provenant de deux différentes sous-hiérarchies de la librairie des collections : pour les sous-classes de SequenceableCollection comme Array et OrderedCollection, mais également pour Dictionary, qui est une sous-classe de Set et Collection.
D'abord, le trait TPutTest est défini, avec les méthodes de test suivantes :

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

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

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

Ensuite, on déclare que TPutTest a besoin des méthodes empty, nonEmpty, anIndex, aValue et anotherValue.
Les méthodes éxigées par un trait peuvent être assimilées aux paramètres du trait, *i.e.,* le comportement d'un groupe de méthodes est paramétré par ces méthodes dont il a besoin. Appliquées aux tests, ces méthodes et les méthodes définissant des valeurs par défaut jouent le rôle de crochets de customisation pour les tests : pour définir une classe de tests, le développeur doit fournir les méthodes requises, et il peut également redéfinir localement d'autres méthodes du trait.

3.4.2. Composer les classes de tests

Une fois que les trait sont définis, une classe de tests peut être définie en composant et en réutilisant ces traits. En particulier, la fixture doit être définie. Cela est fait dans la classe de composition en implémentant les méthodes dont le trait a besoin pour accéder à la fixture. Comme le chevauchement est possible entre les accesseurs de différents traits, la plupart du temps seulement peu d'accésseurs on besion dêtre définis localement après la composition d'un nouveau trait.
La définition suivante montre comment la classe de tests ArrayTest testant la classe Array est définie :

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

Elle utilise 9 traits, définit 10 variables d'instance, contient 85 méthodes, mais seulement 30 d'entre elles sont définies localement. (55 sont obtenues grâce aux traits). Parmis ces 30 méthodes, 12 définissent la fixture.

La superclasse de ArrayTest est CollectionRootTest. Comme expliqué plus loin, la classe CollectonRootTest est la racine des classes de tests pour les collections partageant un comportement commun comme *iteration*. ArrayTest définit plusieurs variables d'instance formant la fixture des tests. Ces variables sont initialisées dans la méthode setUp.

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

```
result := {SmallInteger . SmallInteger . SmallInteger . SmallInteger}.
empty := #().
```

La fixture est ensuite rendue accéssible par les tests en implémentant des méthodes triviales mais nécéssaires, empty et nonEmpty, requises par TEmptyTest :

```
ArrayTest >> empty
    ^ empty
```

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

TPutTest nécessite les méthodes aValue et anIndex, que l'on implémente en retournant une valeur spécifique comme indiqué ci-dessous dans la méthode de test testAtPut. Notez qu'ici la valeur retournée par aValue est absente du tableau stocké dans la variable d'instance example1 et retourné par nonEmpty. Cela assure que le comportement est bien testé.

```
ArrayTest >> anIndex
    ^ 2
```

```
ArrayTest >> aValue
    ^ 33
```

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

Ces exemple illustrent comment une fixture peut être réutilisée par touts les traits composés. Dans l'éventualité où le comportement d'un trait nécéssiterait une fixture différente, un nouvel état et de nouveaux accésseurs peuvent être ajoutés à la classe de tests.

La classe DictionaryTest est un autre exemple de classe de tests. Elle utilise également une version modifiée de TPutTest. Ce trait est adapté en supprimant sa méthode testAtPutOutOfBounds, parce que les bornes sont pour les collections indexées et ne font pas de sens pour les dictionnaires. La définition de DictionaryTest est la suivante :

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

DictionaryTest utilise 10 traits et définit 2 variables d'instance. 81 méthodes sont définies dont 25 le sont localement contre 56 apportées par les traits.

Pour cette classe, une procédure similaire apparaît. Nous définissons la méthode setUp. Notez qu'ici nous utilisons une méthode crochet classToBeTested afin de pouvoir réutiliser cette classe de tests en utilisant l'héritage.

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

DictionaryTest >> classToBeTested
    ^ Dictionary
```

Et de manière similaire on redéfinit la méthode requise pour proposer une clé adaptée aux dictionaires.

```
DictionaryTest >> anIndex
    ^ #zz
```

```
DictionaryTest >> aValue
    ^ 33
```

3.4.3. Combiner l'héritage et la réutilisation des traits

Il est intéressant de noter que la réutilisation basée sur l'héritage n'est pas opposée à celle basée sur les traits. Par exemple, la classe CollectionRootTest utilise les traits suivant : TIterateTest, TEmptyTest, et TSizeTest (voir Figure 3.2). CollectionRootTest est la racine de tout les tests. Par conséquent les méthodes obtenues par ces trois traits sont héritées par toutes les classes de tests. Ces méthodes auraient pu être directement définies dans CollectionRootTest, mais ont été groupées dans un trait pour les raisons suivantes :

- Les traits représentent une réutilisation potentielle. Il est courant dans Smalltalk de définir des classes qui ne sont pas des collection mais qui implémentent tout de même le protocole d'itération. Avoir séparé ces méthodes dans un trait va favoriser leure réutilisation.

- Un trait est une entité de première classe. Il rend les méthodes requises explicites et documente une interface cohérente.

- Finalement, comme l'objectif final est de proposer un nouveau design pour la librairie des collections, ces protocoles seront probablement réutilisés de manière différente.

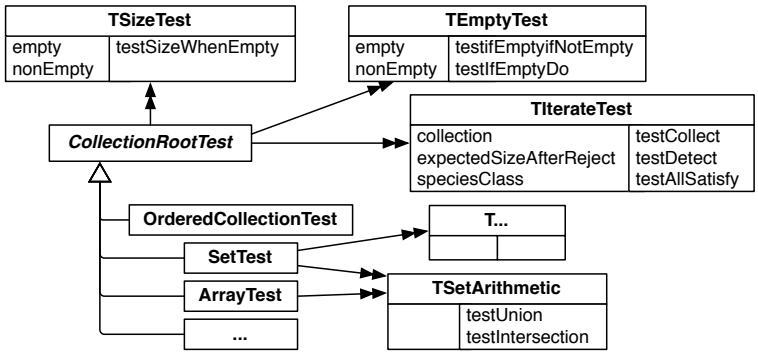

FIGURE 3.2.: Réutilisation des tests par la composition de traits et par l'héritage (la représentation des traits montre les méthodes requises à gauche et celles offertes à droite).

3.5. Contribution

3.5.1. Améliorations apportées

Comme cela est décrit précedemment, la couverture de la librairie des collections n'était pas complète. Un de mes objectifs était de l'augmenter de manière conséquente et ainsi de solidifier l'ensemble des traits de tests. Cela a été fait et a permis plus spécifiquement d'obtenir les modifications décrites ci-dessous.

Définition de nouveaux traits. La couverture de la librairie des collections était en partie incomplète car toutes les méthodes nétaient pas testées pour chaque classe de tests. En effet, beaucoup de méthodes provenant de protocoles communs n'apparaissaient pas dans les tests. Pour ces méthodes, j'ai créé de nouveaux traits. De manière générale, les principaux protocoles communs ont été groupés dans des traits et les méthodes non testées pour une classe sont souvent des méthodes spécifiques à celle-ci (le tableau 3.4 décrit les traits de tests actuellement présents).

Définition de nouvelles classes de tests. L'autre manque au niveau de la couverture obtenue était le nombre de classes testées. Touts les principaux types de collection ont maintenant été testés.

Une meilleure définition des traits. Avant mon stage certains traits et leurs tests étaient définis de manière trop spécifiques ou avec une granularité trop grosse. Tester plus de classes a permis de mettre en évidence ces problèmes. J'ai donc modifié ces traits afin qu'ils conviennent aux classes que je testais ainsi que pour augmenter leur réutilisabilité.

Les traits obtenus à ce stade semblent proposer une structure solide pour tester tout type de collections. J'ai testé des classes en partant de zéro (FloatArray, Heap) et chaque trait s'est bien emboîté aux classes de tests. Cependant, je pense qu'utiliser des traits de tests nécessite un réajustement permanent selon les différents sous-types testés. Il est difficile de les créer en ayant tous les comportements spécifiques en tête (d'autant plus si la librairie est grande). Il n'est donc pas impossible qu'une certaine spéficité ne nous soit pas venue à l'esprit et engendrera un modification des traits.

De meilleurs couvertures. Le tableau 3.3 présente les différentes couvertures que j'ai pu obtenir pour les principaux types de collections. Ces résultats sont obtenus à l'aide d'un outil d'analyse que j'ai réalisé. Celui-ci permet de spécifier statiquement les méthodes à ne pas prendre en compte. Il existe différentes raisons pour ne pas prendre en compte une méthode :

- Elle est définie trop haut dans la hiérarchie et ne fait pas de sens pour un certain sous-type.
- Elle est interne à la classe (privée) et ne fait pas partie de l'interface publique accessible par l'utilisateur.
- Elle est une méthode très spécifique ajoutée pour une utilisation trés spécifique.

	Couverture obtenue
Array	94.5%
String	91%
Bag	88.4%
Interval	93%
OrderedCollection	93.9%
Set	85.9%
SortedCollection	92.3%
LinkedList	90.5%
Dictionary	85%.

Table 3.3.: Quelques couvertures obtenues pour différentes collections

3.5.2. Résultats obtenus

Un nombre important de traits. Comme cela a été décrit précedemment, la librairie des collections dispose d'un grand nombre de comportements s'appliquant orthogonalement à la hiérarchie d'héritage (*Accepts Duplicates*, *Sequenceable*, *Growable* ...). Chaque protocole peut ensuite être défini de manière spécifique à chacun de ces comportements :

- Il peut ne pas être supporté (par exemple les collections qui ne sont pas extensibles ne supportent pas les méthodes de type `add:`).
- Certaines de ses méthodes peuvent fournir un résultat particulier (par exemple la concaténation de collections n'acceptant pas les doublons doit éliminer les possibles doublons).

Dans ce dernier cas, des tests doivent tester cette particularité. Mon choix a été de créer plusieurs versions des traits selon ces comportement spécifiques (ce qui est visible dans le tableau 3.4). Ainsi un trait (associé à un protocole) peut avoir jusqu'à trois version différentes. Cela a considérablement augmenté le nombre de traits. Néanmoins, chacun de ces traits peut être composé efficacement et simplement selon les spécifités de chaque classe testée.

Gains obtenus. Le tableau 3.4 présente, pour chaque trait de tests, différents calculs :

- La première colonne présente d'un coté, le nombre d'utilisateurs déclarant directement l'utilisation du trait et de l'autre, le nombre d'utilisateurs utilisant le trait en prenant en compte l'héritage des classes de tests.
- La deuxième colonne présente d'un coté le nombre de méthodes requises par le trait et de l'autre le nombre de méthodes proposées.
- La dernière colonne présente le nombre de réutilisations des tests dans le système. Ce nombre est normalement égal aux nombre d'utilisateurs directs multiplié par le nombre de tests proposés par le trait. Cependant, il se peut qu'une classe de tests *élimine* un certain test dans sa déclaration car il ne lui convient pas. Dans ce cas, le nombre obtenu est plus petit.

Travaux à venir. La librairie des collections n'est pas encore totalement couverte. Néanmoins, cet objectif ne semble pas nécessaire pour le moment et se complétera naturellement avec une future réimplémentation de la librairie à l'aide de traits. En effet, cette nouvelle librairie ne donnera peut être pas d'équivalents à tous les types de collections existants, certains pouvant paraître trop spécifiques ou superflus. Dans ce cas, les classes de tests existantes seront gardées et testeront les nouvelles versions, et les traits définis serviront à tester les classe non testées présentes dans cette nouvelle hiérarchie.

Table 3.4.: Quelques résultats à propos des traits définis dans la hiérarchie des collections

Test trait	**Users** dir. / inh.	**Methods** req. → prov.	**Effective unit tests**
TAddForUniquenessTest	1 / 2	3 → 5	5
TAddTest	5 / 7	3 → 9	42
TAsStringCommaAndDelimiterSequenceableTest	9 / 10	3 → 10	90
TAsStringCommaAndDelimiterTest	4 / 8	3 → 10	40
TBeginsEndsWith	9 / 10	2 → 5	45
TCloneTest	10 / 14	2 → 4	38
TConcatenationEqualElementsRemovedTest	1 / 2	3 → 4	4
TConcatenationTest	2 / 4	3 → 4	8
TConvertAsSetForMultiplinessIdentityTest	5 / 6	2 → 8	38
TConvertAsSetForMultiplinessTest	4 / 7	1 → 3	12
TConvertAsSortedTest	11 / 15	1 → 4	44
TConvertTest	12 / 16	3 → 10	116
TCopyPartOfSequenceable	9 / 10	4 → 11	95
TCopyPartOfSequenceableForMultipliness	7 / 8	1 → 5	35
TCopyPreservingIdentityTest	1 / 2	1 → 1	1
TCopySequenceableSameContents	9 / 10	3 → 7	59
TCopySequenceableWithOrWithoutSpecificElements	9 / 10	2 → 7	57
TCopySequenceableWithReplacement	7 / 8	4 → 7	47
TCopySequenceableWithReplacementForSorted	2 / 3	3 → 3	6
TCopyTest	12 / 16	5 → 9	105
TCreationWithTest	8 / 11	2 → 8	61
TEmptySequenceableTest	2 / 3	4 → 10	20
TEmptyTest	5 / 7	2 → 11	52
TGrowableTest	2 / 4	4 → 3	6
TIncludesTest	4 / 7	4 → 8	32
TIncludesWithIdentityCheckTest	9 / 12	5 → 10	89
TIndexAccessTest	8 / 9	3 → 12	94
TIndexAccessForMultiplinessTest	7 / 8	2 → 9	57
TIterateSequencedReadableTest	9 / 10	3 → 24	216
TIterateTest	2 / 4	3 → 21	42
TOccurrencesForMultiplinessTest	8 / 11	5 → 6	48
TOccurrencesTest	4 / 7	3 → 4	16
TPrintOnSequencedTest	8 / 9	2 → 7	56
TPrintTest	5 / 10	2 → 7	35
TPutBasicTest	5 / 7	5 → 4	19
TPutTest	4 / 5	4 → 9	36
TRemoveByIndexTest	1 / 2	2 → 11	11
TRemoveForMultiplenessTest	3 / 5	5 → 10	30
TRemoveTest	4 / 7	3 → 9	36
TReplacementSequencedTest	5 / 6	5 → 6	30
TSequencedConcatenationTest	7 / 8	3 → 3	21
TSequencedElementAccessTest	9 / 10	5 → 18	161
TSequencedStructuralEqualityTest	8 / 9	2 → 9	72
TSetArithmetic	11 / 15	6 → 12	131
TSortTest	3 / 4	2 → 5	15
TSubCollectionAccess	9 / 10	1 → 7	63
TSizeTest	2 / 4	2 → 2	4
TStructuralEqualityTest	5 / 10	2 → 4	20

4. Nile, une librairie basée sur les traits

4.1. Un projet complet basé sur les traits

L'approche de Nile est basée sur le design et l'implémentation à base de traits d'une librairie non triviale en partant de zero[1]. L'objectif choisi est de construire une librairie de streams suivant les spécifications ANSI Smalltalk tout en restant compatible avec les implementation Smalltalk actuelles. Nile possède 18% de méthodes en moins et 15% de bytecodes en moins que la librairie des streams de Squeak correspondante. De plus, Nile n'a ni méthodes annulées ni méthodes implémentées trop haut dans la hiérarchie. Il y a seulement trois surcharges de méthodes en comparaison aux quatorze de Squeak.

4.2. Aperçu et coeur de Nile

Nile est construit autour d'un coeur de traits proposant les fonctionnalités de base reflétant le standard ANSI. Le coeur consiste en seulement trois traits et est utilisé ensuite dans plusieurs librairies dont nous parlerons plus tard. Les streams basés sur les collections et sur des fichiers font partie des librairie les plus importantes. Les autres librairies sont des supports pour l'écriture de caractères et les décodeurs (Streams pouvant être chainés). La figure 4.1 présente un aperçu de Nile.

Nile a été créé autour de trois traits indépendants, refétant le standard ANSI :

[1]Ce projet a été l'objet du papier *"Redesigning with Traits : the Nile Stream trait-based Library"* réalisé par Damien Cassou, Stéphane Ducasse et Roel Wuyts, professeur a l'Université Libre de Bruxelles. Il a été soumis à la International Smalltalk Conference. Certaines parties de ce rapport s'inspire de ce document.

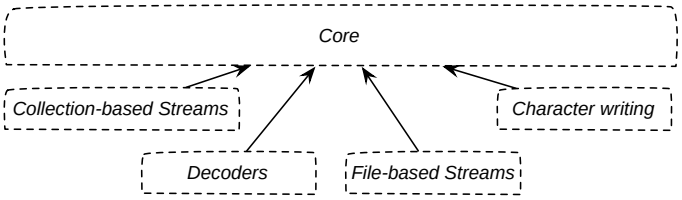

FIGURE 4.1.: Aperçu de Nile : le coeur et ses différentes librairies..

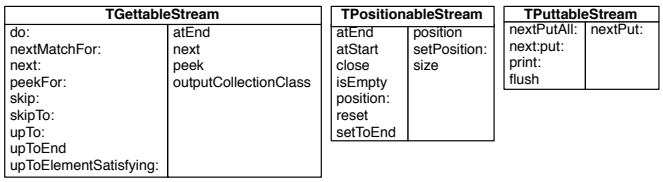

FIGURE 4.2.: Les traits du coeur de Nile.

TPositionableStream, TGettableStream et TPuttableStream. Ils sont montrés sur la figure 4.2.

TGettableStream. Le trait TGettableStream est conçu pour les streams utilisés pour lire n'importe quel type d'éléments. Le trait nécessite 4 méthodes : atEnd, next, peek et outputCollectionClass. La méthode peek retourne l'élément suivant sans modifier le stream tandis que next lit et retourne l'élément suivant et modifie le stream. La méthode TGettableStream>>outputCollectionClass est utilisée pour déterminer le type de collection à utiliser lorsque l'on retourne une collection déléments, comme avec next: et upTo:.

TPositionableStream. Le trait TPositionableStream permet la création de streams positionné de mnière absolue. Cela correspond au SequencedProtocol d'ANSI. La seule méthode requise est size et deux accéseurs pour une variable concernant la position. La vérification des bornes pour la méthode position: est implémentée dans le trait lui même : le parmètre doit etre entre zéro et la taille du stream. Comme les traits utilisés sont sans états, cela signifie que deux méthodes doivent etre implémentées : un accesseur pur, appelé setPosition:, et un véritable accesseur publique appelé position: qui vérifie la valeur de son paramètre.

TPuttableStream. Ce trait est le plus simple de la librairie Nile. Il propose nextPutAll:, next:put:,print: et flush et nécessite nextPut:. Par défaut, flush ne fait rien. Il est utilisé pour assurer que tout a bien été écrit. Les streams basés sur un *Buffer* devrait avoir leur propre implémentation.

4.3. Streams basés sur des collections

Afin de supporter le *streaming* sur les collections, un ensemble de traits dédiés ainsi que ce que est appelé des *trait factories* définissant leurs protocoles de création ont été implémentés. Notez que, en contraste avec la librairie des streams de Squeak

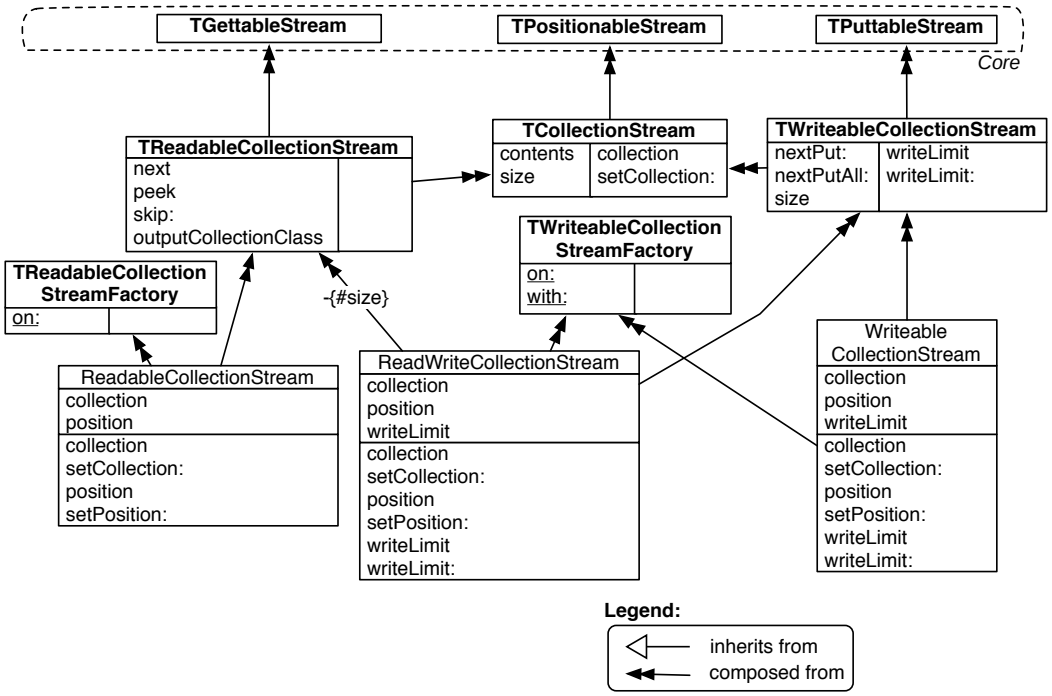

FIGURE 4.3.: La librairie des streams basés sur des collections.

et comme VisualWorks, l'impémentation de Nile fonctionne actuellement avec n'importe quelle collection *sequenceable*, et pas seulement avec Arrays et Strings.

4.3.1. Les traits

Les traits TCollectionStream, TReadableCollectionStream et TWriteableCollection-Stream implémentent les fonctions de bases des streams basés sur des collections (comme indiqué dans l'illustration 4.3 — Notez que dans celle-ci, les traits ont leurs noms en majuscule contrairement aux classes). Ils définissent toutes les méthodes nécéssaires requises par les traits du coeur et requièrent seulement 4 nouveaux accesseurs.

TCollectionStream. Ce trait est inspiré par le standard ANSI. Il est utilisé par chaque stream devant lire ou écrire au travers d'une collection. Il définit contents et size à l'aide de deux nouvelles méthodes : collection et setCollection:. La première doit retourner la collection interne et la seconde offre un *setter* pour cette collection. La méthode size retourne la taille de la collection.

TReadableCollectionStream. Le trait TReadableCollectionStream aide à la création de classes fonctionant sur des collections pouvant être lues. Il implémente les méthodes requises par TGettableStream : next, outputCollectionClass et peek. Il redéfinit également skip: pour des raisons de performances. La méthode requise TGettableStream>>atEnd est fournie par TPositionableStream et ne demande ainsi pas plus de travail.

TWriteableCollectionStream. Le trait TWriteableCollectionStream dépend d'une nouvelle variable d'instance accessible au travers de deux accesseurs writeLimit et writeLimit:. Cette variable permet à la collection interne d'être plus grande que le nombre déléments dans le stream. C'est une technique commune pour éviter la création d'une nouvelle collection à chaque fois qu'un objet est écrit sur le stream. La méthode TWriteableCollectionStream>>size retourne la valeur de writeLimit et nextPut: ajoute son paramètre à droite de la collection. Le trait réimplémenteégalement nextPutAll: pour des raisons de performance.

4.3.2. Classes

Les traits seuls ne sont pas suffisants pour créer une librairie. Les classes sont nécéssaires pour composer et créer de nouvelles instances. La hiérarchie de Squeak fournie trois classes pour les streams basés sur des collection : ReadStream, WriteStream et ReadWriteStream. L'implémentation de Nile possède des classes équivalentes avec des noms plus explicites : ReadableCollectionStream, WriteableCollectionStream et ReadWriteCollectionStream.

Ces classes ne font rien de plus que de déclarer l'utilisation de traits déjà définis, en déclarant quelques variables d'instance et implémentant les accesseurs requis.

La seule difficulté survient avec ReadWriteCollectionStream qui possède un conflit avec la méthode size. La méthode size est implémentée dans TReadableCollectionStream, héritée de TCollectionStream, et dansTWritebaleCollectionStream. La première implémentation reflète la taille de la collection tandis que la deuxième prend compte de la variable writeLimit et de l'implémentation particulère de TWriteableCollectionStream. C'est pourquoi ReadWriteCollectionStream doit utiliser l'implémentation de TWritebleCollectionStream. Pour faire cela, la classe supprime l'implémentation de size provenant de TReadableCollectionStream. Cette opération est visible sur l'illustration 4.3. [2]

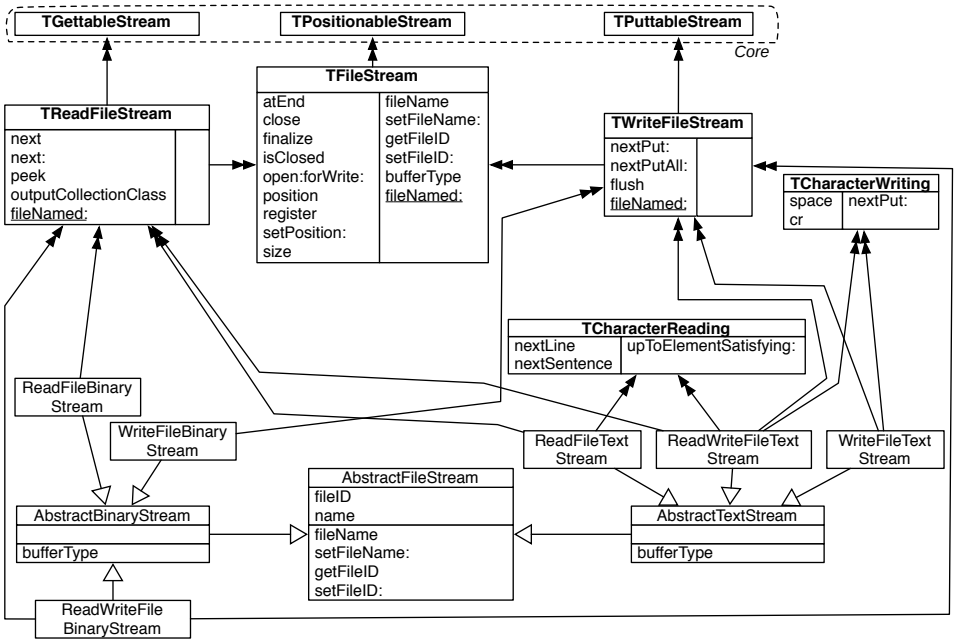

FIGURE 4.4.: Implémentations des streams basés sur des fichiers.

4.4. Streams basés sur des fichiers

Nile inclut une librairie de streams basés sur des fichiers, visible sur l'illustration 4.4. Comme pour tous les autres streams basés sur des fichiers, elle permet de travailler avec des fichiers binaires ou de textes, en supportant trois méthodes d'accès pour chaque cas (*read*, *write* et *readwrite*).

Chaque sorte d'accès au fichier est représentée par différentes classes : le developpeur doit explicitement choisir la classe en se basant sur ce qu'elle doit faire avec le fichier : lire, écrire ou les deux dans un mode texte ou binaire. De cette façon, l'utilisateur posséde seulement les méthodes qu'il peut envoyer dans l'interface du stream. Notez que cela est choix de design et n'a pas d'impact sur la manière de décomposer le comportement en traits.

Chaque stream basé sur un fichier devrait être positionable, c'est pourquoi le trait TPositionableStream est utilisé pas TFileStream. TFileStream est le trait commun pour tous les streams basés sur des fichiers. Il implémente les fonctionalités de base pour l'accès aux fichiers et nécéssite 4 accésseurs, une méthode bufferType et un

[2]Le modèle de traits donne la possibilité au composeur de supprimer une méthode grâce à l'opérateur moins (-).

constructeur fileNamed. La méthode privée bufferType est utilisée pour différencier les fichiers binaires des fichiers textes.

Les traits TReadFileStream et TWriteFileStream utilisent les traits TGettableStream et YPuttableStream provenant du coeur. Ils implémentent les méthodes requises de ces traits. Avoir implémenté les méthodes concernant lécriture et la lecture dans des traits séparés plutôt que dans des classes nous aident ici. De cette manière, nos streams basés sur des fichiers ne possèdent que les méthodes désirées et non toutes les méthodes contrairement à la hiérarchie de Squeak.

Tout en bas de l'illustration 4.4, nous définissons les classes abstraites AbstractFileStream, AbstractBinaryStream et AbstractTextStream afin de factoriser la définition des variables d'instance et des accésseurs. Ces classes abstraites permettent la définition de six classes concrétes sans plus de travail.

Les Streams basés sur des textes utilisent les traits CharacterReading et TCharacterWritting selon le type d'accès au fichier. Même si cela paraît simple, ces deux traits aident à définir des méthodes seulement là où elles sont nécéssaires.

4.5. Contibution

Le coeur de Nile possède une structure déjà riche et solide. Mon travail a consisté en l'utilisation de ces traits pour enrichir les variétés de streams disponibles dans Nile. Ce travail est décrit plus précisement ci dessous.

4.5.1. Améliorations apportées

De nouvelles classes de streams. Afin de rendre Nile interchangeable avec la librairie actuelle des streams dans pharo, j'ai construis un grand nombre d'équivalents aux différents streams existants. Ces classes sont visibles dans le tableau 4.1. Chacune d'elle a été construite en faisant abstraction de son implémentation dans une hiérarchie actuelle limitée par l'héritage simple. Une grande partie de mon travail a été réalisée en consultant l'avis de Damien Cassou, principal développeur de Nile. Ainsi, pour la plupart des classes non présentes dans Nile j'ai analysé son fonctionnement, réfléchi à sa place dans Nile puis implémenter une nouvelle version.

De nouveaux traits. Certaines de ces classes ont également nécéssité l'écriture de nouveaux traits pour des méthodes n'étant pas encore présentes dans Nile. Par exemple, la hiérarchie des streams dispose de FileStreams, effectuant des opérations sur un fichier, ainsi que des sortes de PseudoFileStreams (RWBinaryOrTextStream, MultiByteBinaryOrTextStream), proposant les mêmes opérations qu' un FileStream mais avec des données en mémoire. Un intérêt est par exemple d'utiliser des fonctionnalités d'exportation et d"importation de code (fileOut, fileIn) en mémoire plutôt que sur des fichiers. Cependant, dans les deux cas ces méthodes sont communes aux

Table 4.1.: Nouvelles classes implémentées dans Nile

Classes	Description
NSDecoderInflateStream	Utilisée pour décoder des données compréssées (Zip, GZip, ZLib).
NSDecoderDeflateStream	Utilisée pour encoder des données non compréssées (aux formats GZip, Zip, ZLib).
NSDataStream	Utilisée pour la sérialisation d'objets.
NSCompressedSourceStream	Utilisée pour lire-écrire sur un fichier compréssé segment pat segment.
NSRBScanner	Découpe un String en Token.
NSTranscripter	Stream simple avec affichage utilisé pour obtenir un évaluateur d'urgence.
NSMailAdressTokenizer	Découpe une adresse en plusieurs Token.
NSLimitedWriteStream	Un WriteStream avec une limite sur la taille de la collection.
NSRWBinaryOrTextStream	Simulation d'un FileStream où les données sont stockées en mémoire.
NSMultiByteBinaryOrTextStream	Comme NSRWBinaryOrTextStream mais lisant et écrivant par l'intermédiaire d'un TextConverter.
NSTextStream	Utilisée pour écrire du texte avec du style.

deux types de streams. Des traits regroupant ces méthodes resoudent le problème de méthodes définis trop haut dans la hierarchie actuelle.

Des tests pour chaque nouvelle classe. Chaque classe implémentée dans Nile a été testée. Ces tests ont pu être écrits à l'aide de traits de tests généraux que Damien Cassou avait déjà codé. Ces traits de tests sont en correspondance avec les traits de Nile. Une classe de tests peut ainsi utiliser NSTPuttableStreamTest ou NSTGettableStreamTest selon les traits de Nile correspondant à la classe testée (NSTPuttableStream ou NSTGettableStream).

4.5.2. Resultats obtenus

Des interfaces claires. Le tableau 4.2 présente les traits et les classes que j'ai utilisés pour la définition des différents types de streams. Ces nouvelles classes offrent toute une interface simple et claire. En effet, contrairement à la librairie actuelle, chaque méthode présente dans une classe a une raison bien précise d'y être. Par exemple, là où les méthodes concernant l'écriture ou la lecture de bytes étaient définies dans une superclasse commune (PositionableStream), les traits de Nile m'ont

permis de ne pas polluer les classes de streams plus simple ne nécéssitant pas ce type de méthodes (comme NSTextStream).

Un ensemble cohérent. L'implémentation de ces différentes classes a néssecité pour chacune d'elle une phase d'analyse afin de comprendre leur utilité dans le système. Une des difficultés de cette étape est venue du manque d'unité de la librairie des streams. En effet, dans une hiérarchie limitée par l'héritage simple, certaines classes semblent avoir une place assez aléatoire ne donnant pas d'indications sur leur sens propre. Extraire des comportement de base dans des traits (NSPuttableStream, NSGettableStream) permet aux arbres d'héritage de donner des sens plus fins aux classes (basées sur des collections ou des fichiers par exemple). Il en résulte un ensemble plus cohérent.

Des classes encore manquantes. Il reste encore quelques classes de la hiérarchie actuelle pour lesquelles je n'ai pas eu le temps d'implementer une version dans Nile :

- ObjectSocketStream (utilisée pour lire-écrire au travers d'une Socket).
- JPEGReadStream (utilisée pour lire des données au format JPEG).
- ZipEncoder (utilisée par ZipWriteStream, sous-classe de DeflateStream, pour encoder les données).

Table 4.2.: Structure des nouvelles classes implémentées dans Nile

Classes	**Superclasse**	**Traits utilisés**
NSDecoderInflateStream	NSDecoder	
NSDecoderDeflateStream	NSDecoder	
NSDataStream	Object	NSTGettableStream + NSTPuttableStream
NSCompressedSourceStream	Object	NSTGettablePositionableStream + NSTPuttablePositionableStream
NSRBScanner	NSDecoder	
NSTranscripter	NSCollectionStream	NSTCharacterWriting
NSMailAdressTokenizer	Object	NSTGettableStream
NSLimitedWriteStream	Object	NSTPuttablePositionableStream
NSRWBinaryOrTextStream	NSCollectionStream	NSTByteWriting +NSTByteReading + NSTFileIn + NSTFileOut + NSTCharacterWriting
NSMultiByteBinaryOrTextStream	NSCollectionStream	NSTFileIn + NSTFileOut + NSTCharacterWriting + NSTByteWriting + NSTByteReading
NSTextStream	Object	NSTPuttablePositionableStream + NSTCharacterWriting

5. Réalisation et organisation

5.1. A propos des traits de tests

5.1.1. Organisation du travail

Mon travail concernant la couverture de la librairie des Collection s'est naturéllement découpé en fonction des classes à tester. J'ai ainsi pris les différents types de collections un à un et suivi la démarche suivante :

- Utiliser les traits déjà éxistants
- En cas de problèmes (traits trop spécifiques) modifier les traits.
- Regarder les méthodes non testées à l'aide d'un outil d'analyse.
- Selon les méthodes non testées, créer de nouveaux traits de tests.

Tout au long de ce travail, Stéphane Ducasse et Damien Pollet m'ont orienté vers les problèmes existants, les points manquants et les questions à se poser mais m'ont laissé une liberté totale dans la solution à fournir. J'ai ainsi régulièrement rendu compte de mes choix et de ce qu'il me semblait judicieux de faire.

5.1.2. Difficultés rencontrés

Le challenge principal des traits est qu'il soient réutilisables. Les principaux problèmes rencontrés sont liés à la réutilisation des traits de tests pour différentes classes de tests :

Traits trop gros. Tout au long des différentes classes testées je n'ai rencontré aucun inconvénient à ce que les traits soient le plus petit possible. A l'inverse, plusieurs probèmes sont apparus avec l'utilisation de traits regroupant trop de méthodes :

- Un certain type de collection peut ne pas définir toutes les méthodes testées dans un trait trop gros. Dans ce cas il est toujours possible de spécifier les méthodes non souhaitées lors de la définition de la classe.

  Exemple :

TestCase subclass: #LinkedListTest
uses: TAddTest - **{#testTAddWithOccurences. #testTAddTwice. #testTWriteTwice}**
...

Une LinkedList est une collection contenant des objets de type Link pointant vers un autre objet. Insérer deux obets identiques provoque une boucle infinie. Il faut ainsi faire le tri des tests non désirés lors de la définition de la classe de tests.

- Beaucoup de méthodes dans un trait de test peut entrainer beaucoup de requirements (methodes requises pour le trait). Dans ce cas, celui-ci devient plus pénible et plus dificile à utiliser pour une classe de tests. Un indice pouvant montrer qu'un trait de test a une bonne granularité est que les méthodes de test necessitent un même requirement.

  Exemple :

  La classe SequenceableCollection dispose d'une catégorie *accessing*. En essayant de tester toutes ces méthodes dans un même trait TAccessingTest on se rend vite compte que différents groupes se forment selon les requirements nécessaires. Trois traits différents peuvent apparaître :

  – TIndexAccessTest pour les méthodes accédant à un index à l'aide d'un élément
  – TElementAccessTest pour les méthodes accédant à un élément à l'aide d'un index
  – TSubcollectionAccessTest pour les méthodes accédant à des sous-parties de collections

Tests trop spécifiques à un type d'éléments. Chaque type de collection peut contenir différents types déléments : Link, Integer, Float, Character, Collection ou Object de manière générale. Dans un trait testant des méthodes communes à tout type de collection, les opérations éffectuées sur les éléments doivent être assez générales pour pouvoir être utilisé par toutes les classes de tests.

Exemple :

Une première version de TIterateTest testait la méthode select: à l'aide de métodes spécifiques à des Number :

```
testSelect
| result |
result := self collectionWithoutOddNumbers select: [ :element | element odd].
```

```
self assert: result size = 0.
```

Bien que certaines classes de tests peuvent utiliser des collections avec des éléments de type Number (Array, OrderedCollection, Set), d'autres ont un type précis d'éléments contenus dans leur collection (String, LinkedList). Une solution est d'utiliser des méthodes définies pour Object :

```
testSelect
| result |
result := self collectionWithoutNilElements collect: [ :element | element isNil].
self assert: result size = 0.
```

Tests trop spécifiques à un type de collections. Chaque type de collection a un comportement spécifique : pouvoir ou non contenir des éléments égaux, être ou non ordonnée, être ou non triée, être ou non extensible, être ou non indexée. Un trait testant des méthodes communes à tout type de collection (TIterateTest par exemple) doit rester assez général pour pouvoir être utilisé par toute les classes de tests.

Exemple :

TIndexAccessTest teste des méthodes communes aux collections indexées (indexOf: indexOf:startingAt:). Certaines de ces méthodes peuvent avoir des comportements différents selon qu'il y ait une ou plusieurs fois un élément dans la collection (lastIndexOf:). En testant ces différents cas dans un même trait un problème se pose pour des collections n'acceptant pas d'éléments égaux : celles-ci devraient tout de même être testées avec le comportement basique de ces méthodes. Une bonne solution semble être de séparer ces différents comportements en deux traits :

- TIndexAccesTest pour le comportement de base.
- TIndexAccessForMultiplinessTest pour le comportement spécifique.

5.2. A propos de Nile

5.2.1. Organisation du travail

Un des principaux objectifs était de repérer les classes non présentes dans Nile. Après m'être familiarisé avec Nile, une série détapes s'est vite détachée du travail á réaliser pour chaque classe :

- Analyser le comportement de la classe, comprendre comment l'utiliser. Comment elle est utilisée dans le système ?
- Tester et appréhender ce comportement en écrivant des tests. Ceux-ci pourront être réutilisés pour la version Nile de la classe.

- Discuter (souvent avec Damien Cassou) de la place de cette classe dans Nile.
- Implémenter la classe dans Nile.
- Si l'occasion se présente, créer de nouveux traits.
- Tester la classe.

La plupart de mes choix on été discutés avec Damien Cassou afin d'obtenir de meilleurs solutions et de ne pas me perdre dans mes idées de départ fortement influancées par la librairie actuelle (limitée par l'héritage simple).

5.2.2. Difficultés rencontrés

Faire abstraction de la hiérarchie existante L'héritage simple de la hiérarchie actuelle des Stream propose une division simpliste entre ReadStream et WriteStream. L'utilisation de traits permet la définition de types beaucoup plus riches (sans avoir á dupliquer le code) et ainsi de donner un sens beaucoup plus fin à chaque Stream. Une des difficultés est de se détacher de la hiérarchie actuelle et de réfléchir au sens à donner à chaque classe.

Exemple :

Dans la hiérarchie actuelle, les classes InflateStream et DeflateStream sont codées de la manière suivante :

- InflateStream est un ReadStream qui lit à partir de données compréssées
- DeflateStream est un WriteStream qui écrit et compresse des données

Dans Nile, un système de Decoder a été implémenté permettant de voir les choses de la manère suivante :

- InflateStream à partir de données compressées *décode* vers des données non compressées
- DeflateStream à partir de données non compressées *décode* vers des données compressées

Ainsi, le passage d'une hiérarche à l'autre de ces classes devient une réelle adaptation du code existant à la structure de Nile (et non une simple copie des méthodes).

Choisir entre classe et trait. Une de mes premières erreurs a été de vouloir systématiquement créer de nouveaux traits pour chaque nouvelle classe codée. Chaque type de Stream possède un comportement spécifique à lui même. Créer un trait pour des méthodes correspondant à ce comportement ne paraît pas judicieux :

- Ces traits ne resserviront probablement à aucune autre classe.
- Ces comportements spécifiques nécessitent souvent beaucoup d'informations propres à la classe. Un trait nécessitera donc beaucoup de requirements, le rendant pénible et difficile à écrire et à utiliser.
- Ces traits pollueront inutilement la hierarchie.

Ainsi, il ne faut créer un trait que dans le cas où il est vraiment nécessaire, où il résoud un possible problème de duplication de code dans la hiérarchie actuelle. Bien souvent, les traits nécessaires étaient déjà codés dans Nile.

5.3. Environnement et méthodes de travail

Concertations régulières. J'ai pu rendre compte régulièrement (en général toute les semaines) de l'avancement des mes travaux au cours d'entrevues avec Stéphane Ducasse. Celles-ci nous ont permis à chaque fois de faire le point sur le travail réalisé, de discuter des problèmes rencontrés, de réorienter mon travail et d'établir éventuellement une liste de nouvelles tâches à effectuer.

Dicussions avec la communauté. Mes différents travaux ont permis de mettre en évidence plusieurs problèmes existants dans le langage (liés notament aux hiérarchies des collections et des streams). Ceux-ci ont donné lieu à plusieurs discussions dans la mailing-list de Pharo. J'ai ainsi pu recueillir l'avis d'un grand nombre de personnes et décider avec eux de certains choix.

Gestionaire de versions. Pharo dispose de son propre système de gestion de versions intégré : *Monticello*. Par l'intermédiaire de celui-ci, j'ai pu sauvegarder chaque nouvelle modification apportée dans un répertoire. Une fois un résultat stable obtenu (par exemple plusieurs classes de tests correctement couvertes), Stéphane pouvait intégrer mes changements au langage.

5.4. Organisation du temps

Le premier mois a été consacré à mon adaptation au domaine. Je me suis familiarisé avec Smalltalk et plus spécifiquement Pharo (langage utilisé pour touts mes travaux) à l'aide de plusieurs exercices et de plusieurs livres (notament Squeak by example).

J'ai également pris le temps de m'intérésser aux traits et d'étudier les travaux ayant déjà été réalisés.

Durant les deux mois suivants j'ai travaillé sur les traits de tests pour la librairie des Collection. Après avoir analysé le travail déjà éffectué, j'ai enrichis les traits et classes de tests en résolvant les différents problèmes au fur et à mesure.

Je me suis ensuite penché sur Nile pendant les deux mois suivants. J'ai de la même façon étudié le projet dans une première étape puis compléter celui-ci. J'ai également continué à renforcer mes traits de tests pour les collections en testant certaines classes selon les besoins de la communauté Pharo.

Le dernier mois a été une période mixte pendant laquelle j'ai eu l'occasion de réaliser mon rapport, de compléter mes différents travaux ou d'apprendre de nouvelles choses en vue de mes futures tâches.

6. Bilan du stage

6.1. Réponses aux questions initiales

De manière générale, le stage avait pour objectif d'évaluer les traits dans la réalisation de larges projets de développement et de répondre à une série de questions énoncées au début de ce rapport. Je fais part dans cette section d'avis obtenus au travers de mon travail.

Granularité des traits. Suite à la réalisation de ces travaux, il paraît clair que les traits gagnent à être le plus fin possible. De manière générale, plus le groupe de méthodes est petit, moins il y a de chances que certaines d'entre elles ne conviennent pas pour une certaine classe. Cependant, l'utilisation de traits fins augmente leur nombres et pollue en quelques sortes un peu plus l'interface.

Proportion de code réutilisé. Les traits apportent une grande réutilisabilité grâce à leur granularité fine. C'est d'ailleurs là un de leurs principaux avantages. Ils peuvent être réutilisés de manière orthogonale aux librairies. Pour cela, les requirements sont les morceaux abstraits permettant de faire le lien avec les différentes librairies. Chaque librairie les implémente ensuite selon leur comportement spécifique. Par exemple, les classes Dictionary et Array n'accèdent pas de la même façon a leurs élements (grâce à une clé quelconque ou à index numérique). TPutTest est pourtant réutilisable par les deux classes de tests par l'intermédiaire du requirement index.

Correction des problèmes éxistants dans les hiérarchies actuelles. Contrairement à l'héritage qui impose toute la structure de la superclasse à ses sous-classes, donnant lieu soit à de la duplication de code soit à des méthodes définies trop haut, les traits sont directement utlisables par les classes qui les nécessitent. Dans la hiérarchie des collections, certaines classes peuvent être extensibles (OrderedCollection) ou non (Array). Selon, elle donnent du sens aux méthodes de type add:. Cependant, add: est défini dans Collection pour éviter la copie de code. Là où l'héritage aurait apporté les problèmes habituels, le trait TAddTest permet aux classes de tester ces méthodes seulement si elles font du sens pour elles.

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.

Je pense 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. Dans le cas de traits de tests, les traits de tests peuvent se former naturellement selon les méthodes communes à la hiérarchie.

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

6.2. Bilan personnel

Enrichissement des connaissances. Ce stage a été l'occasion pour moi d'appréhender de nouveaux outils et d'acquérir de nouvelles connaissances. J'ai ainsi pu découvrir Smalltalk et certains bons principes de programmation orienté objets qui y sont associés. Ces derniers me seront utiles quelque soit les technologies utilisées durant ma future vie professionnelle. Plus que les particularités du domaine dans lequel le stage a eu lieu, j'ai également pu m'adapter aux outils utilisés par l'équipe. J'ai par exemple utilisé LaTeXpour la réalisation de ce rapport ainsi que le système de gestion de versions Subversion.

Un contexte intéressant. Ces six mois passés dans l'équipe RMOD m'ont permis de confirmer mon attirance pour le travail de recherche. En effet, durant cette période j'ai pu apprécier différents points :

- Apprendre constamment de nouvelles notions, techniques, technologies.
- Prendre le temps de réfléchir au sens que l'on donne à ce que l'on fait.
- Dans mon cas, appartenir à une communauté construite autour d'un projet.

De plus, le domaine de travail de l'équipe m'a particulièrement plu. Pharo par exemple, est un projet pour lequel je continurai sans doute à contribuer .

Une passerelle vers de futurs travaux. Suite à ce stage, je suis embauché pour deux ans dans l'équipe RMOD en tant qu'ingénieur de recherche. Ces deux années seront l'occasion pour moi de m'intéresser à d'autres domaines et de rencontrer de nouvelles personnes traivaillant autour de la communauté Pharo.

7. 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 rapport de stage est une évaluation des traits dans des contextes différents. D'un coté, ils sont utilisés comme outil de test dans une hiérarchie basée sur l'héritage simple. Nile quand à lui est la première réalisation de projet pensé pour les traits. Dans les deux cas, la structure des traits a été construite en se basant sur une spécification (notament celle de *ANSI Smalltalk*) 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. Durant ce stage, l'utilisation de traits m'a semblé apporter un bénéfice certain. Malgré une mise en place plus difficile, plus longue qu'une classe, où il faut réfléchir aux méthodes à grouper ensemble, aux requirements nécéssaires, j'ai apprécié le gain de temps, la simplicité d'utilisation, la simplicité du design obtenu, que ce soit pour les traits de tests ou pour Nile.

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. Ces travaux seront utilisés pour vérifier la validité et l'intérêt des modèles de traits futurs.

Travaux futurs sur les collections. Les traits de tests que j'ai obtenu semble être bien définis et m'ont permis de couvrir facilement une grande partie de la librairie des collections. Ils pourront servir plus tard à tester de la même manière n'importe quelle autre collection. Dans le prolongement des ces tests, le redéfinition de la librairie à base de traits pourra réutiliser les tests ainsi que les découpages formés par les traits de tests.

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, ces travaux on permis de mieux évaluer les traits dans le cadre de projets conséquents. Grâce à ceux-ci, 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.

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.

Table 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.

Table 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 morceau 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 à 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.