Talented Streams

Implementation

Bachelor's thesis, supplementary documentation
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

This document explains the implementation of scoped talents, talent templates and talented streams. It makes it possible to work with and extend the current code base or recreate the same architectures in a different programming environment.

Contents

List of Figures

List of Listings

1 Introduction

To show how the different modules of our framework work together, we will use the same example as in the thesis [1]. In our scenario, we want to find the winner of a marathon. The results of this marathon are stored in a file, in which each line contains the name and time of a contestant. The lines are ordered by the runners' times, with the winner at the top of the file, as in Listing 1. We use talented streams to iterate over the lines of the presumably large file. Listing 2 shows how one can create and read from such a stream.

```
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.

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

2 Scoped Talents

In Figure 1 we show the architecture of our new scoped meta model, the scoped talents. The scoped talent, represented through the class `TSTalent`, extends `TSScopeMetaObject` which provides the basic interface to define scopes, states and methods. In our implementation a scoped talent is incomplete in comparison to a normal talent: Our scoped talent does not provide altering or aliasing of its definitions, but since it is built on the same bases, the Bifröst [2] meta object framework, it should be possible to introduce this functionality. The heart of a scoped talent is its message graph, an instance of the `TSMessageGraph` class. Instead of implementing scopes with a method dictionary, the graph's edges are all `TSMessageEdge`s and those edges leaving a scope vertex, a `TSScope`, define the methods defined by this scope, including transitions. The message graph's leaf vertices are all plain `TSMethod`s, non-leaf vertices `TSScope`s which at same time define the transition into this scope. The class `TSTalentTemplate` defines a talent factory template, as described in Section 2.1.

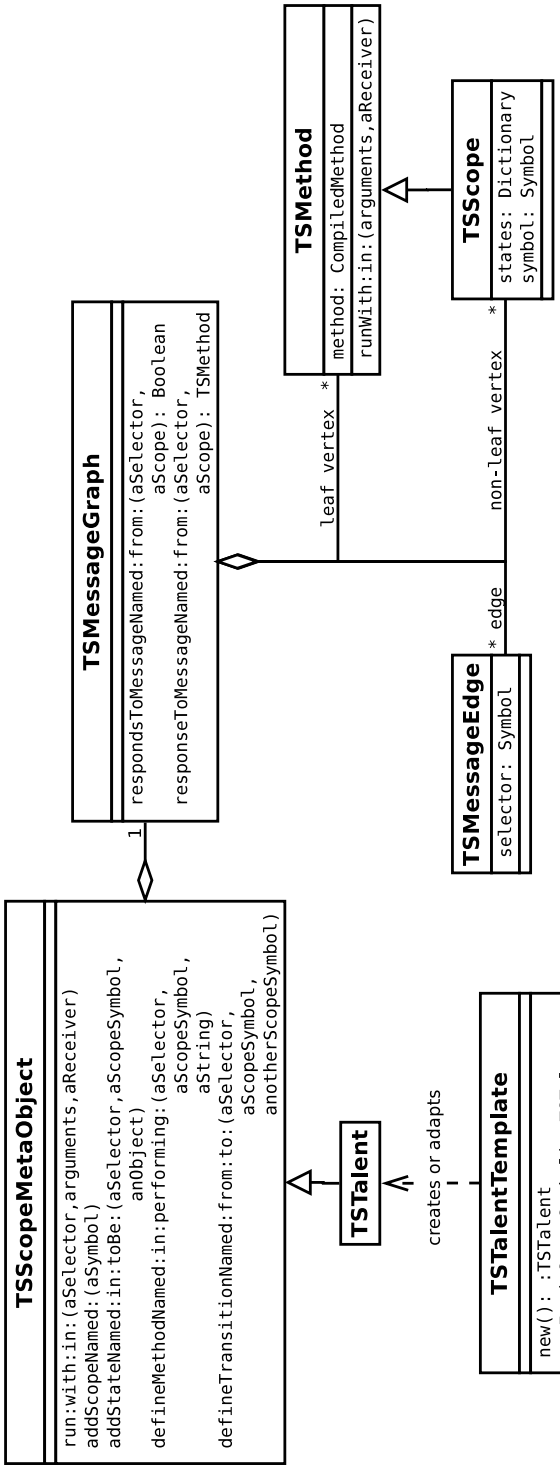

Figure 1: UML class diagram for scoped talents

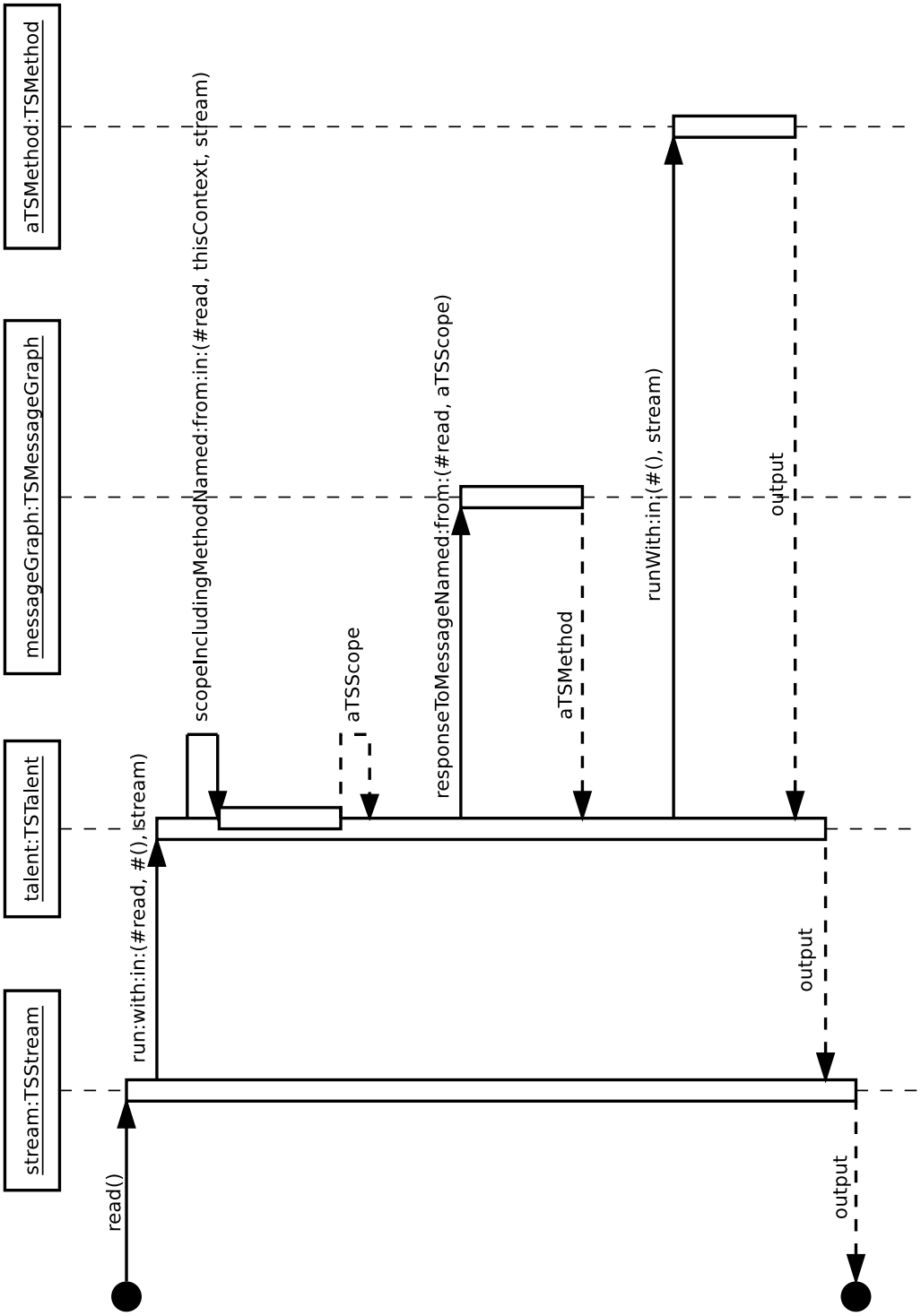

Figure 2: UML sequence diagram for reading from the stream in Listing 2

In Figure 2 we see how the message send `#read`, as it occurred in Listing 2 on line 7 is delegated from the stream to its meta object, which is a scoped talent. The meta object then traverses the current context to find the scope that message was sent in, then looks up the method in its message graph and runs it.

2.1 Talent Templates

Talent templates are classes which act as factories to create or adapt talents. Through the use of classes as templates for talents, we can easily create multiple instances of the same talent. We can use the same development tools *e.g.,* syntax-highlighting in the Pharo browser, that were initially designed to work with classes, for the development of talents. We use a dedicated category called `talent-define` to mark methods that will be used in the created or adapted talent. This removes the possibility to organize a talent's methods into different categories, but since talents are mostly small, we considered this to be an acceptable trade-off.

In Figure 3 we show the basic scenario how a template is used to adapt a talent with the definitions from the template. If the template should create a new talent, use `#new` or `#newIn:` to reify the template's definitions in a specific scope.

3 Talented Streams

One point that is essential for streams, is the way the framework integrates external resources, namely sockets and files. We want to create a new stream framework, independent of any already existing implementation of streams. But in the case of files, we see ourselves confronted with a decision for which no completely satisfying solution exists. We could either reuse the existing `FileStream` which would result in the duplication of the reading and writing features, or we could create a `File` class which would duplicate the file access feature. We decided to reuse `FileStream` and `SocketStream` instead of duplicating any features. With talents, we have the tool to adapt *any object* to what we need and even how we want to use it. We provide talents for the standard `FileStream` and `SocketStream`, as for the `SharedQueue` and `SequenceableCollection`. This way we are able to decouple our framework from an external dependency as much as possible. If the file access should get changed in future, we only have to write a new talent for it, replacing the old one; it will not affect any other talent which is not programmed against the implementation of the replaced talent.

Our framework consists of the utility classes `TSCache` and `TSBuffer`, talent templates extending `TSTalent-Template`, composition builders extending `TSStream -Builder` and a facade `TSStream`. `TSCache` is used to cache stream contents that are already read, so rereading parts of the stream does not need to access the underlying external resource. This cache can also be used to make streams positionable that were not positionable initially, like a stream on a network socket.

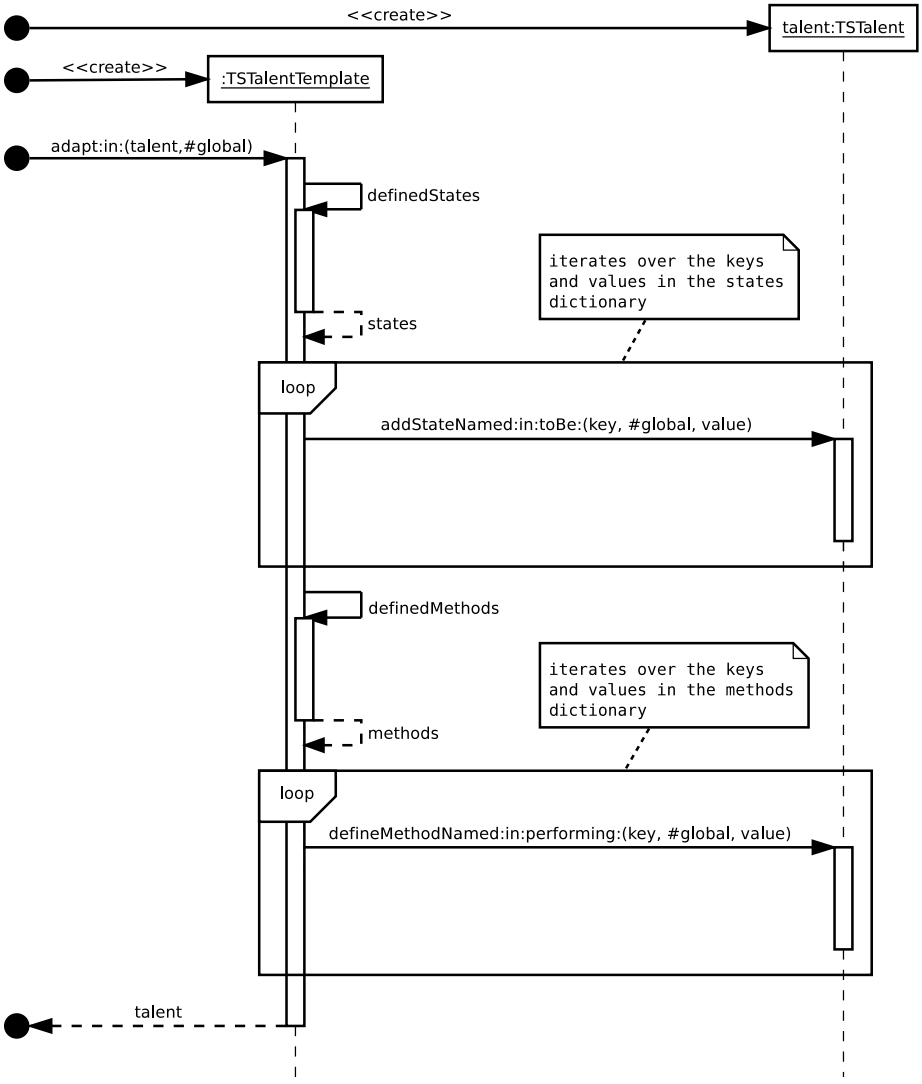

Figure 3: UML sequence diagram showing the adaptation of a talent using a template class.

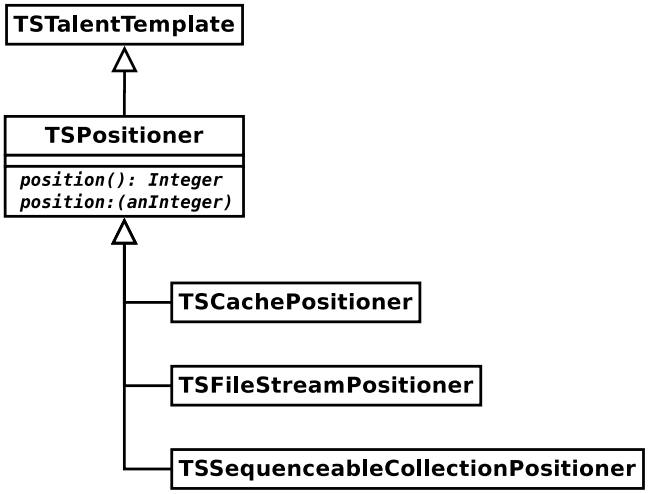

Figure 4: UML class diagram showing the inheritance hierarchy of the positioners.

`TSBuffer` is a simple buffer. It is basically a queue with a special interface for reading and writing. The talent templates are simple classes that are used to create and adapt the talents we are using, as discussed in Section 2.1. They provide a factory for the features positioning (see Figure 4), reading (see Figure 5) and writing (see Figure 6). Composition builders are used to build streams by composing features with each other. They provide a simplified interface for the composition of scoped talents (see Figure 7). The facade `TSStream` proxies the initialization on a resource and the building of readable, writable and positionable streams. Its meta object is a scoped talent, that gets adapted when initializing and building the stream.

In the sequence diagram in Figure 8 we see how the stream from our motivating example with talented streams from Listing 2 is actually created. To keep it short, we excluded the adaptations in the scopes `#asLine` and `#asString`. In this process, the facade, the builders and the templates are involved. Our builder handles the transitions from one scope to another by inserting new scopes in the order they are declared. The feature is inserted by adapting the talent of the stream object: a new scope is inserted and the composition connects scopes with each other by adding or adapting transitions.

The resulting message graph of the stream's scoped talent in Listing 2 is shown in Figure 9. There are always two transitions added when a new scope is added: a relative transition goes from the origin to the target scope, it is called `#readBelow:` for the `TSReadStreamBuilder`, and an absolute transition goes from the global to the target scope, it is named similarly to the target scope *e.g.,* if the target scope is named `#asByte`, this transition is named `#inAsByte:`. With the relative transitions we are able to switch scopes from within the message graph. This is useful in the composition because the composed features do not need to know the identifier of a scope, they just need to know the selector of the

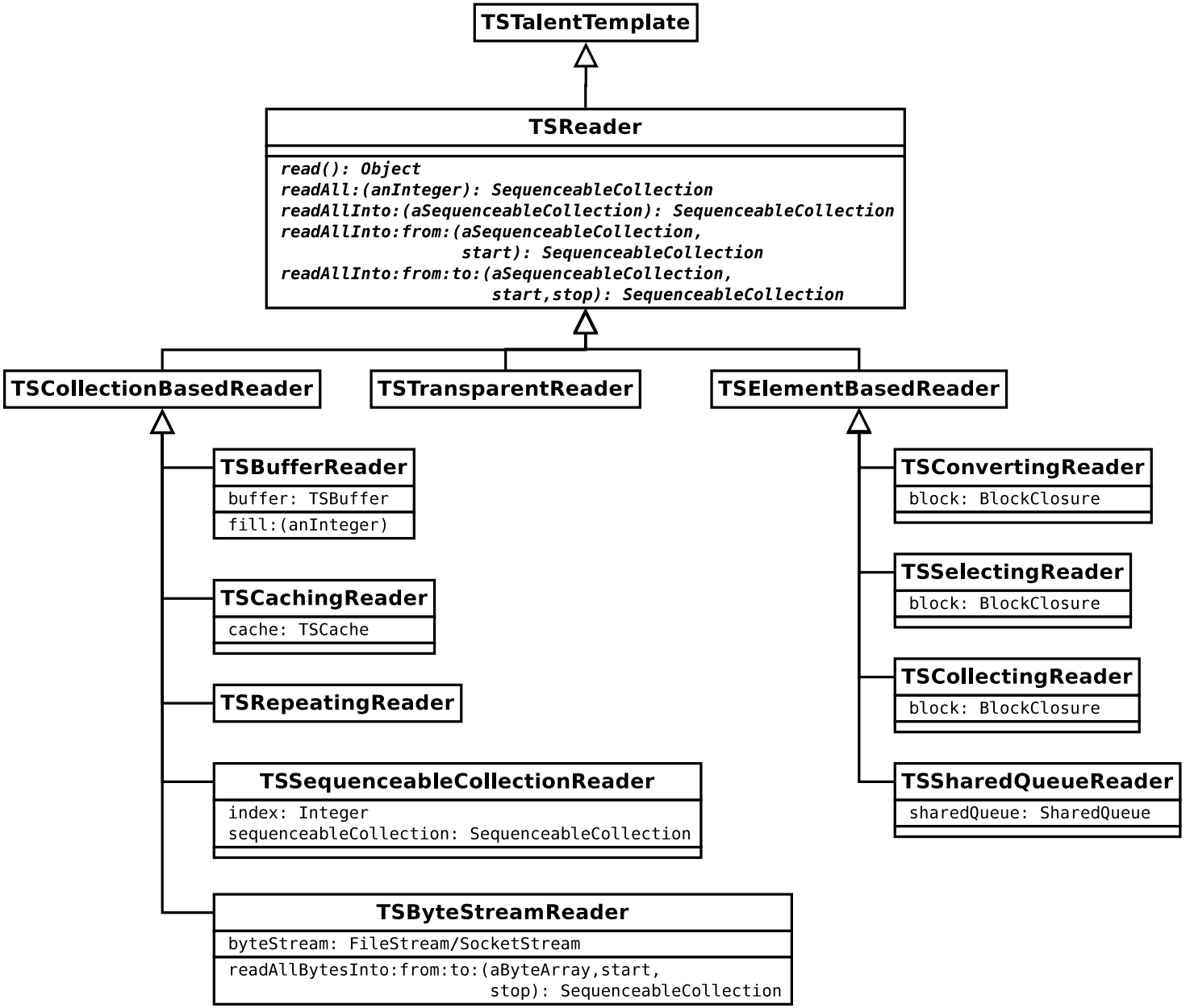

Figure 5: UML class diagram showing the inheritance hierarchy of the readers.

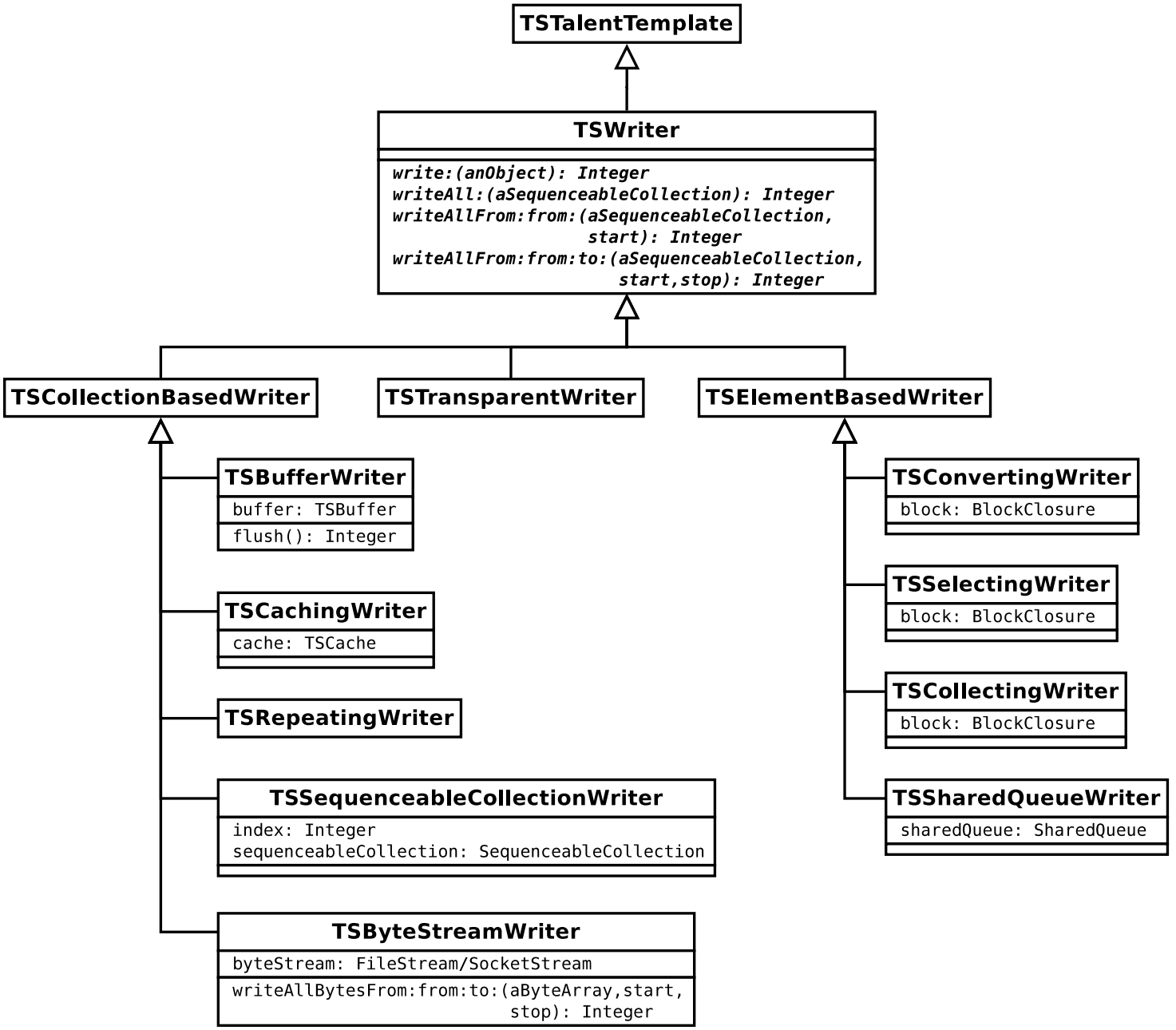

Figure 6: UML class diagram showing the inheritance hierarchy of the writers.

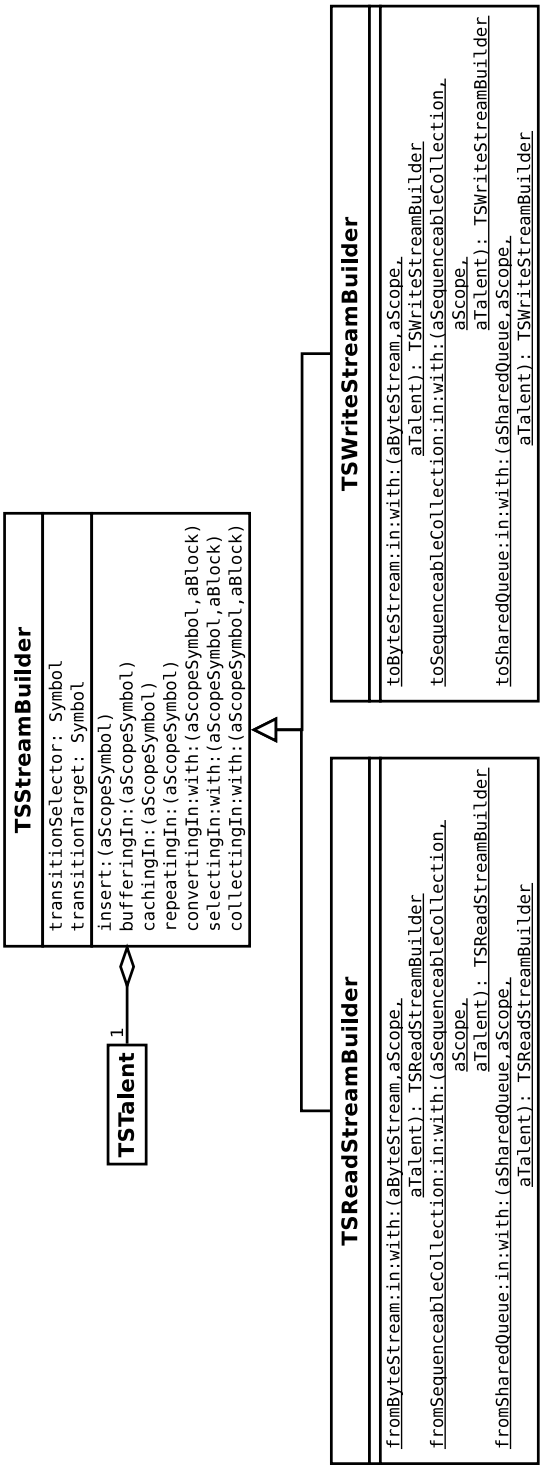

Figure 7: UML class diagram showing the inheritance hierarchy of the builders.

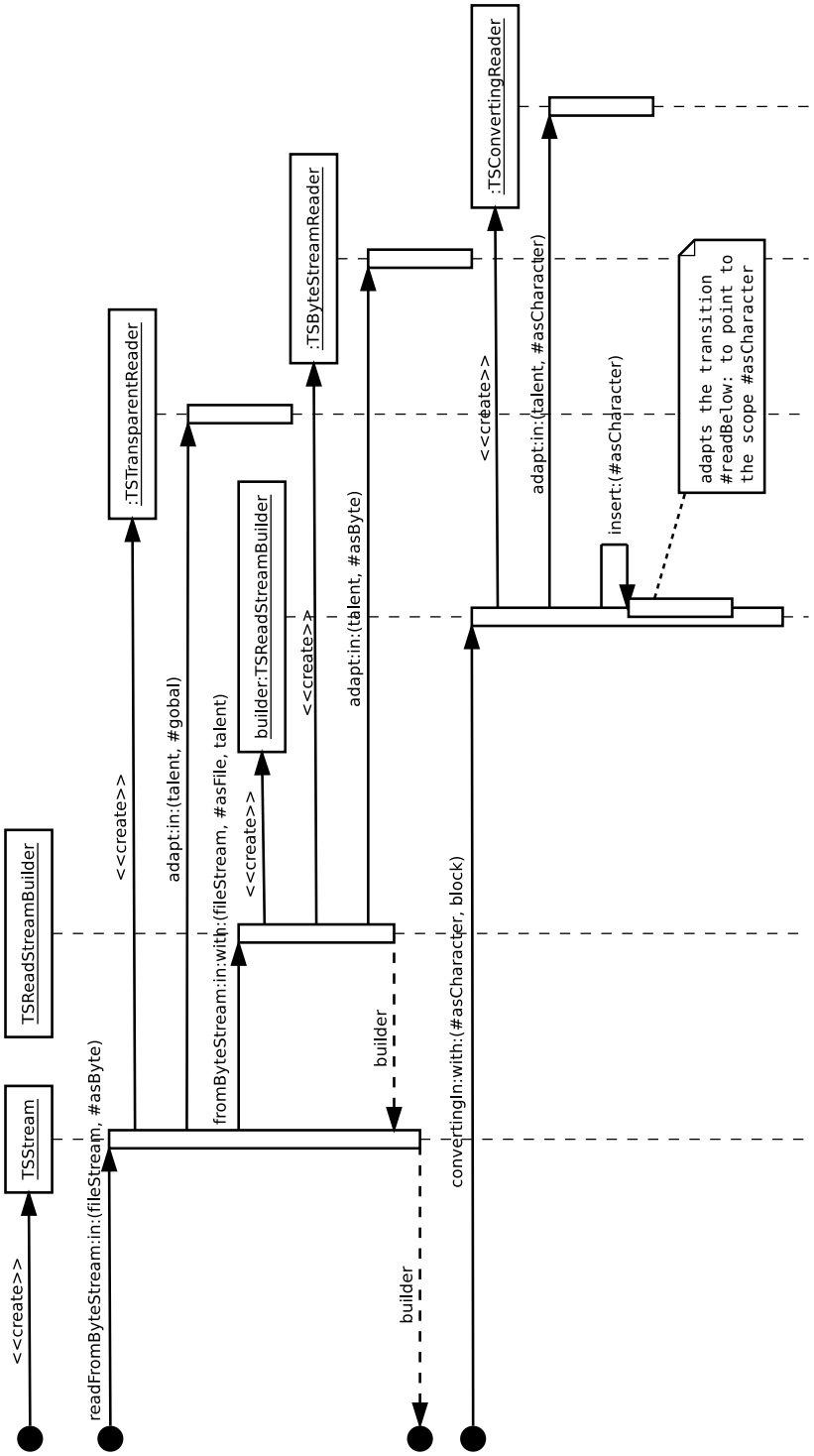

Figure 8: UML sequence diagram for the building of the stream in Listing 2

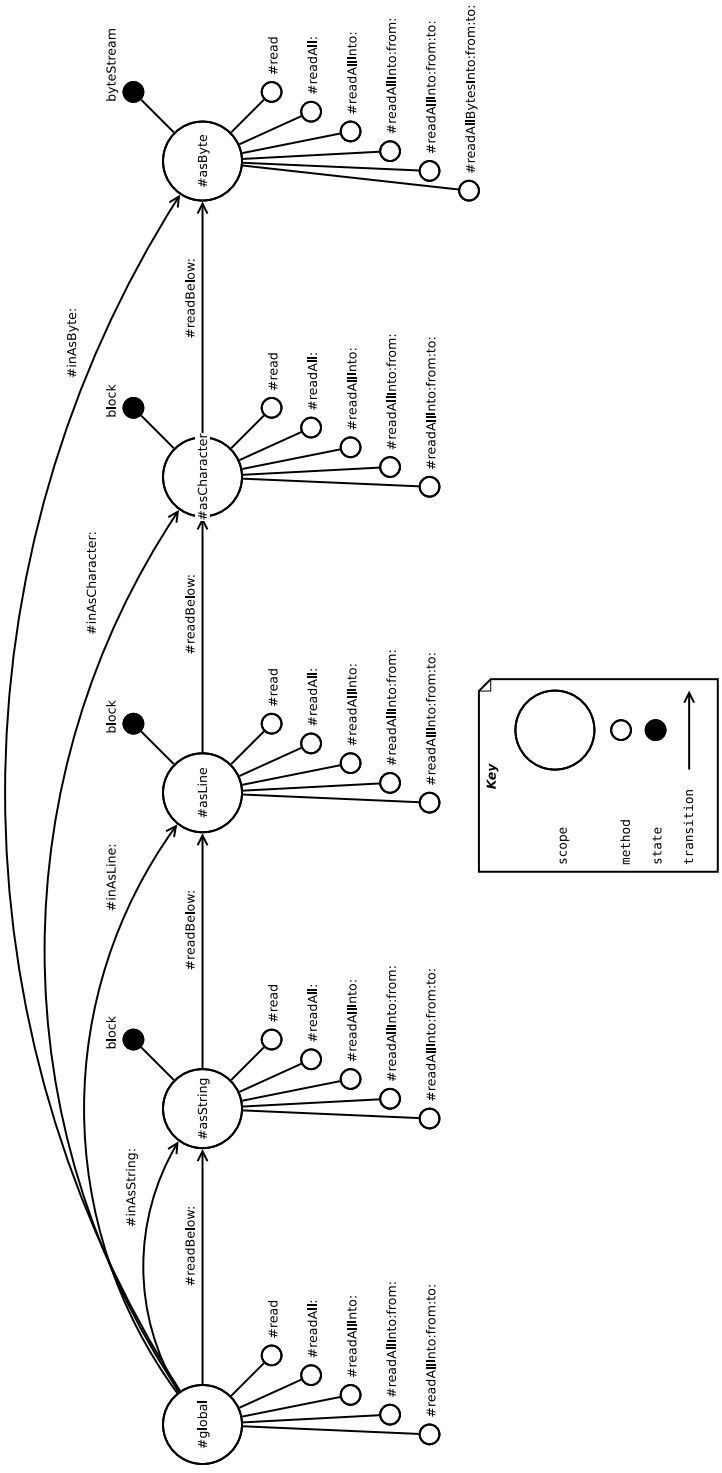

Figure 9: Message graph of the scoped talent of the stream in Listing 2

relative transition. Absolute transitions on the other hand are useful to access specific scopes from the global scope *e.g.,* we can fill a buffer manually from outside the message graph.

We do not have a lot of code duplication in the classes we created, but if we look at Figure 9, we have a lot of methods with the same name, and some of them are actually exactly the same methods. *e.g.,* in Listing 2 we composed our stream from two instances of `TSConvertingReader`, which added some methods twice to the message graph, in different scopes though. We could still lower the level of duplication in the objects we created, if we would not define the same method multiple times. This could be achieved by using dedicated scopes for these methods and call them always in this particular scope.

References

[1] Manuel Leuenberger. Talented streams. objects composed from features. Bachelor's thesis, University of Bern, February 2013.

[2] Jorge Ressia. *Object-Centric Reflection*. Phd thesis, University of Bern, October 2012.

A Get the Code

The Talented Streams framework and the scoped talents are hosted on SqueakSource [1]. You can install it in your Pharo 1.3 image using Gofer [2] by executing the following code in Listing 3 in a workspace.

```
Gofer new
    squeaksource: 'talents';
    package: 'ConfigurationOfTalentsStreams';
    load.
(Smalltalk at: #ConfigurationOfTalentsStreams) loadDefault
```

Listing 3: Code to get the Talented Streams framework with Gofer.

[1] http://www.squeaksource.com/talents.html
[2] http://www.lukas-renggli.ch/blog/gofer