Talented Streams

Objects composed from Features

Bachelor's thesis
at the
Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch/

by
Manuel Leuenberger
February 2013

led by
Prof. Dr. Oscar Nierstrasz
Dr. Jorge Ressia

Abstract

A stream is a data structure to process static resources like files dynamically and efficiently. The variation of processable resources puts numerous requirements on an implementation of the stream data structure. Furthermore, a lot of features need to be provided in multiple combinations *e.g.,* buffering is useful for file streams and socket streams as well. A solution that tries to model all these combinations of features on the class level (*i.e.,* using traits or going the traditional class based framework approach) leads to an explosion in the number of classes or a significant amount of code duplication, which hinders any framework's maintainability. We find these problems in the standard Pharo implementation of streams and Xtreams. To circumvent a class explosion and code duplication, we model composable features instead of combinations of features. Each feature is modeled as a *talent* and can be composed with any other feature. We introduce scoped talents which allow the use of subjective behavior and guarantee an easy composition of features and decomposition as well. Our approach allows us to decouple features from each other and neither leads to a class explosion, nor to a significant amount of code duplication. The resulting framework is called Talented Streams. We also present a way to efficiently work with talents by using classes as factories. This approach allows us to reuse the development tools, which were designed to work with classes, for the development of talents.

Contents

1 Introduction **1**
1.1 Streams In a Nutshell . . . 1
1.2 Stream Features . . . 1
1.3 Traditional Model . . . 3
1.4 Trait Based Model . . . 4
1.5 Talent Based Model . . . 4
1.6 Outline . . . 5

2 Existing Streams Frameworks **5**
2.1 Detecting a Class Explosion . . . 5
2.2 Detecting Code Duplication . . . 6
2.3 Analyzing Pharo Streams . . . 6
2.4 Analyzing Xtreams . . . 10

3 Talented Streams **11**
3.1 Modeling Features . . . 11
3.2 Modeling Composability . . . 11
3.3 Scoped Talents . . . 13
3.4 Scoped Talents in Action . . . 14
3.5 Evolving Scoped Talents to Stream Features . . . 16
3.6 Talent Templates . . . 19

4 Validation **20**
4.1 Motivating Example . . . 20
4.2 Framework Analysis . . . 21
4.3 Evaluation . . . 22

5 Conclusion **22**

6 Acknowledgments **23**

List of Figures

1	UML class diagram of the basic Pharo streams.	7
2	Moose class blueprint of the `PositionableStream` class.	8
3	Moose side-by-side duplication view of Pharo streams.	9
4	Moose side-by-side duplication view of Xtreams.	10
5	Message graph of the scoped talent in Listing 5	15
6	Delegation of the message `#read` from the stream to the scoped talent it acquired in Listing 5. The arrows form the execution path.	16
7	Composition of features by adapting a scoped talent as in Listing 6.	18

List of Listings

1	The first result times of our fictional marathon.	1
2	Reading the first result of the marathon stored in the file in Listing 1 using Pharo streams.	3
3	Reading the first result of the marathon stored in the file in Listing 1 using Xtreams.	3
4	Aliasing a method invalidates all message sends to the aliased implementation using the old selector.	12
5	Using scoped talents to convert bytes to characters.	14
6	Using features modeled as scoped talents to convert bytes to characters.	17
7	Source of `TSConvertingReader>>read`	17
8	Reading characters from a file using Talented Streams.	19
9	Reading the first result of the marathon stored in the file in Listing 1 using Talented Streams.	20

1 Introduction

1.1 Streams In a Nutshell

The stream data structure is used to read and write data sequentially. A stream can process data that is either not available completely when accessing it the first time or exceeds some limitations of the processing unit. For example, we can use a stream to log the names and times of marathon participants in a file as soon as they finish. For the reverse operation *i.e.,* reading the runner's results of a marathon, we can use a stream as well. This way we can create a live ticker for the marathon, so people interested in this event can follow it. After the marathon, we may want to know the fastest marathon participant. We could request the full set of results, but if we request a stream over these results, we can take only what we need. Using a stream, instead of loading the whole file into the memory, saves resources by decreasing the time needed to process the data and reducing the amount of allocated memory. But these small examples form only a small subset of possible applications of streams: We may want to provide a live video stream in various resolutions for the marathon, including a multi-lingual audio commentary. Since there exist so many variations of streams *e.g.,* video, audio, binary, character based, buffered, limited, operating on files, sockets, pipes *etc.*, it is hard to create a model for it, in which all these variations fit.

```
Willy Ackermann, 2:07:29
Peter Ulrich, 2:15:34
Ursula Steiner, 2:16:01
Konrad Meyer, 2:17:51
Marco Stopfer, 2:17:52
...
```

Listing 1: The first result times of our fictional marathon.

To compare existing stream frameworks and our own framework, we will use a simple scenario as a motivating example, built around a fictional marathon. Our marathon took place in the past and had thousands of contestants. Their names and finishing times are stored in a file as in Listing 1. Each line contains the name of the contestant and their respective time. The lines are ordered ascending in the duration a contestant needed to complete the marathon. Our goal is to find out who won the marathon in what time. Or in other words, we want to read the first line of a text file. If we use a stream to do this, we can solve this task very efficiently: We can load the file byte by byte into the memory and stop as soon as we detect a line break character.

1.2 Stream Features

If we speak of a feature, we mean a cohesive unit of behavior and state, a fine-grained module. Each feature satisfies a functional requirement *i.e.,* buffering.

In the following list, we collected features that a stream framework should implement. Some of these features are essential *e.g.,* reading and writing, others may not be essential, but are still desirable *e.g.,* selecting. We see that the functional requirements to a stream framework are quite numerous, yet we have left out non-functional requirements like performance. The number of features itself is not a problem, the problem is that we need a stream for every possible combination of these features. If we have n features, we have 2^n possible combinations of which each should be instantiable.

reading
: A stream should expose a simple interface for reading data, without the need for the user to be aware of the stream's underlying structure.

writing
: A stream should expose a simple interface for writing data, without the need for the user to be aware of the stream's underlying structure.

input
: A stream should be able to read from various inputs *e.g.,* files, sockets, pipes, collections.

output
: A stream should be able to write to various outputs *e.g.,* files, sockets, pipes, collections.

buffering
: A stream should be able to buffer read and written data in order to reduce system calls and thereby improve the performance.

caching
: A stream should be able to cache read and written data in order to reduce system calls and thereby improve the performance.

positioning
: A stream should be positionable if it operates on a positionable input or output *e.g.,* files or sequenceable collections.

converting
: A stream should be able to convert the data it is processing.

selecting
: A stream should be able to select or reject the data it is processing.

collecting
: Data can be collected to form a new datum.

1.3 Traditional Model

If we map each possible combination of features to a dedicated class, we end up with an explosion in the number of classes. Therefore a class-based model can either provide only a limited number of stream variations, or it must provide some sort of composition of features. Throughout this paper, we will use the term *Pharo streams* to describe the set of classes which inherit from the `Stream` class in Pharo Smalltalk [1], including `Stream` itself.

```
stream := FileStream fileNamed: 'results.txt'.
stream upTo: Character lf.
```

Listing 2: Reading the first result of the marathon stored in the file in Listing 1 using Pharo streams.

The Pharo streams only provide a limited set of streams *e.g.,* a buffered writing stream is missing in the category `Collections-Streams`, though it exists in the `Zinc-HTTP-Streaming` category. The same features are implemented in different classes, since Pharo streams lack a way to compose features, which results in code duplication. As we will see in Section 1.4, a trait based model is able to minimize code duplication, but it can not avoid a class explosion. In Listing 2 we show how we can solve our motivating problem, getting the winner of a marathon by reading the first line of a text file, with Pharo streams. Admittedly, this solution looks very simple. This is because our motivating problem is a fairly common scenario, and Pharo streams have been developed and improved over years to provide easy solutions for common problems.

```
1 stream := (XTIOHandle forFile: 'results.txt' forWrite: false) reading.
2 ((stream collecting: [ :read | read asCharacter ])
3     contentsSpecies: String;
4     ending: Character lf) rest.
```

Listing 3: Reading the first result of the marathon stored in the file in Listing 1 using Xtreams.

Xtreams [2] has a more robust and more extensible architecture than the standard Pharo streams. Instead of rendering each combination of features into a class, it models each feature in a class, and it provides a simple interface to compose them. Xtreams separates the Core which implements the features reading, writing and to some extent positioning and buffering, Terminals which implement the features input and output, Transforms which implement the features converting, selecting and more, Substreams which implement the composition of features by wrapping a stream within another stream. In Listing 3 we show how we can solve our motivating problem with Xtreams. It looks a little bit more complex than the one with Pharo streams, but it shows the power of Xtreams, which is the composition of features. By sending `#reading` on line 1 to an `XTIOHandle` we create a basic file read stream, by sending `#ending:` on line

[1] http://www.pharo-project.org/home
[2] http://code.google.com/p/xtreams/

4 to this stream, we create substream that is ends with the first occurrence of a line break. By sending `#rest` we collect the contents of the stream. The problem that arises here, is the duplication of the input and output features in the `XTIOHandle`. These features are implemented in the class `FileStream` too, which is part of every Pharo environment. And the features buffering and positioning are only partially composable, which produces some code duplication.

1.4 Trait Based Model

Class based frameworks mostly reuse functionality through inheritance. Traits [10, 11] model behavior in a more abstract way, we can compose multiple traits with each other and reuse the same traits in different classes, which allows us to create more complex architectures *e.g.,* multiple inheritance or collections [1]. Cassou *et al.* have developed the core of a streams framework with traits, called Nile [2]. They have shown that traits can significantly reduce code duplication and improve the overall architecture, by mimicking multiple inheritance. But since traits target classes, the number of needed classes cannot be reduced: Suppose each feature is implemented as a trait. If we want a stream for every possible combination of features, there still needs to be a unique class for each combination. Each such class would use a composition of the traits implementing the respective features. But since the amount of classes is still dependent of the amount of possible combinations of features, the problem of a class explosion remains. Traits by themselves are not a solution to the problem of a class explosion, only a runtime composition can avoid it.

1.5 Talent Based Model

Ressia *et al.* describe talents as "dynamically composable units of reuse" [9]. A talent is really just a talent, a kind of special ability some have and some do not. For example, one person may have the talent to play football [3] like Lionel Messi, while another person plays football like a slice of toast and a third person does not even know what football is or mistakes it to be handegg. In OOP [4], talents define state and behavior that can be acquired by an object. If we have an object, we can let it acquire or lose a talent at *runtime*. This makes them different from traits which are used at *compile-time*. Using traits for the composition of features demands that we have to create a class, even if we just need a single instance with these features. Multiple talents can be composed to a new talent and their definitions *i.e.,* their behavior and state, can be edited by adding, altering or removing parts of it.

Talents allow us to model each stream feature as a fine-grained entity, and we are completely free how we compose them into a useful object. Talents allow us to compose multiple features by merging their respective sets of methods and states. This makes it necessary to resolve possible naming conflicts manually through aliasing or exclusion *e.g.,* if both composed talents define a method

[3] Hereby we mean real football and not US handegg.

[4] object-oriented programming

named `#read`. The more features we define the higher the probability that such conflicts arise. To reduce the risk of such conflicts, we contextualize the definition of methods and states within scopes. We use an extension of talents, which we call scoped talents, allowing us to specify a scope within which a certain feature should be defined. Scoped talents allow us to define multiple states and methods in different scopes. By means of dynamic scopes we can compose multiple features easily, even if some of them implement a method with the same selector. Since we decoupled features from each other's implementation we do not have a significant amount of code duplication, and since we compose our stream from features at runtime, we neither have a class explosion.

1.6 Outline

In Section 2 we try to identify the cause of code duplication and a class explosion, before we present our approach that avoids these problems in Section 3. In Section 4 we validate that our approach solves the identified problem. We conclude in Section 5 by rating our solution and discussing possible improvements and applications.

2 Existing Streams Frameworks

In this section we analyze two existing streams frameworks, Pharo streams and Xtreams, to see how much and why they suffer under the following problems:

class explosion
: If we have n features, we have 2^n possible combinations thereof, of which each should be instantiable. If each combination is modeled as a class, the number of classes explodes with the rising number of implemented features.

code duplication
: If we do not decouple features from each other, it complicates their reuse and we may have to reimplement them in various places. This results in code duplication and creates hidden dependencies between modules.

We use a Moose 4.6 release image [5] for the analysis. Moose [5] is a software and data analysis tool. It allows one to visualize dependencies between modules *e.g.,* classes or methods, using various metrics *e.g.,* number of methods or lines of code, by generating a meta model for the software under analysis.

2.1 Detecting a Class Explosion

We cannot identify a class explosion by the number of classes a framework consists of. It is not the number of classes that qualifies a class explosion, it is an architecture that models combinations of features in classes. The easiest way

[5] http://www.moosetechnology.org/download/4.6

to detect a class explosion is by analyzing how a framework can be extended with the introduction of a new feature. If such an extension cannot be achieved by simply adding a new module *e.g.,* in the form of a class, but the number of new classes needed to use this feature rises with each extension, we speak of a class explosion.

2.2 Detecting Code Duplication

We use SmallDude [6], a part of Moose, to detect and visualize code duplication. SmallDude allows to parametrize the detection algorithm, we used the default configuration. With this configuration, only code chunks with at least ten cloned lines are considered duplicated.

2.3 Analyzing Pharo Streams

2.3.1 Motivating Example

In Listing 2 we solved our motivating problem using the class `FileStream`, which unifies the features reading, writing, input, output and even more features in a monolith. This means that is it very hard to reuse these features, since they are tightly coupled with each other. There are the standard classes `FileStream` , `Socket` and `SocketStream` that wrap the system access for files and sockets. Obviously, a class `File` is missing, which would wrap the file system, similar to the way pursued with the `Socket` class.

2.3.2 Framework Analysis

Black *et al.* [1] have already pointed out some shortcomings in the Pharo streams and refactored them by using traits, although they concentrated on the standard collections. Shortly summarized, these findings are the following: The class `ReadWriteStream` duplicates the reading feature found in `ReadStream` by only subclassing `WriteStream`. `PositionableStream` implements a lot of methods that are not in any way related to positioning. We will now try to replicate these findings and we think that the Pharo streams deserve a deeper analysis which may bring other shortcomings to the surface.

There are 49 classes that inherit from the basic `Stream` class. These classes are spread over multiple categories, while only the base streams reside in the `Collections-Streams` category. `CrLfFileStream` and `TranscriptStream` are deprecated and `MockSocketStream` is only used for testing. This leaves 47 classes that can be considered to be concrete streams. The number is not of a size that would flood our eyes with tears, but if we look at the `Zinc-HTTP-Streaming` category, we see that new stream classes are introduced. They wrap a basic `ReadStream` or `WriteStream`, because the wrapped streams could not be configured to provide the needed features *e.g.,* there is no standard writing stream that provides buffering. The class `SocketStream` does *not* inherit from `Stream`.

[6] http://www.moosetechnology.org/tools/smalldude

It provides the same interface, but *e.g.,* buffering is implemented while it would already be implemented in `ReadStream`.

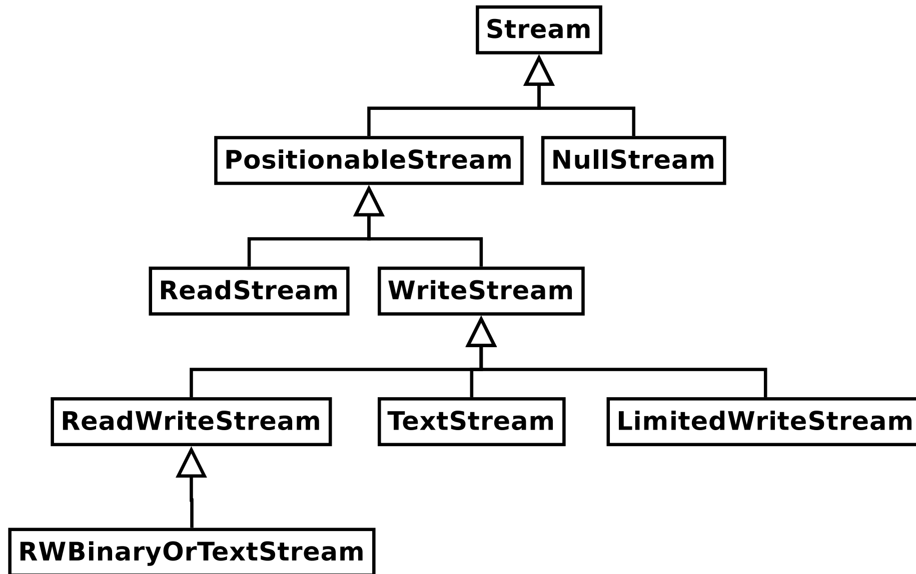

Figure 1: UML class diagram of the basic Pharo streams.

As we see in Figure 1, the architecture of the core of Pharo streams is extremely unbalanced. `ReadWriteStream` only inherits from `WriteStream` and not from `ReadStream`. If we inspect how the reading feature is reintroduced, we see that it is just a bunch of duplicated code. But our OOP loving heart cannot come to a rest now, because if we follow the inheritance tree a bit deeper, we see that `RWBinaryOrTextStream` does not inherit from `TextStream`. Also `RWBinaryOrTextStream` is a strange name, but we see symptoms of a class explosion: Classes are hard to name, since they model a hard-wired combination of multiple features in one single class.

And here we have covered up one big issue with single inheritance: Looking from the viewpoint of simplicity, single inheritance seems like a good idea, but to maximize the reusability of functionality, multiple inheritance is necessary. But as practice has shown, the price to pay for using multiple inheritance *i.e.,* the increased complexity, is just too high. In other words, quoting Cook: “Multiple inheritance is good, but there is no good way to do it.” [3] With traits, we are able to mimic multiple inheritance, but the composition of traits is intended to produce classes. So even if we have a class that uses all traits we need, except for one, we have to create a new class in order to get an instance that can satisfy all our requirements.

Figure 2 is a Moose class blueprint [7] of the `PositionableStream` class. It

[7] http://www.themoosebook.org/book/externals/visualizations/blueprint

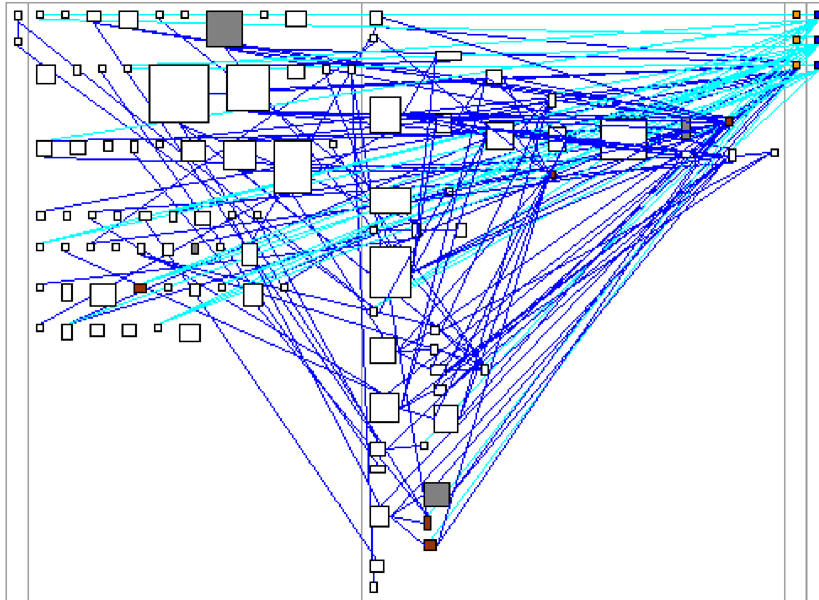

Figure 2: Moose class blueprint of the `PositionableStream` class.

visualizes a class' methods, represented as rectangles, and their interdependence, represented as lines, and consists of five columns. In the first column are the constructors, in the second are the public methods of an instance, in the third are the private methods of an instance, in the fourth are the accessors and in the fifth are the instance variables. The blue lines are invocations of other methods, the cyan lines are accesses to attributes. The width of a box represents the number of invocations in the method, the height stands for the method's lines of code. Grey methods return a constant value, brown ones override a method of the super class. We see that `PositionableStream`, from which most other streams inherit, has quite a high complexity: A lot of methods are implemented and they are highly interdependent. There are so many methods in `PositionableStream` because some of them are implemented too high in the hierarchy *e.g.,* `#int16` or `#boolean`. These methods implicitly assume that the stream is binary and have nothing to do with positioning.

In Figure 3 we show the code duplication inside the Pharo streams. The classes involved in a duplication are listed on the left and right side. The red lines connecting them stand for detected duplications. The thicker the line, the more duplicated chunks of code are detected. We find what we expected: The reading feature in `ReadWriteStream` is duplicated. But what is more surprising is the fact that `NullStream` contributes to duplication; quite a strange result if we keep in mind that a null object [12] should not do anything but provide an empty implementation of an interface. We also see that the duplication could be significantly reduced if the core streams would be refactored *e.g.,* there is duplication inside `PositionableStream`, `ReadStream` and `WriteStream`. But code

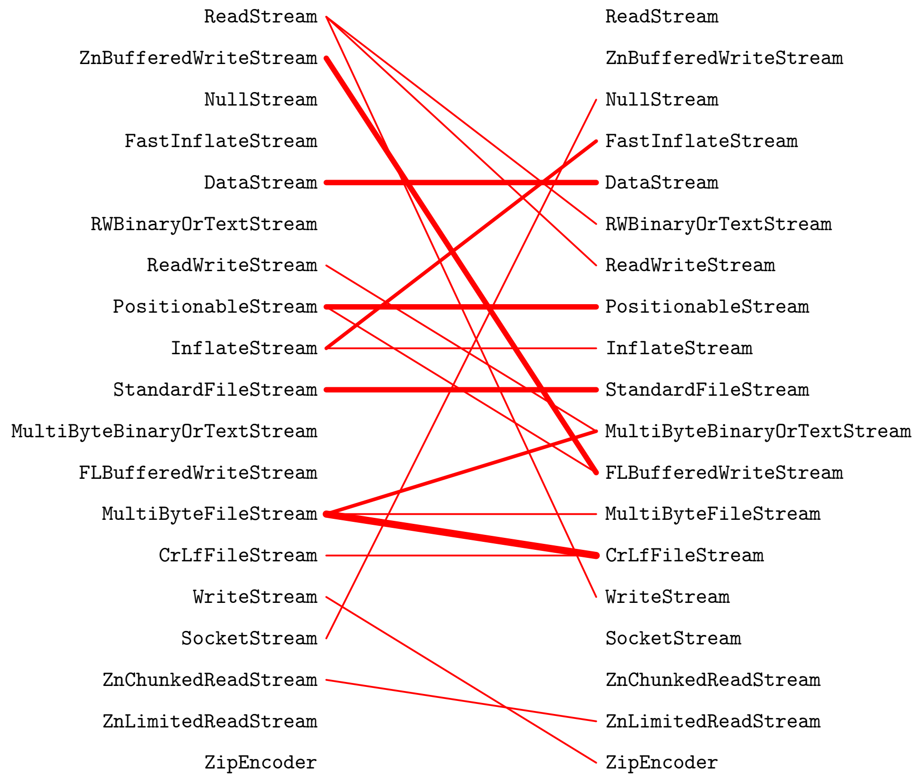

Figure 3: Moose side-by-side duplication view of Pharo streams.

duplication is not the only problem from which Pharo streams suffer, the whole architecture can be criticized too. Methods like `PositionableStream>>int16` and `PositionableStream>>boolean` implicitly assume that the stream is binary, while other methods like `WriteStream>>tab` assume it to be character-based. Since the combination of features is modeled in classes *e.g.,* `RWBinaryOrTextStream`, the architecture is prone to class explosion when extended.

2.3.3 Evaluation

Since the Pharo streams model the combination of features in classes, the framework is prone to class explosion, though we do not have hundreds of classes but just 47. Pharo streams suffer from a significant amount of code duplication. Extending this framework is hard: If we want to introduce a new feature to every stream, we would either have to create a new class for every existing class, or implement the feature too high in the inheritance hierarchy, in `Stream` or `PositionableStream`, which would make it impossible to create new streams that should not have this feature.

2.4 Analyzing Xtreams

2.4.1 Motivating Example

In Listing 3 we solved our motivating example with Xtreams. While `FileStream`, used in Pharo streams, aggregates features like buffering, positioning and reading in one god class *i.e.,* a class with a lot of responsibilities and high complexity, Xtreams separates features far more effectively. Access to the file system is provided by `XTIOFileHandle`, decoupled from any streaming feature. A file handle is then encapsulated by the base of the stream, a `XTFileReadStream`. In order to split the contents of the file line by line, a `XTTestReadSubstream` is then stacked on top of it by sending `#ending:`. This stacking of streams is a way to compose them, which in turn makes it possible to decouple features in the implementation and increases their reusability.

2.4.2 Framework Analysis

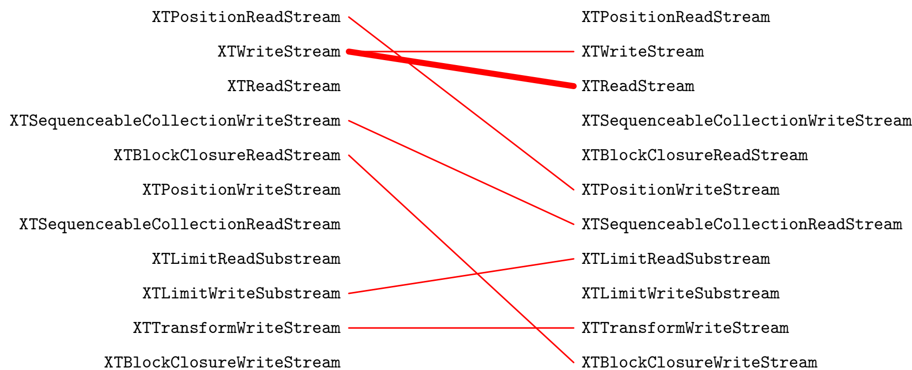

Figure 4: Moose side-by-side duplication view of Xtreams.

In Figure 4 we measured the duplication inside the Xtreams core. Duplication mostly comes from `XTReadStream` and `XTWriteStream`, so it could be ruled out by a common superclass. There are also four methods, namely `XTFileWriteStream>>insert:from:at:`, `XTRingBuffer>>growBy:`, `XTSharedQueueReadStream>>streamingInsertInto:` and `XTSharedQueueReadStream>>streamingWriteInto:` that call `Object>>shouldNotImplement`, which means that these methods were implemented too high in the hierarchy. Xtreams separates reading and writing features using inheritance properly, but this robust architecture still suffers from a significant amount of code duplication, because some features must be mixed in in both inheritance paths *e.g.,* buffering and positioning. Xtreams duplicates the access to files and sockets by providing a dedicated implementation. The features would already be implemented in the standard classes `FileStream` and `Socket`. But there are good reasons to do it this way, since those two classes could definitely profit from a refactoring.

There are 54 classes in Xtreams, counting the classes in the categories `Xtreams-Core`, `-Support`, `-Terminals`, `-Substreams` and `-Transforms`. This number is in the same order of magnitude as the amount of classes in Pharo streams. But undeniably, we can create many more combinations of features to build a specialized stream with Xtreams than with Pharo streams.

2.4.3 Evaluation

Xtreams shows no evidence of a class explosion: Though the number of classes is quite high, Xtreams provides a lot of features, each implemented in its own class. There is some code duplication, parts of it could be eliminated by refactoring, without having to change the fundamental architecture. Extending Xtreams is possible by extending `XTReadSubstream` or `XTWriteSubstream`, so there is no need to adapt multiple classes to introduce a new feature.

3 Talented Streams

In this section we explain how we avoid a class explosion and minimize code duplication by providing a meta model for reusable and composable features.

3.1 Modeling Features

We model the features listed in Section 1.2 using talents, trying to decouple them as much as possible from each other, in order to reuse them in any possible combination. By separating features from each other in the implementation, we enforce their decoupling by design. But we find a lot of features that are not completely orthogonal to each other: reading and input are highly correlated, as are writing and output. Buffering can be independent of input and output, but not of reading and writing *etc.*. We were not able to completely decouple these non-orthogonal features from each other.

3.2 Modeling Composability

A feature in isolation is useless, only a combination of multiple features provides a useful solution for a problem. We see ourselves confronted with a situation where the two seemingly orthogonal concepts of decoupling and composability should be unified. While features should be decoupled from each other to keep our framework maintainable, testable and extensible, they should be composable at the same time to maximize their reusability. Our goal is to model composability itself in a reusable way, while this model to gain composability impacts the way we have to implement a decoupled feature as little as possible.

3.2.1 Classes

Classes can be used to model composability through inheritance, polymorphism and aggregation. If we only rely on inheritance to compose features, our model

is prone to a class explosion, as the analysis of Pharo streams in Section 2.3 has demonstrated. While Xtreams models the composition through aggregation and polymorphism to avoid a class explosion, the actual composition of features is hard-wired into the framework and cannot be reused outside of it. If we want to build a collection framework following the principles of Xtreams, implementing each feature in its own class, we would have to reimplement the whole composition. This is a waste of resources according to the DRY principle [6]. If we were able to model a reusable composition of features, we could increase our efficiency, avoiding the duplicated implementation of the composition itself.

3.2.2 Traits

Traits are composable by design. We can easily compose one trait with another by creating a new trait that reuses the original traits. The problem is that traits can only be applied to classes (or other traits). To create a specific stream, we would have to compose the needed traits, create a class which uses these traits and create an instance of this class. The creation of a class has a big impact on the runtime system: In Pharo Smalltalk, a class is a global variable which will never be garbage collected unless it is removed from the system explicitly. If we work with many stream variations, we have to create a class for each and we run into a class explosion at runtime. If we compare these high costs to the low outcome of the gained ability to create an object with the specific combination of features we want, the creation of a permanent class for a temporal problem does not pay off. We could use anonymous classes, but we would still need to create a class, which is avoidable as we will see.

3.2.3 Talents

Talents are a generalization of traits and therefore composable. By using talents we can avoid the overhead needed for creating a class. With talents we are able to get the object we want by adapting an existing one, instead of having to create a factory [4] *i.e.,* a class, for it beforehand. Nevertheless, the composition of talents is still linear, which means that we have to explicitly resolve conflicts like conflicting interfaces: If we want to compose two talents that define a method for the same message selector and we need both method implementations, one method needs to be aliased. The aliased method can now only be executed by using the new message selector, and all message sends which were targeted against the implementation of the aliased method would need to be adapted too. Talents allow us to compose state and behavior dynamically, but the linearity of the composition limits its power since the constraint of unique method implementations forces us to alias conflicting method selectors.

```
1 talent := Talent new.
2 talent
3     defineMethodNamed: #fibonacci:
4     performing: 'fibonacci: n
5         (n <= 1)
```

```
6             ifTrue: [ ^ 1 ]
7             ifFalse: [ ^ (self fibonacci: n-1)+(self fibonacci: n-2) ]'.
8 talent >> { #recursiveFibonacci: -> #fibonacci: }.
9 talent
10     defineMethodNamed: #fibonacci:
11     performing: 'fibonacci: n
12         | a b |
13         a := 0.
14         b := 1.
15         n timesRepeat: [ |c| c := a + b. a := b. b := c ].
16         ^ b'.
17 object := Object new.
18 object acquires: talent.
19 object recursiveFibonacci: 5
```

Listing 4: Aliasing a method invalidates all message sends to the aliased implementation using the old selector.

In Listing 4 we alias the recursive method `#fibonacci:` to `#recursiveFibonacci:` on line 8 and add an iterative implementation with the selector `#fibonacci:` on line 9. The message sends of `#fibonacci:` in the recursive implementation now target the iterative implementation since these message sends were not adapted. Therefore the recursive implementation is only partially recursive now. If we rely solely on aliasing to work around the unique method implementation contraint, this can have unwanted side-effects. And these side effects may not even be evident: Calculating the fifth Fibonacci number on line 19 will generate the correct result, but is not calculated recursively. Due to these fragile compositions which may be created using aliasing, we decided not to use aliasing at all and provide another solution to the problem of conflicting method selectors.

3.3 Scoped Talents

We created a new type of talent allowing us to compose talents that implement a method with the same message selector. Our approach reuses the existing runtime abstractions to determine the implementation a message send is addressing. Every message is sent in a context: At dispatch time the sender and receiver of a message are known, as well as the method stack that represents the path which has led to the dispatch of the message. We use this context to select a method that corresponds to a message send, instead of directly looking up the message selector in a method dictionary. This way the same message can have a different meaning in different contexts and we can avoid the adaptation of the system before the message send occurs. This is called subjective behavior and has been the subject of an implementation called Subjectopia by Langone *et al.* [7]. We implement subjective behavior by extending talents to store the different method implementations and evaluate the context of a message send to select the appropriate implementation.

We introduced *scopes* to talents to break the linearity of the composition and to be able to use subjective behavior to avoid the unique method implementa-

tion constraint. Scopes frame a set of methods and can be easily organized to represent the dependencies in-between them. If we define the iterative version of the Fibonacci algorithm in Listing 4 in a different scope, it will not interfere the recursive implementation. A scope has a unique identifier, a state and a method dictionary. Multiple scopes are embedded within a *message graph*, which enables us to compose methods and states with the same selector. Scopes are connected through *transitions i.e.,* edges within the message graph, which define how we can execute code within a specific scope. A transition is a method that is used to execute a `BlockClosure` within a specific scope *i.e.,* the states and behavior of a scope can be referenced directly, without having to know the identifier of the scope. This message graph is part of our new meta object, the *scoped talent*. Whenever a message is received by a scoped talent, it evaluates the context, in which this message was received, to determine the scope within the message graph that was addressed. This decision strategy allows us to use subjective behavior. Scopes and transitions give the composition of state and behavior a structure which is more dynamic than aggregation and is easy to alter or decompose.

3.4 Scoped Talents in Action

Scoped talents can be used in a similar way as normal talents, the only difference being that a definition needs to specify the scope to which the definition is targeted. In Section 1.1 we introduced our motivating scenario of processing a file containing the ordered results of a marathon. We did this using existing stream frameworks in Section 2 and we will now show how this can be achieved using scoped talents. In this section we restrict ourselves to reading characters only from a file, instead of full lines, to keep the examples short.

```
1  fileStream := FileStream fileNamed: 'results.txt'.
2  stream := Object new.
3  talent := TSTalent new.
4  stream acquires: talent.
5  talent addScopeNamed: #asByte.
6  talent
7      addStateNamed: #fileStream
8      in: #asByte
9      toBe: fileStream.
10 talent
11     defineMethodNamed: #read
12     in: #asByte
13     performing: 'read ^ fileStream next'.
14 talent
15     defineMethodNamed: #read
16     in: #global
17     performing: 'read self readBelow: [ ^ self read asCharacter ]'.
18 talent
19     defineTransitionNamed: #readBelow:
20     from: #global
```

```
21      to: #asByte.
22  stream read.
```

Listing 5: Using scoped talents to convert bytes to characters.

With the first five lines in Listing 5 we create the object named `stream` and let it acquire a scoped talent named `talent`. On line 6 we create a new scope called `#asByte` in which we want to define how to read a byte from the source file stream. With the statement starting on line 7 we add the `fileStream` as a state in the scope `#asByte`. On line 10 we add a method named `#read` in the scope `#asByte` to read a byte. In the global scope, which always exists in a scoped talent, we then define that we want to convert the byte we read from the file to a character. Therefore we add another method named `#read` on line 14, but this time in the scope `#global`. This method will send `#readBelow:`, which is required to be a transition from the scope `#global` to the scope where we read a byte, which is the scope named `#asByte` in our example. We add this transition from `#global` to `#asByte` with the statement starting on line 18. We will later explain what happens when the stream is read on line 22.

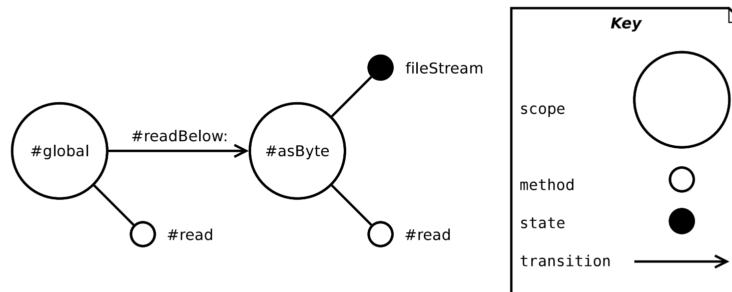

Figure 5: Message graph of the scoped talent in Listing 5

Figure 5 shows the message graph of the scoped talent of the stream defined in Listing 5. As we see, it consists of the two scopes named `#global` and `#asByte`, each of which has its own states and methods, and a transition named `#readBelow:` from the `#global` scope to `#asByte` scope. The message graph allows us to compose features by defining each feature in its own scope *e.g.,* the converting feature is defined in the `#global` scope, the input feature in the `#asByte` scope. Through this separation of features we do not need to modify a feature's definitions prior to the composition, while transitions allow us to make features work with each other. Note that we neither directly access the file for the conversion, nor do we reference the scope in which the file is read directly, but we access it through the transition named `#readBelow:`. While the scope `#asByte` is decoupled from any other scope, the method named `#read` in the scope `#global` demands that a transition named `#readBelow:` exists and that the targeted scope responds to the message `#read` as well.

Figure 6 shows what happens if the stream defined in Listing 5 receives the message `#read` on line 22. The stream in this scenario is just an instance of the `Object` class, all definitions being made on the meta object level, using a scoped

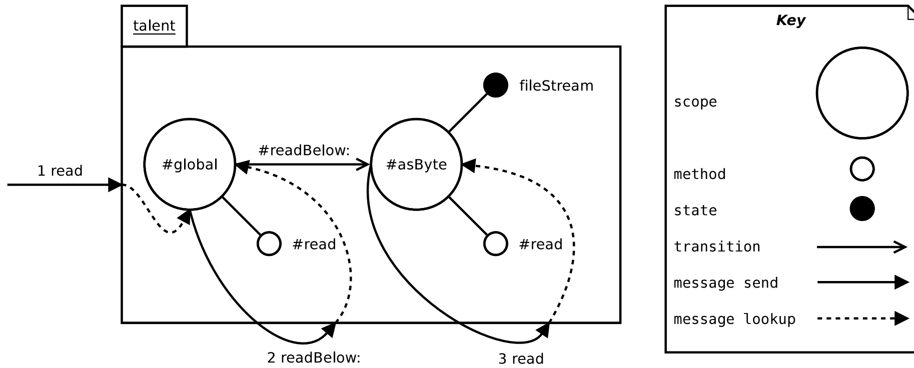

Figure 6: Delegation of the message `#read` from the stream to the scoped talent it acquired in Listing 5. The arrows form the execution path.

talent. As normally, the message selector *i.e.,* `#read`, is looked up in the method dictionary of the stream's class and executed. At this point, the message is delegated to the meta object, since the executed method is reflective. A normal talent would just select the method stored in its method dictionary which is associated with the `#read` selector, but a scoped talent evaluates the context of the message to derive the stack of scopes, which have been entered during the execution, to select the method to run from the multiple method dictionaries of the scopes in its message graph. Just follow the arrows, including the transition, to see which message is sent and how it is looked up in the different scopes.

3.5 Evolving Scoped Talents to Stream Features

We modeled each feature listed in Section 1.2 as a scoped talent, using class templates as factories. These templates are discussed in Section 3.6. Through factories we gain reusability since we can create instances of the same feature very easily, while composability is given through the use of scoped talents. Decoupling can be achieved by defining only a fine-grained feature as a scoped talent.

```
1  fileStream := FileStream fileNamed: 'results.txt'.
2  stream := Object new.
3  talent := TSTalent new.
4  stream acquires: talent.
5  fileStreamReader := TSByteStreamReader new.
6  fileStreamReader instVarNamed: #byteStream put: fileStream.
7  fileStreamReader adapt: talent in: #asByte.
8  convertingReader := TSConvertingReader new.
9  convertingReader instVarNamed: #block put: [ :read | read asCharacter ].
10 convertingReader adapt: talent in: #global.
11 talent
12     defineTransitionNamed: #readBelow:
13     from: #global
14     to: #asByte.
15 stream read.
```

Listing 6: Using features modeled as scoped talents to convert bytes to characters.

Listing 6 reads a character from a file through a stream, just as in Listing 5. But instead of explicitly defining each scope, behavior and state, we collected behavior and state of a feature in a template *i.e.,* `TSByteStreamReader` and `TSConvertingReader`, which we use to create or adapt the scoped talent of the stream. We also define a transition from the `#global` to the `#asByte` scope, since the `convertingReader` demands this transition to point to the scope which reads the elements to be converted. The emerging pattern is that we can use features as granular building blocks, reified in scopes, and transitions to glue them together. Figure 7 shows how granular features are composed as in Listing 6 by adapting a scoped talent during the composition. The composition of features very much resembles the combination of Lego bricks to build a bigger model. Just as a Lego brick, each feature is specialized to serve for a specific purpose, and each feature can be used in combination with any other feature by embedding it in a scoped talent.

```
1 read
2     ^ block value: (self readBelow: [ self read ])
```

Listing 7: Source of `TSConvertingReader>>read`

In Listing 7 we have the source code of the method `TSConvertingReader>>read`. This is called when reading from the stream created in Listing 6. This method seems to be recursive, but in reality, it is not. The message `#read` is sent in a block that is executed in a different scope, because it is the argument to the transition `#readBelow:`. With the transition `#readBelow:` we navigate from the scope `#global` to the scope `#asByte`. So the message `#read` is actually executed in the scope `#asByte` and reads a byte from the results file.

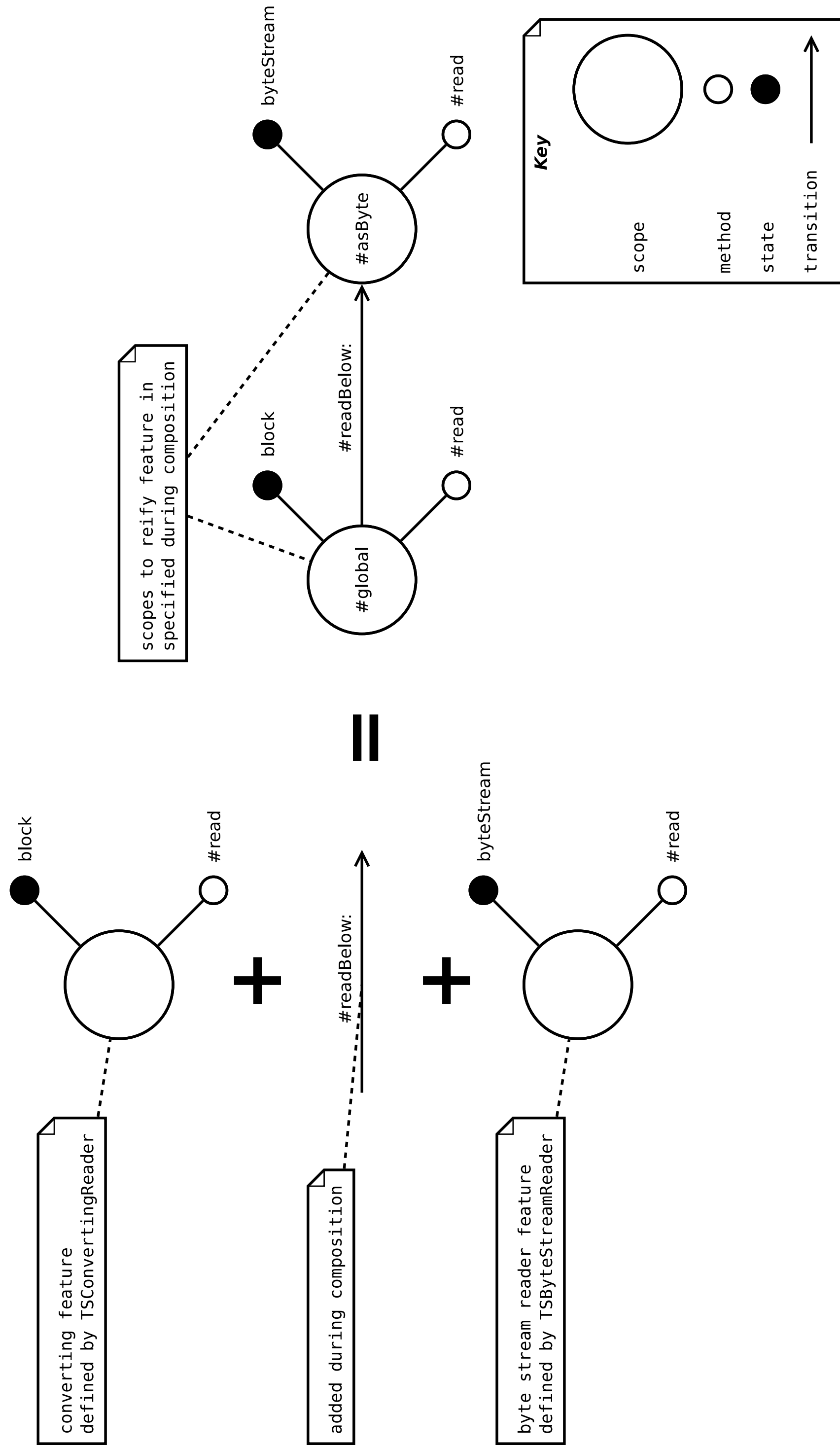

Figure 7: Composition of features by adapting a scoped talent as in Listing 6.

```
1 fileStream := FileStream fileNamed: 'results.txt'.
2 stream := TSStream new.
3 (stream readFromByteStream: fileStream in: #asByte)
4     convertingIn: #asCharacter with: [ :read | read asCharacter ].
5 stream read.
```

Listing 8: Reading characters from a file using Talented Streams.

While building our stream from features and transitions is certainly simpler than building it using the lower level abstractions of a scoped talent *i.e.,* scopes, behavior and state, we can proxy the verbose composition thereof by adopting some conventions for the features we are composing. Our implementation has three different types of features: readers, writers and positioners. Every reader implements the reading feature and another feature from the list in Section 1.2. In case of the writers, they all implement the writing feature. The positioners all implement the positioning feature. Some features do not require any transitions; we call them *terminal features*, since they directly operate on a resource *e.g.,* the positioners. But most features are *non-terminal features*, requiring one transition to process the outputs of another feature. For non-terminal reader features, we called this transition `#readBelow:` and analogously `#writeBelow:` for writers. With these conventions we are able to simplify the composition of stream features. In Listing 8 we solve again the problem of reading a character from a file, but this time proxying the composition through a builder. The composition starts with a terminal feature and can be extended by adding an arbitrary number of non-terminal features, connecting them sequentially through the appropriate transitions.

3.6 Talent Templates

During the implementation we had to work a lot with talents. To create, organize and refactor talents, we found that the tools provided by Pharo, mainly the system browser, do not give us all the features we need. In order to work with talents, a first approach by Jorge Ressia was to create a dedicated talents browser. But during the development of this project, we found that the explicit separation of talents and classes has a lot of side effects *e.g.,* committing talents to a version control system. We think that the Pharo system browser is great, it gives us all the features we need if we work with classes, and most of these features would be useful for the development of talents too. For example, we missed pre-compilation, auto formatting and syntax highlighting badly, as much as all the other refactoring tools. Essentially, classes and talents both are representations of the same abstractions *i.e.,* state and behavior. Therefore we decided to create a template, which is a class, from which a new talent can be created or adapted. The same way classes are used as a factory for instances, we use a class as a factory for talents. By using a class as factory to create talents, we can reuse all the development tools from the system browser that were originally designed to work with classes. This means that we can develop

talents with the same convenience as we have with classes: We can use syntax highlighting, pre-compilation *etc.*, and we can even use inheritance for talents. Though it should be said that we do not claim that this is the best way to work with talents. We created these class templates to meet the requirements of an existing tool to reuse its features, but we should also think about whether the tool meets the right requirements. The system browser was designed to work with classes exclusively, so there may be the need to adapt the tool in order to create a better user experience for the development of talents. In other words, quoting Marshall McLuhan: "We shape our tools and thereafter our tools shape us." [8]

4 Validation

We want to verify in this section that our framework solves the problems we identified.

4.1 Motivating Example

```
1 fileStream := FileStream fileNamed: 'results.txt'.
2 stream := TSStream new.
3 (stream readFromByteStream: fileStream in: #asByte)
4     convertingIn: #asCharacter with: [ :read | read asCharacter ];
5     collectingIn: #asLine with: [ :read | read ~= Character lf ];
6     convertingIn: #asString with: [ :read | read as: String ].
7 stream read.
```

Listing 9: Reading the first result of the marathon stored in the file in Listing 1 using Talented Streams.

In Listing 9 we show how we can solve our motivating problem with our framework. We access the file by reusing the functionality of `FileStream`. We chose to access files through this class, because a class named `File` which would encapsulate system calls to the file system is missing and we did not want to duplicate this feature. With the next lines, we compose the features we need: We need to convert the bytes of the file to characters, collect them in a line and convert the line to a string. The selectors `#asByte` `#asCharacter`, `#asLine` and `#asString` are the identifiers of the scopes where we reified these features, their order defining how they are connected. As we see, the problem to read a line from a file takes as much as seven lines with our framework, which is more than the two lines needed for Pharo streams and more than the four lines needed for Xtreams. This is because we have to explicitly compose features dynamically to create a stream, instead of relying on a predefined static combination.

[8] Marshall McLuhan, Understanding Media: The Extensions of Man, 1964

4.2 Framework Analysis

We have seen in Listing 9 that the creation of a stream is more verbose in our framework than the creation in Pharo streams or Xtreams. This can be seen as a negative side effect of the architecture of the framework, but on the positive side it is exactly this verbosity which makes our framework more versatile than Pharo streams and Xtreams: Any combination of features is instantiable, not just those the framework's developer had thought of. As soon as our framework is extended by adding new features, we unfortunately need to adapt the composition builder, or we lose the ease of the simple composition. If we do not adapt the composition builder, we have to use the even more verbose pattern as in Listing 6 for this particular feature. This is a weak point in our framework: While the features are decoupled from each other, they are coupled to the composition they are used with. Possible solutions for this problem would be either to let the composition builder acquire a talent that proxies the composition for the new feature or pushing the responsibility of composition to the talents themselves.

Our implementation [8] does not put a high performance as a requirement. But since a slow streams framework is as useless as a framework missing essential features, we are not going to leave the performance issue without a comment: There is no compilation going on after the composition of a talented stream. This leaves the lookup of methods and states as the only potential performance brake. Since we are using subjective behavior for both method execution and state access, it is impossible to predict the specific method or state addressed at compile-time. Therefore these accesses can not be inlined at compile-time and need to resolved at runtime exclusively. As discussed in Section 3.3, the scope in which a method or state should be accessed is derived from the context at runtime. Our decision strategy that resolves these runtime accesses traverses the execution stack until it finds the first active scope that defines a method or state named appropriately. The traversal of the stack is of linear time complexity in the number of currently executing methods, as well as the actual identifier lookup. This results in a polynomial time complexity, which could be reduced to a linear complexity, since the identifier lookup can be implemented in constant time complexity using hashing.

The duplication detector of Moose did not detect any significantly big, duplicated chunks of code in our Talented Streams, nor the underlying scoped talents. Though we have some minimal code duplication inside the readers and writers, because methods like `TSElementBasedWriter>>writeAll:`, `TSElementBasedWriter->>writeAllFrom:from:` and `TSElementBasedWriter>>writeAllFrom:from:to:` are just proxied calls to `TSElementBasedWriter>>write:`. Each feature is implemented as a talent and is therefore highly reusable, and it also enforces the decoupling of features by design.

There are 37 classes in the category `Talents-Streams-Model`, our streams framework. 31 are feature templates, of which 8 are abstract, the remaining 23 concrete.

4.3 Evaluation

Our framework does not suffer from a class explosion, since each feature is implemented as a fine-grained, composable talent. Introducing a new feature would only need us to create a single talent for that feature. This is because we model composable features instead of the combinations of features. We do not have a significant amount of code duplication, since we explicitly separated features from each other by design. The price for this flexibility is a performance impact: While the state accesses and method execution can be inlined at compile-time in a traditional class based model, it can not in our model since we use subjective behavior. The current implementation of these accesses has a polynomial time complexity, which may be unacceptable in many cases. Though a linear time complexity can be achieved, it will never be possible — in the current model — to reduce it to the constant time complexity of an inlined identifier lookup as in the class based approaches.

5 Conclusion

We have built a feature-rich stream framework that neither leads to a class explosion when extending it, nor does it have a significant amount of code duplication. We reduced code duplication by defining only fine-grained features. Whenever we were tempted to copy a method, or parts of it, we created a new feature for it instead. Multiple features can be easily composed to a scoped talent. Since all these features are composable with each other at runtime, it is not necessary to have a class which represents each possible combination of features. This framework is built on top of dynamic units of reuse called talents. Talents were extended with *scopes* to be able to model subjective behavior for method execution and state access. We have proved that scoped talents can be used to model a complex architecture while overcoming class based architectures' limitations. We also found a way to integrate the talent development into the standard class-oriented tools by using classes as templates for talents.

The core of OOP is to work with objects which encapsulate common state and behavior. We advocate for object reuse, instead of class reuse. Currently, classes are used as factories to create objects of the same type and inheritance is the composition mechanism that enables behavior and state reuse. The concept of classes has absorbed a lot of responsibility and power over time. *Object*-oriented programming has become — or always was — *class*-oriented programming in many programming languages. Traits have shown very well that inheritance is not the only possible way of reuse, but they still target classes. With talents we want to bring the objects back into the focus of OOP, pushing classes into the back row. Not because we dislike them, but as we have shown, some problems, like streams, require a more dynamic and flexible approach. These problems are difficult to solve with classes. We want to jolt on this socket a bit to spark a new discussion that questions and rethinks the very foundations of OOP.

Our streams framework is far from being as complete as Xtreams. We have not implemented an appropriate exception handling *e.g.,* buffers need to be managed by the user. When one tries to read from an empty buffer, an exception is raised; there is no recovery. The composition of scoped talents is powerful, but there are still some problems we have not solved, *e.g.,* we need to take a lot of care when naming temporary variables: If they conflict with the name of a state in any scope, these will, in our current implementation, lead to strange runtime errors that are hard to debug and even harder to predict, since the pre-compiler is not able to detect them. And as we have seen in Listing 7, some talents rely on the composition strategy they are used with: the class `TSConvertingReader` relies on the method `#readBelow:` to be a transition, and is therefore coupled to the composition. An introduction of a new feature *e.g.,* a new talent template, would need us to adjust the builders to perfectly integrate it into our framework. We tried to decouple the definition and the composition of features from each other, because we interpreted features and their composition to be orthogonal. But in our model they are not completely orthogonal, which resulted in their coupling as we have observed. A possibility to explore is whether talents should define their composition by themselves.

Performance is an important non-functional requirement which we did not fully analyze. In our implementation of subjective behavior for method and state access, the necessary traversal of the method stack at runtime increases the time to execute an operation by a polynomial factor. Using a different implementation of subjective behavior, restricting each stream to be able to execute methods and access states in only a single scope in every method, could reduce the added time complexity to a constant factor. Therefore, a performance comparable to that of Pharo streams or Xtreams should be possible to achieve.

The composition mechanism and the decision strategy used for subjective behavior are the complex parts of scoped talents. However, the explicit reification of composition of state and behavior through scopes enables dynamic reuse. This dynamic reuse feature is the key for solving the code duplication and class explosion problem.

6 Acknowledgments

I thank Jorge Ressia for supporting me during this project. Your precise questions during our discussions always got my gears grinding and I learned much more during this project than I expected.

References

[1] Andrew P. Black, Nathanael Schärli, and Stéphane Ducasse. Applying traits to the Smalltalk collection hierarchy. Technical Report IAM-02-007, Institut für Informatik, Universität Bern, Switzerland, November 2002. Also available as Technical Report CSE-02-014, OGI School of Science & Engineering, Beaverton, Oregon, USA.

[2] Damien Cassou, Stéphane Ducasse, and Roel Wuyts. Redesigning with traits: the Nile stream trait-based library. In *ICDL'07: Proceedings of the 15th International Conference on Dynamic Languages*, pages 50–75, Lugano, Switzerland, August 2007. ACM Digital Library.

[3] Steve Cook. OOPSLA '87 Panel P2: Varieties of inheritance. In *OOPSLA '87 Addendum To The Proceedings*, pages 35–40. ACM Press, October 1987.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Professional, Reading, Mass., 1995.

[5] Tudor Gîrba. The Moose book, 2010.

[6] Andrew Hunt and David Thomas. *The Pragmatic Programmer.* Addison Wesley, 2000.

[7] Daniel Langone, Jorge Ressia, and Oscar Nierstrasz. Unifying subjectivity. In *Proceedings of the 49th International Conference on Objects, Models, Components and Patterns (TOOLS'11)*, volume 6705 of *LNCS*, pages 115–130. Springer-Verlag, June 2011.

[8] Manuel Leuenberger. Talented streams. implementation. Bachelor's thesis, supplementary documentation, University of Bern, February 2013.

[9] Jorge Ressia, Tudor Gîrba, Oscar Nierstrasz, Fabrizio Perin, and Lukas Renggli. Talents: Dynamically composable units of reuse. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2011)*, pages 109–118, 2011.

[10] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behavior. Technical Report IAM-02-005, Institut für Informatik, Universität Bern, Switzerland, November 2002. Also available as Technical Report CSE-02-014, OGI School of Science & Engineering, Beaverton, Oregon, USA.

[11] Nathanael Schärli, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, and Andrew Black. Traits: The formal model. Technical Report IAM-02-006, Institut für Informatik, Universität Bern, Switzerland, November 2002. Also available as Technical Report CSE-02-013, OGI School of Science & Engineering, Beaverton, Oregon, USA.

[12] Bobby Woolf. The null object pattern. In *Design Patterns, PLoP 1996.* Robert Allerton Park and Conference Center, University of Illinois at Urbana-Champaign, Monticello, Illinois, 1996.