Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Master Thesis:

"Static Code Validation for Traits"

Author:
Aizcorbe, Juan Sebastian
LU: 83/98
Juansebastian.aizcorbe@gmail.com

Director:
Lic. Hernán Wilkinson
hernanwilkinson@gmail.com
Buenos Aires December 06Th of 2012

Resumen

Traits es un nuevo concepto en la programación orientada a objetos que extiende a la herencia simple permitiendo compartir comportamiento entre clases utilizando composición.
Debido a tratarse de un nuevo modelo de programación, es necesario el estudio de sus distintas características para detectar sus fortalezas y debilidades, como así también resulta necesario el desarrollo de herramientas que ayuden en su introducción y utilización efectiva.
Entre las características a estudiar se encuentra la identificación de errores específicos del uso de Traits y la factibilidad de detectarlos y corregirlos automáticamente por medio de una herramienta.
Esta tesis identifica distintos tipos de errores específicos cuando se usan los Traits y los clasifica según el elemento generador dicho tipo de error. Con ese estudio se logró también definir con mayor rigurosidad las características sintácticas y semánticas de los elementos que conforman los Traits.
También se presenta la implementación de una herramienta de chequeo estático de código basada en Smalllint para detectar los errores específicos de Traits detallados en el trabajo de investigación como así también cambios al ambiente de Pharo que mejoran la implementación de este modelo.
Por último se presenta el análisis de los resultados de utilizar la herramienta de chequeo estático de código en muestras reales de programas implementados utilizando Traits.

Abstract

Traits is a new concept on object-oriented programming which extends simple inheritance and lets the programmer share behavior between classes using composition.
Since Traits is a new programming model, an analysis of its characteristics is needed to detect its strengths and weaknesses. It is also needed to develop tools to help its addition and effective use.
One of Traits characteristics to study is the identification of errors generated by Traits use and the feasibility of their detection and their correction using and automatic tool.
This thesis identifies several Traits specific error types and classifies them according to the Traits element which generates the error. This study also achieves a more strict definition of the syntactic and semantic characteristics of the Traits elements.
It also presents a tool implementation for static code checking based on Smalllint for detecting the Traits errors described in the previous research and changes to Pharo to improve its Traits model implementation.
Lastly, this thesis also presents an analysis of the static code checking tool use on real code samples implemented using Traits.

Acknowledgements

Contents

List of Figures

1 Introduction

Traits is a new concept on *object-oriented programming* which lets the programmer define and share behavior between *classes* using *composition* instead of *inheritance*. Traits face the limitations of *Single Inheritance* and others *inheritance* mechanisms like *Multiple Inheritance* [MIWeb] and *Mixin Inheritance* [MixWeb], extending the widespread and broadly accepted *Single Inheritance* behavior sharing mechanism instead of replacing it [S/05].

With *Traits*, like with any new concept, it is important to identify *errors*, *error types*, and best practices and also to develop tools which help the programmer fix or avoid *errors* when using them. With a *Traits error* analysis and tools for detecting these *errors* will help to introduce *Traits* use, to have a clearer view of *Traits* related concepts and to have a flatter learning curve.

Static Code Analysis tools are then useful in the *error* detection tools scope. These tools analyze syntactic and semantic source code features to detect problems without any user interaction. The importance of the automatic *error* detection tools is shown by the many main programming languages have one or more of such tools [StAnWeb].

The objectives of this thesis are:

- The identification of *errors* and *error types* in the use of *Traits*.
- The classification of those *traits errors* and *traits error types* into *categories* and *sub categories*.
- The development of an automatic *static code* checking tool for the programmer to detect the previously identified *traits errors* and *traits error types*.

Smalllint is an automatic *static code* checking tool inspired on **Lint**, which is implemented in several **Smalltalk** dialects. This tool defines a *rule* set, where each *rule* purpose is to detect a specific *error type*. For detecting *errors*, each *rule* is evaluated receiving as argument the *classes* or *compiled methods* to be checked. The *static code checking tool* to be developed in this thesis will be based on **Smalllint**, which is included in the standard **Pharo 1.0** image. **Smalllint** will be extended to, in addition to *classes* and *methods*, let it check *traits*, used *trait compositions* and *Traits* specific aspects.

This thesis is the complementary work of other *Traits* related works developed in our computer science department [G/07] [AB/07], helping to the introduction of *Traits*, improving the available tools and expanding the knowledge about using *Traits* as part of an *Object Oriented Language*.

1.1 Thesis Outline

This thesis is structured as follows:

Chapter 1 introduces the thesis background i.e. *traits* and *code analysis* models, its motivation and its evaluation against other available alternatives. Chapter 2 states the thesis goals, justifying the need to identify and typify *Traits errors* as well as the need to develop automatic tools to detect them. Chapter 3 presents and describes the *Traits error types* and their categorization. Chapter 4 describes the **Smalllint** extension and the *Traits error rules* implementation. Chapter 5 presents the results of running the adapted tool on several sample software projects. Concluding this work, Chapter 6 and 7 present conclusions and future works based on the experience collected during this thesis.

1.2 Traits

Traits is a new simple compositional model for structuring *object-oriented* programs. The purpose of *Traits* is to decompose *classes* into reusable building blocks by providing first-class representations of the different aspects (i.e. independent, but not necessarily cross cutting concerns) of the behavior of a *class* [SDNB/03]. For example, the aspect of being comparable is a concern of different entities like **Numbers**, **Dates**, **Weights** and others. Then, a shared *trait* between those entities would be a block defining the aspect of being comparable. Following this model, *Traits* enables a new programming style in which *traits* rather than *classes* are the primary units of reuse [BLAC/04].

1.2.1 Trait Motivation

Inheritance is the fundamental reuse mechanism in *object-oriented* programming languages, its most prominent variants are *Single Inheritance*, *Multiple Inheritance* and *Mixins Inheritance*. *Single Inheritance* is widely accepted as the *object-oriented* paradigm *sine qua non*, but it is also not expressive enough to factor out common features. To overcome *Single Inheritance* limitations, language designers have proposed various forms of *Multiple Inheritance*, as well as other mechanisms, such as *Mixins* that allow classes to be composed incrementally from sets of features. However these *inheritance* schemas also suffer from conceptual and practical reusability problems [DNSWB/06]. Following each *inheritance* schema and their limitations are described.

Single Inheritance

Inheritance in *object-oriented* languages is well established as an incremental modification mechanism that can be highly effective at enabling behavior reuse between similar *classes*. Unfortunately, *Single Inheritance* is inadequate for expressing classes that share features not inherited from their (unique) common parents [DNSWB/06].
Examples of such limitations appear at **Squeak Stream** *hierarchy*:

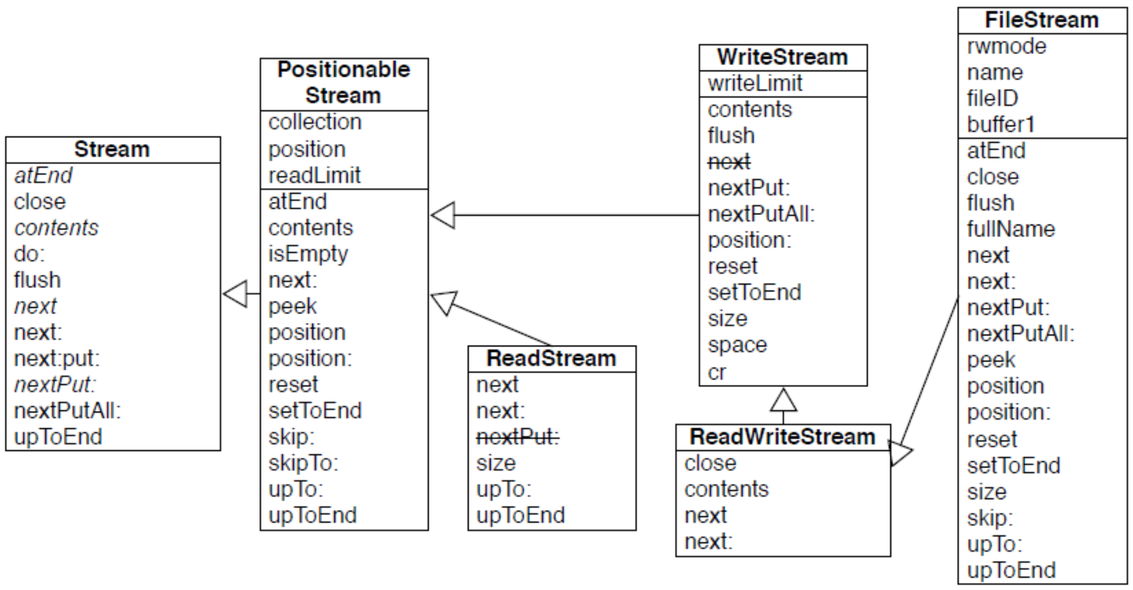


Figure 1: The **Squeak** core **Stream** hierarchy [CDW/07].

This **Stream** *class hierarchy*, implemented using *single inheritance* has the following problems:

Messages Implemented Too High in the Hierarchy

A common technique to avoid code duplication consists on implementing a *message* in the topmost common *superclass* of all *classes* which need this method. Even if efficient, this technique corrupts the interface of *classes* which do not need this *message* implementation. For example, **Stream** *class* defines **nextPutAll:** which calls **nextPut:**

```
Stream
nextPutAll: aCollection
       aCollection do: [:v| self nextPut: v].
^aCollection
```

Figure 2: **Stream** class **nextPutAll:** implementation calling abstract **nextPut:** [CW/07].

The *method* **nextPutAll:** writes all elements of the parameter **aCollection** to the stream by iterating over the collection and calling **nextPut:** for each element. The *message* **nextPut:** is abstract and must be implemented in *subclasses*, and even if **Stream** defines methods to write to the stream, some subclasses are used for read-only purposes, like **ReadStream**. Those *classes* must then explicitly cancel the *message* implementations they do not need. This approach, even if it was probably the best available solution in the first implementation, has some drawbacks. Firstly, **Stream** *class* and its *subclasses* are corrupted with a number of *message* implementations that are not available in the end. This situation makes more difficult understanding and/or extending the *hierarchy*. To add a new *subclass*, a developer must analyze all of the *messages* implemented in the *superclass* and cancel all the unwanted ones [CW/07].

Unused Superclass State

In the **Stream** *hierarchy*, **FileStream** *class* is a *subclass* of **ReadWriteStream** and an indirect *subclass* of **PositionableStream** which is explicitly implemented to stream over collections (see Figure 1). In this case, the instance variables ***collection***, ***position*** and ***readLimit*** inherited from the **PositionableStream** and ***writeLimit*** inherited from **WriteStream** are not used by **FileStream** nor any of its *subclasses* [CDW/07].

Simulating Multiple Inheritance by Copying

ReadWriteStream is conceptually both a **ReadStream** and a **WriteStream**. However, **Smalltalk** is a *single inheritance based language*, so **ReadWriteStream** has to choose between be implemented as a *subclass* of **ReadStream** or a *sublclass* of **WriteStream**. The behavior from the other *class* has to be copied, leading to code duplication and all of its related maintenance problems.
Squeak stream *hierarchy* designers decided to implement **ReadWriteStream** as a **WriteStream** *subclass*, and then copy the *methods* related to reading from **ReadStream**.
One of the copied *methods* is **next**, which reads and returns the next element in the stream. This leads to a strange situation where **next** is cancelled out in **WriteStream** (because it should not be doing any reading), only to be reintroduced by **ReadWriteStream**. The reason for this particular situation is the combination of **next** defined too high in the *hierarchy* and *single inheritance* [CDW/07].

Reimplementation

Figure 1 shows that **next:** is implemented five times. Not a single implementation sends *messages* to **super** which means that each *class* completely re-implements the *method* logic instead of specializing it. This statement should be tempered because often in **Squeak** stream *hierarchy*, *messages* override other *messages* to improve speed execution avoiding deep *hierarchy* searches in the *method lookup*. However, a re-implemented *message* in nearly all of the *classes* in a *hierarchy* implies the existence of *inheritance hierarchy* anomalies [CDW/07].

Multiple Inheritance

Multiple Inheritance enables a *class* to inherit features from more than one parent *class*, thus providing the benefits of better code reuse and more flexible modeling. However, *Multiple Inheritance* uses the notion of *class* in two contradictory roles, namely as the generator of instances and as the smallest unit of code reuse. This causes the problems and limitations that will be described next [SDNB/03].

Conflicting Features

One of the problems with *Multiple Inheritance* is the ambiguity that arises when conflicting features are inherited along different paths. A particularly problematic situation is the *"Diamond Problem"* (also known as *"Fork-Join Inheritance"*) that occurs when a *class* inherits from the same base *class* via multiple paths. Since *classes* are instance generators, they need to provide some

minimal behaviour (*e.g.*, implementations for *messages* **=**, **hash**, and **asString**), which is typically enforced by making them inherit from a common root *class* (*e.g.*, **Object**). However, this is precisely what causes the conflicts when several *classes* are reused [SDNB/03].
Conflicting features can be *conflicting message* implementations or *conflicting state variables*.

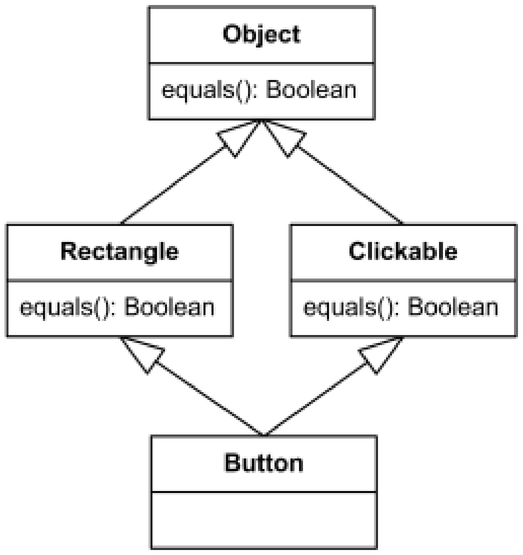


Figure 3: *Diamond Problem* example on **GUI** framework [DiProblWeb].
In the context of **GUI** software development, Figure 3 shows, a **Button** *class* that inherits from both **Rectangle** (for appearance) and **Clickable** (for functionality/input handling) *classes*, and both *classes* inherit from the **Object** *class*. In this example, both **Rectangle** and **Clickable** implements **equals** message. Considering this, in the case of an **equals** *message* sent to a **Button** *instance*, there is no predefined criteria to decide which **equals** *message* implementation should be the inherited one [DiProblWeb].

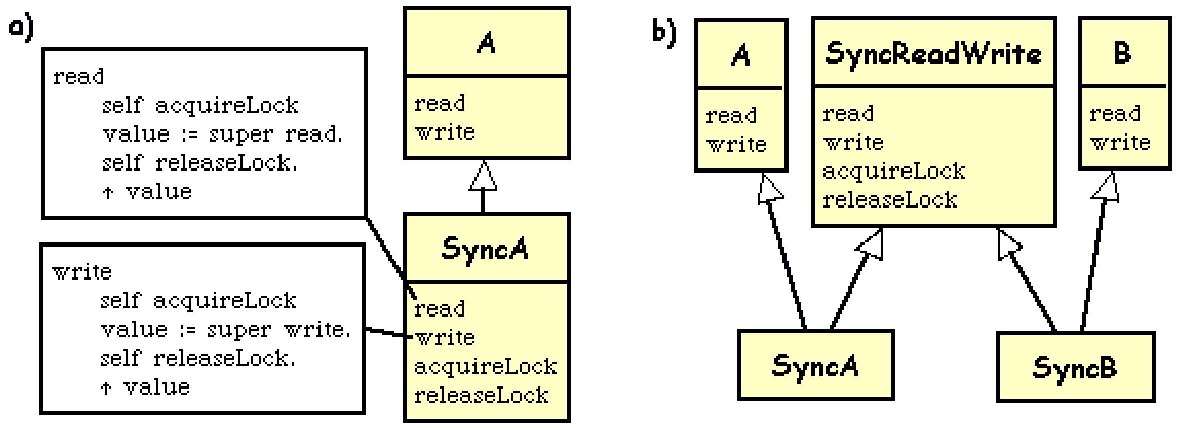
Whereas *message* implementation conflicts can be resolved relatively easily (e.g., by overriding), conflicting *state* is more problematic. Even if the declarations are consistent, it is not clear whether conflicting *state* should be *inherited* once or multiply [SDNB/03].

Accessing Overridden Features

Since identically named features can be inherited from different base *classes*, a single keyword (e.g., **super**) is not enough to access *inherited message* implementations unambiguously. For example, **C++** forces to explicitly name the *superclass* to access an overridden *message*. This leads to tangled *class* references in the source code and makes the code vulnerable to changes in the architecture of the *class hierarchy*. Explicit *superclass* references are avoided in languages, such as **CLOS**, that imposes a linear order on the *superclasses*. However, such a linearization often leads to unexpected behavior and violates *encapsulation*, because it may change the parent-child relationship among *classes* in the *inheritance hierarchy* [SDNB/03].

Limited Compositional Power (Factoring Out Generic Wrappers)

Multiple Inheritance allows a *class* to reuse features from multiple base *classes*. But unlike *Mixin Inheritance*, it does not allow writing a reusable entity that both uses and exports adapted forms of *messages* implemented in unrelated *classes*.



```
read
    self acquireLock
    value := super read.
    self releaseLock.
    ↑ value
```

```
write
    self acquireLock
    value := super write.
    self releaseLock.
    ↑ value
```

Figure 4: Limited compositional power of *multiple inheritance* examples [SDNB/03].

Figure 4 illustrates *multiple inheritance* limited compositional power. Assume that *class* **A** implements *messages* **read** and **write** that provide unsynchronized access to some data. If it becomes necessary to synchronize access, it can create a class **SyncA** that inherits from **A** and overrides the *messages* **read** and **write** so that they call the inherited implementation under control of a lock (Figure 4 a).

Now supposing that *class* **A** is part of a framework including another *class* **B**, implementing **read** and **write** *messages*, and that it is wanted to use the same technique to create a synchronized version of **B**. Naturally, it is wanted to factor out the synchronization code so that it can be reused in both **SyncA** and **SyncB**.

With *Multiple Inheritance*, the only way of sharing code among different classes is to inherit a common *superclass*. This means that the synchronization code has to be moved into a *class* **SyncReadWrite** that will become the *superclass* of both **SyncA** and **SyncB** (Figure 4 b). But a *superclass* cannot explicitly refer to a *message* like **read** that a possible *subclass* inherits from another *superclass*. It is possible to implicitly access such a *message* implementation, by calling an abstract *message* using a **self-send** that will eventually be implemented in the *subclass*. However, the whole point of this example is that unsynchronized reads are not and should not be available in **SyncA**. Thus, **SyncReadWrite** *class* cannot access the **read** and **write** *message* implementation provided by **A** and **B**, and it is not possible to factor out all the necessary synchronization code into **SyncReadWrite** [SDNB/03].

Mixin Inheritance

A *Mixin* is an *abstract subclass* specification that may be applied to various parent *classes* to extend them with the same set of features. *Mixins* allows the programmer to achieve better code reuse than *Single Inheritance* while maintaining the simplicity of the *inheritance* operation.

However, although *inheritance* works well for extending a *class* with a single orthogonal *mixin*, it does not work so well for extending a *class* from many *mixins*. The problem is that usually the *mixins* do not *quite* fit together, i.e., their features may conflict, and that *inheritance* is not expressive enough to solve such conflicts. Following several *Mixins* limitations are described [SDNB/03]:

Total Ordering

Mixins composition is linear i.e. all the *mixins* used by a *class* must be inherited one at time. *Mixins* composed after others override all the identically named features provided by previous *mixins*. While trying to resolve conflicts by selecting features from different *mixins*, it may be found that a suitable total order does not exist. As a consequence, with *Mixins*, the composite entity does not control which *mixins* are composed. The way in which the individual features override and extend one to another is imposed by the total ordering imposed on the *mixins*. Obtaining the desired combination of features may require introducing glue code in new intermediate *mixins*, or even modifying the component *mixins* [DNSWB/06].

Dispersal of Glue Code

Due to total ordering, the composite entity is not in full control of the way that the *mixins* are composed: the conflict resolution code must be hardwired in the intermediate *classes* that are created when the *mixins* are used, one at time. Obtaining the desired combination of features may require modifying the *mixins*, introducing new *mixins*, or, sometimes, using the same *mixin* twice.

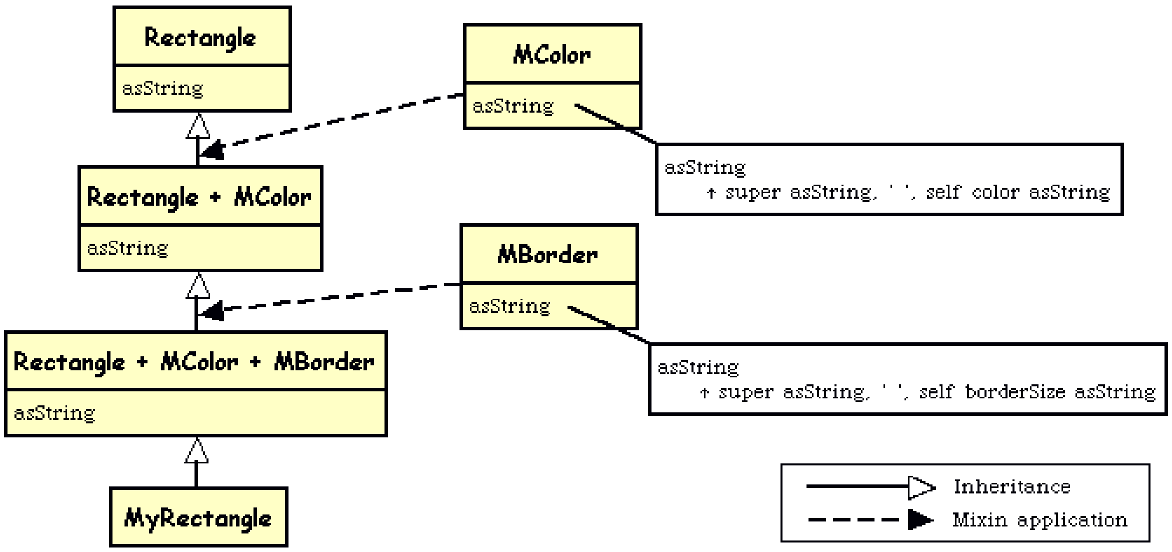


Figure 5: Dispersal of glue code due *Mixins* total ordering.

Figure 5 illustrates a dispersal of glue code example where *class* **MyRectangle** uses two *mixins* **MColor** and **MBorder** that both provide an ***asString*** *message*. In the *mixins* composition it can be chosen which of them should come first, but it cannot be specified how the different implementations of **asString** are glued together. This is because the *mixins* must be added one at time: in **Rectangle + MColor + MBorder** can be accessed the behaviour of **MBorder** and the mixed behaviour of **Rectangle + MColor**, but not the original behaviour of **MColor** and **Rectangle**. Thus, if it is wanted to adapt how the implementations of **asString** are composed, the involved *mixins* have to be modified [SDNB/03].

Fragile Hierarchies

Because of composition linearity and the limited means for solving conflicts, the use of multiple *mixins* result in *inheritance* chains that are fragile regarding changes. Adding a new *message* implementation to one of the *mixins* may silently override an identically named *message* of a *mixin* that appears earlier in the chain. Furthermore, it may be impossible to re-establish the original behaviour of the composite without adding or changing several *mixins* in the chain. This problem is especially critical if one modifies a *mixin* that is used in many places across the *class hierarchy* [SDNB/03].

1.2.2 Traits Model

This section introduces *Traits* model, its properties and its evaluation against the previously presented behavior sharing mechanisms limitations which motivated *Traits* creation.

Defining Traits

A *trait* essential objective is to be a first-class collection of named *methods*. *Methods* in a *trait* must be "*pure behaviour*"; they cannot directly reference any *instance variables*, although they can do so indirectly. *Traits* differ from *classes* in that they do not define any kind of state, and they rely on *composition* instead of *inheritance* as a behavior sharing mechanism.

Example:

In this example, some objects representing graphics are going to be constructed. Each graphical object can be decomposed into two aspects: its geometry, and how it is drawn on a canvas. Both aspects have been modeled by *traits*: **TCircle** trait which represents the geometry behavior and the **TDrawing** *trait* which represents the drawing behavior.
Following each *trait* is presented in two columns, the left column lists the *provided messages* and the right column lists the *required messages*.

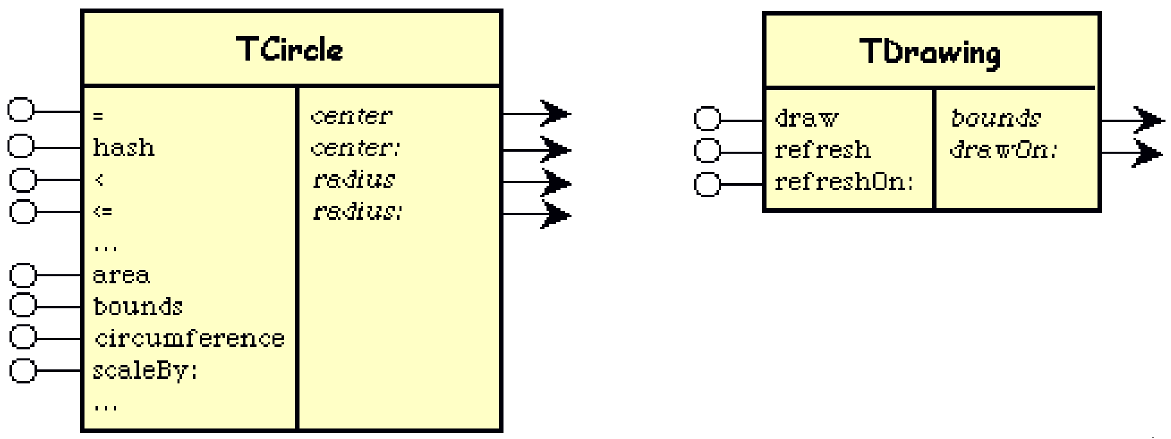


Figure 6: *Traits* examples with their *provided and required messages* [SDNB/03].

Figure 6 presents **TCircle** and **TDrawing** traits:

- **TCircle** *trait* contains (or provides) *messages* such as **area**, **bounds**, **circumference**, **scaleBy:**, **=**, **<**, and **<=** and requires *messages* **center**, **center:**, **radius**, and **radius:**, which parameterize its behavior.
- **TDrawing** *trait* provides **draw**, **refreshOn:**, and **refresh** *message* implementations, and is parameterized by **bounds** and **drawOn:** *required messages*.

`Trait named: #TDrawing uses: {}`	***Bounds*** `    self requirement`

`draw`
`    ^self drawOn: World canvas`	

`refresh`
`    ^self refreshOn: World canvas`
	drawOn: aCanvas
	`    self requirement`

`refreshOn: aCanvas`
`    aCanvas form`
`        deferUpdatesIn: self bounds`
`        while: [self drawOn: aCanvas]`	

Figure 7: TDrawing *trait* definition example [SDNB/03].

Figure 7 shows **TDrawing** implementation source code. In the implementation, a *required message* is declared by a *method* which **self-sends requirement** *message* (see **drawOn**: *message* implementation).

Composing Classes from Traits

Traits are completely backward compatible with *single inheritance* and are used to achieve structure and reusability within a *class* definition. This relationship is summarized with the equation:

Class = Superclass + State + Traits + Glue

Traits composition enjoys the *flattening property*. This property says that the semantics of a *class* defined using *Traits* is exactly the same as that of a *class* constructed directly from all the non-overridden *message* implementations provided by its used *traits*. This property enables the possibility of viewing a *class* as a flat collection of *message* implementations or as a composition of blocks with no change in its semantics.

Another property of *trait composition* is that the composition order is irrelevant, and hence conflicting *trait messages* must be explicitly disambiguated. Conflicts between *messages* defined in *classes* and *messages* defined by added *traits* are solved using the following two precedence rules:

- *Class messages* take precedence over *traits messages*.
- *Traits messages* take precedence over *superclass messages*. This follows from the *flattening property*, which states that *traits messages* behave as if they were defined in the *class* itself.

Example:

Circle *class* is composed by **TCircle** and **TDrawing** *traits*. **TDrawing>>bounds** and **TDrawing>>drawOn:** requirements are fulfilled by **TCircle** *trait* and **Circle** *class*. All the other requirements are fulfilled by *accessor messages* implemented by **Circle** *class*.

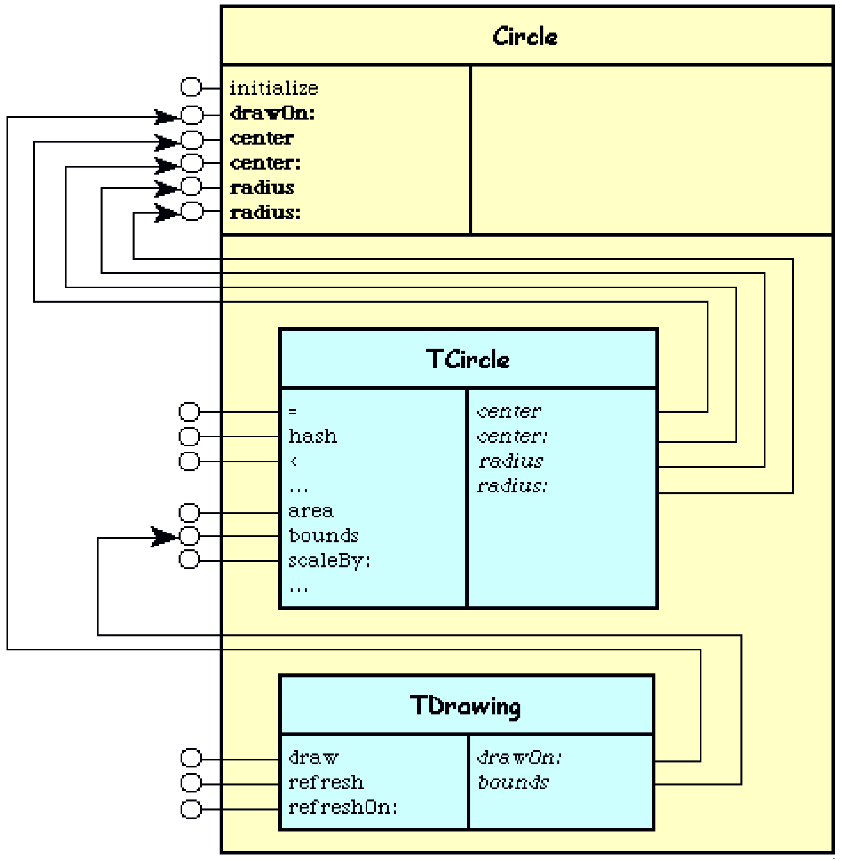


Figure 8: *Trait composition* into a *class* and provided/required message solving example [SDNB/03].

TDrawing *trait* requires **bounds** and **drawOn:** *messages*. **TCircle** *trait* provides a **bounds** *message implementation* which already fulfils one of the requirements. Therefore, **Circle** *class* has to provide only **center**, **center:**, **radius**, and **radius:** *messages* required by **TCircle** *trait* and **drawOn:** *message* required by **TDrawing** *trait*.

```
Object subclass: #Circle
       uses: TCirle + TDrawing
       instanceVariableNames: ’center radius’
```

```
Initialize
       center := 0@0.
       radius := 50
```

```
center
       ^center
```

```
radius
       ^radius
```

```
drawOn: aCanvas
       aCanvas fillOval: self bounds
              color: Color black
```

```
center: aPoint
       center := aPoint
```

```
radius: aNumber
       radius := aNumber
```

Figure 9: *Traits composition* into a *class* code example [SDNB/03].

center, **center:**, **radius**, and **radius:** *messages* are simply accessor *messages* to two instance variables. **drawOn:** *message* draws a circle on the canvas that is passed as the argument. In addition, **Circle** *class* also implements an **initialize** *message* to initialize the two instance variables.

Composite Traits

In the same way that *classes* are composed of *traits*, *traits* can be composed of other *traits*.

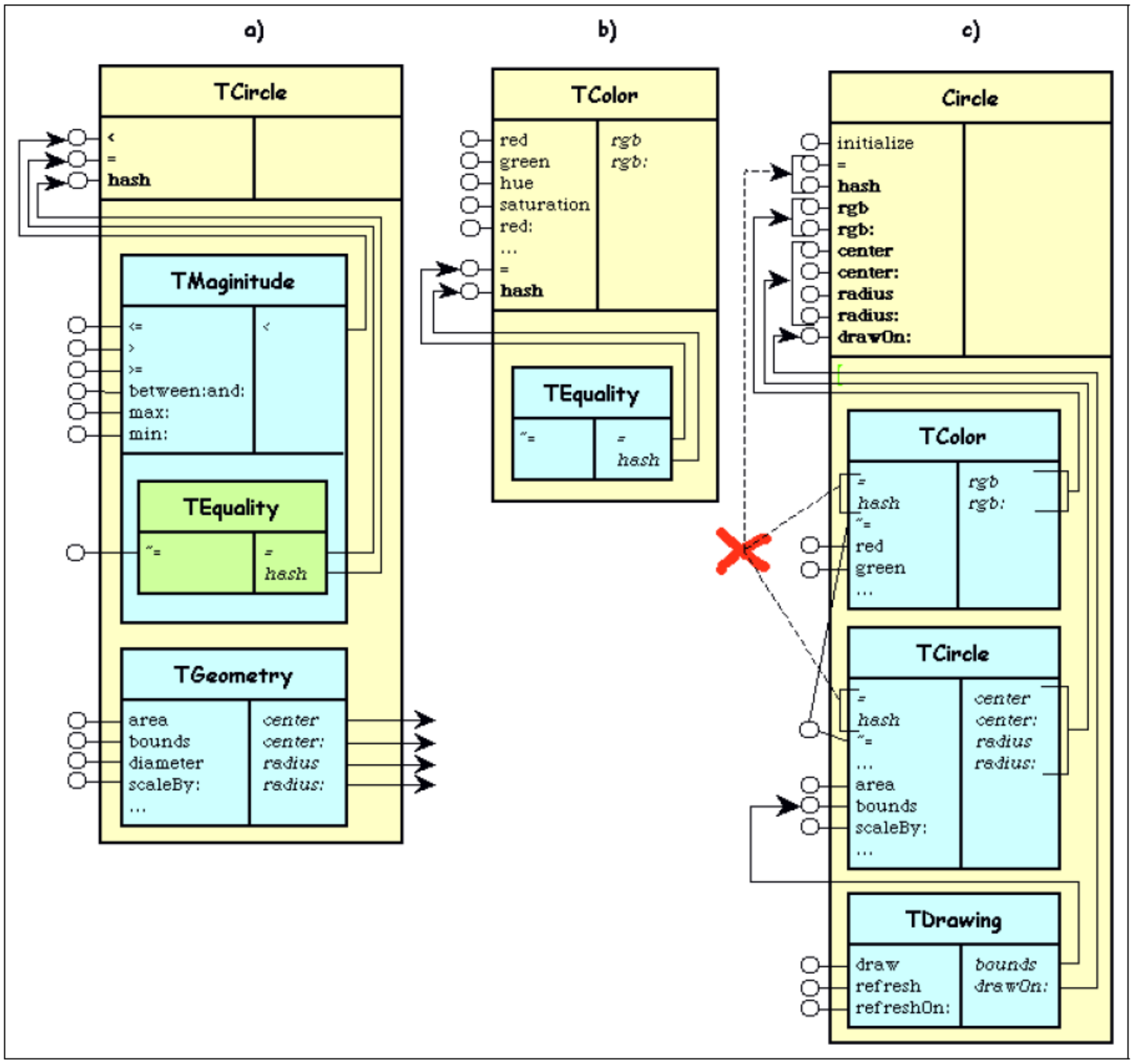


Figure 10: Composite *traits* and *composition conflict* resolution examples [SNDB/03].

Unlike classes, most *traits* do not have to be complete, which means that it is not mandatory to define all the *messages* that are required by their composing *traits*. Unsatisfied requirements of composing *traits* simply become *required messages* of the *composite trait*. Again, the composition order is not important and *messages* defined in the *composite trait* take precedence over the *messages* implemented by its composing *traits,* and, in case of multiple levels of composition, the *flattening property* remains valid.

Example

Figure 10 shows **TCircle** *trait* that contains two different aspects: namely comparison operators and geometric functions. Figure 10 a) shows how **TCircle** is redefined as the composition of **TMagnitude** and **TGeometry** *traits* in order to separate these aspects and improve the code reuse. Also **TMagnitude** *trait* is specified as a nested *trait*; it uses **TEquality** *trait* which *requires* **hash** and **=** *messages*, and provides **~=** *message*. **TMagnitude** *trait* itself requires **<**, and *provides messages* such as **max:**, **<=**, **between: and:**, and **>=**. Note that **TMagnitude** does not provide any of the *messages* required by its composing *trait* **TEquality**, which means that these requirements are just propagated as requirements of **TMagnitude**. Finally Figure 11 shows **TCircle** *trait which* is composed from **TMagnitude** and **TGeometry** *traits*. **TCircle** defines the *required messages* **=**, **hash**, and **<** for the *trait* **TMagnitude**. The first line of **TCircle** definition contains the *trait composition use clause*, which defines the *traits* composing the *trait* being defined.

```
Trait named: #TCircle
uses: TMagnitude + Tgeometry

= other                                     Hash
       ^self radius = other radius                ^self radius hash and: [self center
       and: [self center = other center]          hash]

< other
       ^self radius < other radius
```

Figure 11: TCircle *trait* definition code**.**

Trait Composition Conflict Resolution

A *trait composition conflict* arises if and only if two *traits* are composed providing identically named *messages* that are not originated in the same *trait*. In particular, this means that if the same *message* implementation (i.e. originated in the same *trait*) is obtained more than once via different paths, no conflict is produced.

Trait composition conflict must be explicitly solved by implementing a *message* in the *trait composition* client (can be a *class* or a *trait*). *Trait composition* also supports *message exclusion*, which lets the programmer avoid a conflict before it occurs.

To grant access to *conflicting messages*, *traits composition* support *message aliasing* operation. *Message aliasing* is used to make a *trait message* implementation available under another name; this is particularly useful if the original name is excluded by a *trait composition conflict*.

Example

To draw colored circles, a circle must contain color behavior. Figure 10 b) shows **TColor** *trait* definition, making reusable color behavior. This *trait* provides the usual color *messages* such as **red**, **green**, **saturation**, etc. Because colors can also be tested for equality, **TColor** uses **TEquality** *trait*, and implements the required *messages* **=** and **hash** as shown in Figure 12.

Trait named: #TColor **uses: Tequality**	
Hash **^self rgb hash**	**= other** **^self rgb = other rgb**

Figure 12: TColor implementing **TEquality** *required messages* [SDNB/03].

Figure 10 c) shows that when **TColor** *trait* is added to **Circle** *class*, a *composition conflict* arises because **TColor** and **TCircle** *traits* provide different implementations for **=** and **hash** *messages*. Note that **~=** *messages* does not give rise to a *trait composition conflict* because in both **TCircle** and **TColor** *traits* the implementation is created in the same *trait*, namely **TEquality**.
To solve the *composition conflicts*, one option is to redefine the conflicting *message* in the composition client. Figure 13 shows **Circle** *class trait composition use clause*, where **circleHash** and **circleEqual**: *alias messages* are defined for **TCircle>>hash** and **TCircle>>=** *messages*, and **colorHash** and **colorEqual**: *alias messages* are defined for **TColor>>hash** and **TColor>> =** messages. Then, after having alternative and non-conflicting *message* names, class **Circle** is able to redefine **=** and **hash** messages combining the original *messages* behavior.

Object subclass: #Circle instanceVariableNames: 'center radius rgb' uses: (TCircle @ {#circleHash -> #hash. #circleEqual: -> #=) + TDrawing + (TColor @ {#colorHash -> #hash. #colorEqual: -> #=)	
Hash ^self circleHash bitXor: self colorHash	= anObject ^(self circleEqual: anObject) and: [self colorEqual: anObject]

Figure 13: Conflict resolution through *message aliasing* and *traits composition* client message reimplementation [SNDB/03].

Alternatively, the *composition conflict* can be solved using only one of the *conflicting messages* implementation. For doing it, all the other *conflicting messages* must be excluded. Figure 14 shows how excluding **TColor>>=** and **hash** from the *used trait composition* avoids the *composition conflict*, providing **Circle** with **TCircle>>=** and **hash** *message* implementations [SNDB/03].

```
Object subclass: #Circle
instanceVariableNames: 'center radius rgb'
uses: TCircle + TDrawing + (TColor- {#=, #hash})
```

Figure 14: Conflict resolution through *message exclusion* [SNDB/03].

In this section are introduced the available *trait transformatio*ns for handling *traits provided messages*:

- **Message Aliasing Transformation**: A *message aliasing* transformation defines an alternative *message* name for a *trait provided message*. A *message aliasing* transformation is defined as **TransformedTrait @ {listOfAliasing}**, where **TransformedTrait** is the *trait* to be transformed and **{listOfAliasings}** is a list of *message aliasing mappings*. The *message aliasing mapping syntax* is **#newMessageName -> #oldMessageName,** where **#oldMessageName** is the *message* name originally provided by the transformed *trait*, and **#newMessageName** is the new *message* name for the original *message*.
- **Message Exclusion Transformation**: A *message exclusion* transformation removes a *message* from the list of a *trait provided messages*. *Message exclusion* is defined as

TransformedTrait – {listOfExclusions}, where **TransformedTrait** is the *trait* to be transformed and **{listOfExclusions}** is a list of *message* names to be excluded.

- **Message Rename:** This is not a real transformation, but could be considered a different transformation by itself. A *message rename* happens when a *trait provided message* is *excluded* and *aliased* at the same time. This is equivalent to changing the *message* name to a new one.

Traits Evaluation

This chapter explains how *Traits* overcome *Single*, *Multiple* and *Mixin Inheritance* limitations already described in the previous sections.

Single Inheritance

Traits does not replace *Inheritance*, it enables the definition of behaviour blocks that can be composed in any place of a *class hierarchy*, promoting a new model of *object oriented* programming. Under this new model, *classes* have the responsibility of instance creation, and *traits* have the responsibility of being the smallest unit of code reuse. In this way, the limitations of *Single Inheritance* can be bypassed using *Traits*, because the factored out behaviour can be attached directly to the desired *classes*, avoiding any kind of code duplication, reimplementation, or need to place it too high in the *hierarchy*.
Under *Single Inheritance* plus *Traits Model*, a *class* can be considered as:

Class = State + Traits + Glue Code

So, even if *Traits* is just about behaviour sharing, it tends to promote *classes* to define all its state independently from its behaviour, giving a more flexible way to handle unused *superclass* state problem [SDNB/03].

Multiple Inheritance

This section presents the evaluation of *Traits* against *Multiple Inheritance* limitations.

Conflicting Features

Since *traits composition* supports composing several *traits* in parallel, conflicting features are also an issue. However, the problem is less serious with *Traits*. *Traits* cannot define state, so the "*Diamond Problem"* does not arise. Although a *class* may obtain the same *message* implementation from the same *trait* via multiple paths, these multiple copies do not give rise to a *composition conflict*, and will therefore be unified [SDNB/02].

Accessing Overridden Features

With *Traits*, regarding to access overridden features, it was decided not to take approaches based on naming the *superclass*/*trait* in the source code of the *methods*. Instead, it was decided to use a simple form of *message aliasing*. This avoids both tangled *class* references in the source code and code that is hard to understand and fragile with respect to changes. *Message aliasing* also allows accessing conflicting *messages* under non conflicting *message* names [SDNB/02].

Limited Compositional Power

Like *Mixins*, *Traits* can explicitly refer to a *messages* implemented by the *superclass* of the *class* that uses the *trait*. Considering this, the presented "*Limited Compositional Power*" problem (Figure 4) can be solved by implementing the synchronization *messages* **read**, **write**, **acquireLock**, and **releaseLock** in a reusable *trait*. This *trait* is then used in both **SyncA** and **SyncB** *classes*, which do not need to implement any *message* other than accessors for the **lock** variable [SDNB/02].

Mixin Inheritance

This section presents the evaluation of *Traits* against *Mixin Inheritance* limitations.

Total Ordering

Trait composition is symmetric and does not impose total ordering, but it can express ordering by means of nesting. In addition, *trait composition* can be combined with *inheritance* which allows a wide variety of partially ordered compositions [SDNB/02].

Dispersal of Glue Code

When *traits* are combined, the glue code is always located in the combining entity, reflecting the idea that the superordinate entity is responsible of plugging together the components that implement its aspects. This property nicely separates the glue code from the code that implements the different aspects. This makes a class easier to understand, even if it is composed from many different components [SDNB/02].

Fragile Hierarchies

Since *traits* are designed to be used in many different *classes*, robustness with respect to change has been a leading principle in designing trait composition. In particular, *Traits* require every *message conflict* to be explicitly solved. The consequence is that solving conflicts require some extra work, but it is also that the behaviour of the composite is what the programmer expects.
In addition, any problem caused by changes to a *trait* is limited to the direct user of that *trait*, whether that is a *class* or a *composite trait*. This is because the *trait client* always controls how the components are plugged together. With *Traits*, change is localized: a single change in a component requires at most one compensating change in each direct user of the component in order to re-establish the original behavior [SDNB/02].

1.2.3 Conclusions about Traits

Traits is proposed as the primitive unit of code reuse, using *composition* instead *inheritance* as mechanism of behaviour sharing. *Traits* extends *Single Inheritance* and offers a behaviour sharing model that overcome limitations of different variants of *inheritance* but without losing any of the desired *Single Inheritance* properties.

Traits model has the following properties:

- Two responsibilities are clearly separated: *traits* are purely units of reuse, and *classes* are generators of instances.
- *Traits* specify no state (do not have internal collaborators), so the only conflict that can arise when combining *traits* is a *message* name *composition conflict*. Such a conflict can be solved by *overriding* or by *message exclusion*.
- *Traits* are simple software components that both *provide* and *require messages* (*required messages* are those ones that are used, but not implemented by a *trait*).
- *Classes* are composed of *traits*. In the composition process *trait composition* conflicts must be explicitly solved, and *traits required messages* can possibly be provided.
- *Traits* can be inlined, a process that is called "*flattening*": the fact that a *message* is implemented in a *trait* does not affect its semantics i.e. it is the same to implement a *message* in a *trait* than directly on its clients (its clients can be either *traits* or *classes*).
- Problems with *Multiple Inheritance* disappear with *Traits*, because *Traits* do not rely on the *inheritance hierarchy*.
- Problems with *Mixins* also disappear, because *Traits* impose no composition order.

[DNSWB/06].

1.3 Code Analysis

Code Analysis is the process of (semi)automatically analyzing the behavior of computer programs. The two main approaches in *code analysis* are *static* and *dynamic code analysis*. Some of the most important *code analysis* applications are *program correctness* and *program optimization* [CodAnWeb].

1.3.1 Dynamic Code Analysis

Dynamic code (or *program*) *analysis* implies the execution of the code, in a real or virtual processor. To make *dynamic code analysis* effective, the target program must be executed with enough test inputs to produce interesting behavior. The *dynamic code analysis* is focused on analyzing the behavior of a program; regardless on which component generates the behavior (or misbehaviour). It is worth to note that the user should usually define tests inputs to execute an effective analysis of the code (there are tools like code coverage that helps programmer to produce effective test input sets for the analyzed code) [DynCodAnWeb].

1.3.2 Static Code Analysis

Static code analysis is the analysis of code that is performed without actually executing the code. The complexity of the analysis performed by tools varies from those ones that only consider the behavior of individual statements and declarations, to those ones that include the complete source code of a program in their analysis. Uses of the information obtained from the analysis vary from highlighting possible coding errors (e.g., the **Lint** tool) to formal methods that mathematically prove properties about a given program (e.g., the behavior matches its specification). The *static code analysis* is based on language properties, like its *syntax* and/or its *semantics* (e.g. *denotational semantics*, *axiomatic semantics*, *operational semantics*, and *abstract interpretation*) [StCodAnWeb].

1.3.3 Dynamic vs. Static Code Analysis

This thesis puts the focus on full automatic tools to detect *trait* related *typified errors*. The mentioned goal and the need of applying *code analysis* to *Traits* related elements leads to select the technique of *Static code analysis*. It also should be noted that, since *static code analysis* is based on language features, it can analyze already existing programs focusing on specific *traits related errors*. *Static code analysis* will be able to perform an effective analysis, even in situations where there is not enough information to perform an effective *dynamic code analysis* e.g. when there is lack of knowledge about the program behavior and/or lack of input sets for running dynamic tests.

1.3.4 Static Code Analysis Tools

The first automatic *static code analysis* tool was **Lint**, released to the public in the seventh version (**V7**) of the **UNIX** operating system in 1979. This tool detects syntactically correct code, but which could have portability problems moving to different compilers, wasteful or *error* prone constructions which nevertheless are, strictly speaking, legal. Some of the issues analyzed by **Lint**, are undecidable problems (e.g. deciding whether *exit* is ever called is equivalent to solve the "*halting problem*"). Due this problem, most of the **Lint** algorithms are a compromise, being possible to miss some *errors* and also flagging false positive results [J/77].

Currently, **Lint** is the generic name applied to *static code analysis* tools for detecting *errors* and suspicious construction on code of a given programming language.

Many of the most important programming languages use **Lint** like *static code analysis* tools. Some examples of this are: **JLint** on **Java**, **Splint** on **C/C++**, **SmallLint** on **Smalltalk** and many others [StAnWeb].

2 Trait Error Typification and Automatic Trait Validation

This thesis tackles the identification of *Traits* related *errors*, their typification and categorization depending on which *Traits* aspect or characteristic is involved in the *error* occurrence, and the adaptation of a *static code analysis* tool for detecting the identified *errors* on programs code.

Traits is a new behavior sharing mechanism different than *inheritance*. As any new construction, its use can introduce new kinds of *errors*, which should be identified and typified to be detected when *Traits* are used. For an *error* to be a *Traits* related *error*, it has to be produced by a *Traits* specific aspect e.g. a *trait composition clause*, a *message exclusion* transformation or any other *Traits* aspect.

Error groups or *categories* can be identified by using the *Trait characteristic* which produces the *error* as the common denominator identifying each group. Identifying *Traits error types* and *categories* is a necessary step for adapting *static code analysis* tools to detect them.

Currently, implementations of *Traits* are available in **Smalltalk** dialects like **Squeak** and **Pharo**. On these programming languages, an important *static code analysis* tool is **Smalllint** which is not capable of analyzing and detecting *Traits* related *errors*. One possible reason for this is that there is not a *Traits* related *error* typification available yet, and because the flattening property, *classes* composed with *traits* can be considered as standalone *classes*. Because of these issues, the *static code analysis* tools can work on composed *classes* as usual, without considering the composition of *traits* on them.

The *traits error typification* means an extension of *Traits* knowledge because it implies the study of possible use cases and variations, considers *syntactic* and *semantic* aspects of *Traits* and also sets a starting point to detect problems and weak points on *Traits* use. Thus, it enables the development of new, more useful and reliable *Traits* versions. Categorizing *traits error types* depending on which *Traits* aspect or characteristic produces an error is useful, since it groups the *error types* with unambiguous criteria and sets a more abstract perspective on *Traits errors* (i.e. you do not have to remember all the individual *errors types*, but just remember a few more abstract *Traits error categories*). This classification eases the detection of non-analyzed *Traits* aspects and the definition of new *Traits error categories*. It also helps to extend the *Traits error* categories by adding new non-typified *Traits errors*, since analyzing a *Traits error category* aspect narrows the *Traits error* domain under study.

This proposed definition of *Traits error types* and *categories* implicitly define the *Traits* aspects associated with each *Traits error category* and their relevance on *error* generation, helping thus to acquire *Traits* related concepts, like *message exclusion*, *traits composition* and others.

A *static code analysis* tool (i.e. **Smalllint**) adapted to check *Traits* related *errors* will help on *Traits* introduction, use and use correctness, which is especially useful for new programmers. This kind of tool will help to avoid unnecessary and preventable *errors*, to the introduction of best practices and to refactor already existing programs using *Traits*.

3 Traits Error Types and Categorization

This chapter shows the identified *Traits error types* and *categories*. Each *Traits error type* defines a specific problem on *Traits* use. A *Traits error type* could be a proper *error* which is wrong under any possible scenario, or a warning for those *errors* that could be perfectly valid and correct uses in some scenarios, but still not recommended.

The categories group *Traits error types* by the *Traits* aspect or characteristic where the *error* is generated, like in the *trait composition clause*, in a *trait transformation*, in the *trait* use by a *class* or by other *trait* or any other aspect.

Section 3.1 shows the *Traits error category* and *sub category* hierarchy, and the included *Traits error types*. Next, all the *Categories* and *Sub Categories* are listed, including their respective descriptions.

Section 3.2 lists all the identified *Traits error types*, including the *Category*/*Sub Category* to which each of them belongs, its *trait error* description and an example of the *error*.

3.1 Categories and Sub Categories

All the identified *errors* in this thesis are specific to *Traits*, i.e. there is a *Traits* aspect where the *error* is generated. All the identified *Traits error types* can be organized in an unambiguous manner considering the *Traits* aspect where the *error* is produced. Next in this section the *Category/Sub Category* organization for the identified *Traits error types* is presented.

Category	Sub Category	Error Type
Transformations	Traits Transformation Consistency	Switched Message Aliasing
		Undefined Aliased Message
		Equals New and Old Message Name Aliasing
		Aliasing Collision
		Undefined Excluded Message
		Empty Trait Transformation Set
		Already Defined Alias Message
		Duplicated Alias Messages
		Excluded Alias Message
		Duplicated Trait Transformation Definition
		Invalid Message Exclusion Set
		Invalid Message Aliasing Set
		Chained Message Aliasing
	Message Rename	Unimplemented Self-Sent Message due Message Renaming
	Message Exclusion	Unimplemented Self-Sent Message due Message Exclusion
		Excluded and Not Provided Explicit Required Message
		Always Excluded Message
		Too Many Excluded Messages
	Trait Transformation Meta-level Error	Misplaced Meta-level Instance Message Aliasing
		Misplaced Meta-level Instance Message Exclusion
		Misplaced Meta-level Class Message Aliasing
		Misplaced Meta-level Class Message Exclusion
Composition	Trait Composition Conflict	Trait Composition Conflict Method
		Trait Composition Conflict due Aliasing
	Unnecessary Trait Transformation	Unnecessary Message Exclusion
		Unnecessary Message Aliasing
Traits Use	Message Overriding	Override with Identical Method
		Overridden Aliasing
		Always Overridden Message
		Too Many Overridden Messages
	Required Messages	Unimplemented Required Messages
		Hidden Implementation by Explicitly Required Message
		Not Explicitly Declared Required Message
		Unused Required Message
	Super Sent Messages	Super-Sent Message Lookup Bypasses Used Trait Composition Provided Message
		Trait Method Super-Sends a Message
Best Practices	Trait Definition and Use	Unused Trait
		Traits Names doesn't Start with T

Figure 15: *Category* and *Sub Category hierarchy* and the *Traits error types* included on them.

A *category* is defined by the *Traits aspect* where the *Traits error* is produced, and a *sub category* is defined by an aspect of its parent *category*. The *Traits error types* are included in the *category*/*sub category* corresponding to the aspect where the *error* is produced.

3.1.1 Categories and Sub Categories Description

This section introduces the *categories* and *sub categories* that classify the identified *Traits error types*. Each *Category* is named as "*Category: Category Name*" and each *Sub Category* as "*Sub Category: Sub Category Name*".

Category: Transformations

Description: These *errors* happen when a *trait transformation* is applied to a single *trait* in a *trait composition use clause*.

Sub Category: Traits Transformation Consistency

Description: These *errors* occurs when a *trait transformation* is defined without following the operation preconditions, like applying *message aliasing* on non-existing *messages*, or defining a *trait transformation clause* which does not follow the expected syntax.

Sub Category: Message Rename

Description: These *errors* are related to the use of a *message rename* in the *trait composition use clause*.

Sub Category: Message Exclusion

Description: These *errors* are related to the use of *message exclusion* in the *trait composition use clause*.

Sub Category: Trait Transformation Meta-level Error

Description: These *errors* occur when a *trait transformation* suitable for an *instance message* is applied to a *class message* or vice versa i.e. applying a *trait transformation* valid for a *trait* to its *classTrait* or applying a *trait transformation* valid for a *classTrait* to its corresponding *trait*.

Category: Composition

Description: These *errors* are produced in a *trait composition use clause* because *trait composition conflicts* or unnecessary use of *trait transformations*.

Sub Category: Trait Composition Conflict

Description: These *errors* occur when a *trait composition use clause* defines a *trait composition conflict*, i.e. two *traits* or *trait transformations* in the *trait composition use clause* provides the same message to the *trait composition* and its *trait composition client* does not solve it, resulting on a *trait composition conflict message* provided to the *trait composition client*.

Sub Category: Unnecessary Trait Transformation

Description: These *errors* occur when an unnecessary *trait transformation* is applied on the *trait composition use clause*.

Category: Traits Use

Description: These *errors* are related to the way that *trait compositions* are used by their *trait composition clients*.

Sub Category: Message Overriding

Description: These *errors* occur when a *trait composition client* overrides a *trait composition provided message*.

Sub Category: Required Messages

Description: These *errors* are related to the *used trait composition implicit* or *explicit required messages*.

Sub Category: Super-Sent Messages

Description: These *errors* are related to **super-sent** *messages* from or to *trait composition provided messages*.

Category: Best Practices

Description: It is a heterogeneous *error category* which includes *Traits* related community practices and recommendations.

Sub Category: Trait Definition and Use

Description: These *errors* are related with *trait* definition and general use.

3.2 Traits Error Types

This section lists all the identified *traits error types*. Each *Traits error type* description indicates its *category* and *sub category*, its *Traits error type* definition, and usually a code example showing an occurrence of the described *error*.

3.2.1 Error Type: Switched Message Aliasing

Category: Transformations

Sub Category: Traits Transformation Consistency

Error Type Definition: This *traits error* happens when a *message aliasing* is applied to a *trait*, but switching the message order in the *message aliasing mapping* i.e. the *message aliasing mapping* is defined as **messageOld -> messageNew** instead of the expected **messageNew -> messageOld** syntax. When this happens, it is an attempt to map an undefined message (**messageNew** is not defined in the *trait* since it should be a new message name) to an already existing *trait message* (**messageOld** is the *trait provided message* expected to be aliased). This *error* could be considered an *"Undefined Aliased Message"* and/or an *"Already Defined Alias Message" error*, but when both errors happen together they configure a new *trait error type*.

Example:

```
Trait named: #TExampleTrait uses: {}
```

```
m1
        ^self aMessage.
```

```
Object subclass: #ExampleClass
        uses: TExampleTrait @ {#m1 -> #m2}
        instanceVariableNames: ''
        classVariableNames: ''
```

Figure 16: An example code showing a *message aliasing mapping* with the *message* order switched.

In this example, **ExampleClass** defines the *aliasing mapping* **m1 -> m2** on **TExampleTrait**, but **ExampleTrait** does not define **m2** *message* at all. In this case, if the *message aliasing mapping* were defined as **m2 -> m1** (i.e. switching the *message aliasing mapping* order) it would be a valid mapping. We could infer that the programmer really wanted to define the valid mapping but switched the right mapped messages order.

3.2.2 Error Type: Undefined Aliased Message

Category: Transformations.

Sub Category: Traits Transformation Consistency.

Error Type Definition: This *traits error* happens when a *message aliasing mapping* **newMessage -> oldMessage** is applied to a *trait* that does not define **oldMessage**. This means that there is an attempt to define a *message aliasing* for a non-existing *message*.

Example:

```
Trait named: #TExampleTrait uses: {}
```

```
m1
       ^self aMessage.
```

```
Object subclass: #ExampleClass
       uses: TExampleTrait @ {#m3 -> #m2}
       instanceVariableNames: ''
       classVariableNames: ''
```

Figure 17: *Undefined aliased message* example code.

In this example, **ExampleClass** defines a *message aliasing mapping* **m3 -> m2** for **TExampleTrait**, this means **m3** is an *alias* for **m2**, but **TExampleTrait** does not define any **m2** *message*.

3.2.3 Error Type: Equals New and Old Message Name Aliasing

Category: Transformations.

Sub Category: Traits Transformation Consistency.

Error Type Definition: This *traits error* happens when a *message aliasing mapping* **aMessage -> aMessage** is defined in a *trait composition use clause*. In this case the alias and the aliased message are the same *message*, making this *message aliasing transformation* a null result operation. This *error type* overlaps *"Undefined Aliased Message"* when the *aliased message* is not defined by the *trait*, and overlaps *"Already Defined Alias Message"* when the *aliased message* is defined by it.

Currently, **Pharo Smalltalk** *Traits* implementation avoids this *Traits error type* occurrence.

Examples:

1)

```
Trait named: #TExampleTrait uses: {}
m1
       ^self aMessage.
```

```
Object subclass: #ExampleClass
       uses: TExampleTrait @ {#m1 -> #m1}
       instanceVariableNames: ''
       classVariableNames: ''
```

Figure 18: An example code showing a *trait provided message* aliased to the same *message* name.

In this example, **ExampleClass** defines a *message aliasing transformation* from **m1** to **m1**. Despite of **TExampleTrait** defines **m1**, the *alias message* is the same as the original one, making this a null transformation.

2)

```
Trait named: #TExampleTrait uses: {}
m1
       ^self aMessage.
```

```
Object subclass: #ExampleClass
       uses: TExampleTrait @ {#m2 -> #m2}
       instanceVariableNames: ''
       classVariableNames: ''
```

Figure 19: An example code showing a non-existing *message* aliased to the same *message* name.

This example is similar to the example 1) with the difference that the *aliased message* is not defined by **TExampleTrait**.

3.2.4 Error Type: Aliasing Collision

Category: Transformations.

Sub Category: Traits Transformation Consistency.

Error Type Definition: This *traits error* happens when, in a *trait composition use clause*, there are two *message aliasing mappings* **aliasMessage -> originalMessage1** and **aliasMessage -> originalMessage2** defining the same alias message for two different messages. These two *messages* aliased to the same alias *message* name is a *trait error* since it is not clear which *message* implementation *method* should be evaluated in case of receiving **aliasMessage**.

Example:

```
Trait named: #TExampleTrait uses: {}
m1
       ^self aMessage
m2
       ^self  anotherMessage
```

```
Object subclass: #ExampleClass
       uses: TExampleTrait @ {(#m3 -> #m1)
       (#m3 -> #m2)}
       instanceVariableNames: ''
       classVariableNames: ''
```

Figure 20: Example of multiple *messages* aliased to the same *message* name.

In this example, **ExampleClass** defines the *message aliasing mappings* **m3 -> m1** and **m3 -> m2** applied to **TExampleTrait**. Because of it, **m3** is an alias for both **m1** and **m2** which is an inconsistent *message aliasing mapping* definition since an *alias message* should refer to just one *aliased message*.

3.2.5 Error Type: Undefined Excluded Message

Category: Transformations.

Sub Category: Traits Transformation Consistency.

Error Type Definition: This *traits error* happens when a *trait composition use clause*, there is a *message exclusion* for a *message* not defined in the transformed *trait*. This *trait transformation* has no effect on the transformed *trait* and adds unnecessary complexity to the *trait composition use clause*.

Example:

```
Trait named: #TExampleTrait uses: {}
m1
        ^self aMessage.
```

```
Object subclass: #ExampleClass
        uses: TExampleTrait - {#m2}
        instanceVariableNames: ''
        classVariableNames: ''
```

Figure 21: Example code of a *message exclusion* of an undefined *message*.

In this example, **ExampleClass** defines a *message exclusion* for **m2** on **TExampleTrait**, which does not define the excluded *message*.

3.2.6 Error Type: Empty Trait Transformation Set

Category: Transformations.

Sub Category: Traits Transformation Consistency.

Error Type Definition: This *traits error* happens when a *trait composition use clause* includes a *trait transformation* with an empty *trait transformation set*. If the transformation set is empty it will have no effect on the transformed *trait*, adding unnecessary complexity to the *trait composition use clause*.

Examples:

1)

```
Trait named: #TExampleTrait uses: {}
m1
        ^self aMessage.
```

```
Object subclass: #ExampleClass
        uses: TExampleTrait @ { }
        instanceVariableNames: ''
        classVariableNames: ''
```

Figure 22: Empty *message aliasing* set example code.

In this example, **ExampleClass** defines an empty *message aliasing* mapping set for **TExampleTrait**.

2)

```
Trait named: #TExampleTrait uses: {}
m1
        ^self aMessage.
```

```
Object subclass: #ExampleClass
        uses: TExampleTrait - { }
        instanceVariableNames: ''
        classVariableNames: ''
```

Figure 23: Empty *message exclusion* set example code.

In this example, **ExampleClass** defines an empty *message exclusion* set for **TExampleTrait**.

3.2.7 Error Type: Already Defined Alias Message

Category: Transformations.

Sub Category: Traits Transformation Consistency.

Error Type Definition: This *traits error* happens when a *trait composition use clause* defines a *message aliasing mapping* **newMessage -> oldMessage** but the transformed *trait* defines **newMessage** too. In this case, it is not clear if the *message implementation* for **newMessage** will be the *method* associated with the *trait* defined **newMessage** message or with the aliased **oldMessage**.

Example:

```
Trait named: #TExampleTrait uses: {}
m1
        ^self aMessage.
m2
        ^self anotherMessage
```

```
Object subclass: #ExampleClass
        uses: TExampleTrait @ {#m2 -> #m1}
        instanceVariableNames: ''
        classVariableNames: ''
```

Figure 24: Example code showing a *message aliasing* defining an alias name which is already defined in the transformed *trait*.

In this example, **ExampleClass** defines the *message aliasing mapping* **m2 -> m1** applied to **TExampleTrait** which already defines **m2** *message*. Thus, it is not clear if the implementation for **m2** at **ExampleClass** will be **TExampleTrait>>m1** or **TExampleTrait>>m2** *method*.

3.2.8 Error Type: Duplicated Alias Messages

Category: Transformations.

Sub Category: Traits Transformation Consistency.

Error Type Definition: This *traits error* happens when a *trait transformation* defines two *message aliasing mappings* **newMessage1 -> oldMessage** and **newMessage2 -> oldMessage** i.e. it defines two different *alias messages* for the same aliased *message*. Strictly speaking, this is not a *trait error*, but having two *alias messages* with the same implementation is a suspicious issue to consider.

Example:

```
Trait named: #TExampleTrait1 uses: {}
```
```
m1
       ^self aMessage.
```
```
Trait named: #TExampleTrait2 uses: {}
```
```
m1
       ^self anotherMessage
```
```
Object subclass: #ExampleClass
       uses: TExampleTrait1 @ {(#m2 ->
       #m1)(#m3 -> #m1)} - {#m1} +
       TExampleTrait2
       instanceVariableNames: ''
       classVariableNames: ''
```

Figure 25: Example code showing the same *message* aliased to two different *alias messages*.

In this example, **ExampleClass** *used trait composition* defines two *message aliasing mappings* **m2 -> m1** and **m3 ->m1** applied to **TExampleTrait1** and excludes **m1** message from the *trait composition*, avoiding a *trait composition conflict* with **m1** provided by **TExampleTrait2**. If the intention was aliasing **TExampleTrait1>>m1** *message* to avoid losing its associated *method* at **ExampleClass**, it would be enough with defining just one *message aliasing mapping*.

3.2.9 Error Type: Excluded Alias Message

Category: Transformations.

Sub Category: Traits Transformation Consistency.

Error Type Definition: This *traits error* happens when a *trait composition use clause* applies to the same *trait* a *message aliasing mapping* **newMessage -> oldMessage** and a *message exclusion* on **newMessage** i.e. a *message exclusion* removes an *alias message*. Defining these transformations have no effect since the added *alias message* is removed by the *message exclusion*, adding unnecessary complexity to the *trait composition use clause*.

Example:

```
Trait named: #TExampleTrait uses: {}
```

```
m1
        ^self aMessage.
```

```
Object subclass: #ExampleClass
        uses: TExampleTrait @ {#m2 -> #m1} -
        {#m2}
        instanceVariableNames: ''
        classVariableNames: ''
```

Figure 26: Example code shows an *alias message* which is excluded in the same *trait transformation*.

In this example, **ExampleClass** *trait composition use clause* applies to **TExampleTrait** a *message aliasing mapping* **m2 -> m1** and a *message exclusion* for **m2**, which is provided by the *message aliasing*. It would be same result using **TExampleTrait** with no *trait transformations* applied at all.

3.2.10 Error Type: Duplicated Trait Transformation Definition

Category: Transformations.

Sub Category: Traits Transformation Consistency.

Error Type Definition: This *traits error* happens when a *trait composition use clause* defines a duplicated *trait transformation* applied to the same *trait*. Defining the same *trait transformation* more than one time has no difference than defining the transformation only once.

Currently **Pharo Smalltalk** *Traits* implementation avoids this *Traits error type* occurrence.

Examples:

1)

```
Trait named: #TExampleTrait uses: {}
```

```
m1
       ^self aMessage.
```

```
Object subclass: #ExampleClass
       uses: TExampleTrait @ {(#m2 ->
       #m1)(#m2 -> #m1)}
       instanceVariableNames: ''
       classVariableNames: ''
```

Figure 27: Example code showing a *message aliasing mapping* defined twice on the same *trait*.

In this example, **ExampleClass** defines the *message aliasing mapping* **m2 -> m1** applied to **TExampleTrait** two times.

2)

```
Trait named: #TExampleTrait uses: {}
```

```
m1
       ^self aMessage.
```

```
Object subclass: #ExampleClass
       uses: TExampleTrait - {#m1.#m1}
       instanceVariableNames: ''
       classVariableNames: ''
```

Figure 28: Example code showing a *message exclusion* defined twice on the same *trait*.

In this example, **ExampleClass** defines the *message exclusion* for **m1** applied to **TExampleTrait** two times.

3.2.11 Error Type: Invalid Message Exclusion Set

Category: Transformations.

Sub Category: Traits Transformation Consistency.

Error Type Definition: This *traits error* happens when a *trait composition use clause* defines a *message exclusion* set containing a *message aliasing mapping*. The *message exclusion* set must contain *message* selectors. Containing any other kind of entity is an *error* because it does not represent a *message* selector to be excluded. One possible reason for this *error* is that the programmer was actually trying to define a *message aliasing* instead of a *message exclusion*.

Example:

```
Trait named: #TExampleTrait uses: {}
```

```
m1
        ^self aMessage.
```

```
Object subclass: #ExampleClass
        uses: TExampleTrait - {#m2 -> #m1}
        instanceVariableNames: ''
        classVariableNames: ''
```

Figure 29: Example code showing a *message aliasing mapping* in the *message exclusion* definition set.

In this example, **ExampleClass** defines a *message exclusion* set including **m2 -> m1** *message aliasing mapping* instead of containing just message selectors to be excluded.

3.2.12 Error Type: Invalid Message Aliasing Set

Category: Transformations.

Sub Category: Traits Transformation Consistency.

Error Type Definition: This *traits error* happens when a *trait composition use clause* defines a *message aliasing mapping* set containing a *message selector*. The *message aliasing mapping* set must contain *message aliasing mappings*. Containing any other kind of entity is an error because it does not represent a *message aliasing mapping*. One possible reason of this *error* is that the programmer was actually trying to define a *message exclusion* instead of an *message aliasing*.

Currently **Pharo Smalltalk** *Traits* implementation avoids this *Traits error type* occurrence.

Example:

```
Trait named: #TExampleTrait uses: {}
```

```
m1
        ^self aMessage.
```

```
Object subclass: #ExampleClass
       uses: TExampleTrait @ {#m1}
       instanceVariableNames: ''
       classVariableNames: ''
```

Figure 30: Example code showing a *message* selector in the *message aliasing mapping* set.

In this example, **ExampleClass** defines a *message aliasing mapping* set including **m1** instead of containing a valid *message aliasing mapping*.

3.2.13 Error Type: Chained Message Aliasing

Category: Transformations.

Sub Category: Traits Transformation Consistency.

Error Type Definition: This *traits error* happens when a *trait composition use clause* defines two *message aliasing mappings*: **newMessage1 -> oldMessage** and **newMessage2 -> newMessage1** i.e. a *message aliasing* is applied to an *alias message*. A *message aliasing* must be applied to a *trait provided message*, since the objective of defining a *message aliasing* is enabling to have the aliased *message implementation* under the *alias message* name. The expected result of a chained (or transitive) *message aliasing* can be obtained defining all the steps in the *message aliasing* chain as aliases of the original *message* (in this example the defined *message aliasing mappings* should be **newMessage1 -> oldMessage** and **newMessage2 -> oldMessage**). Nevertheless this workaround is not recommended since it would be a "*Duplicated Alias Messages*" *trait error*.

Currently **Pharo Smalltalk** *Traits* implementation avoids this *Traits error type* occurrence.

Example:

```
Trait named: #TExampleTrait uses: {}
```

```
m1
        ^self aMessage.
```

```
Object subclass: #ExampleClass
        uses: TExampleTrait - {(#m2 ->
        #m1)(#m3 -> #m2)}
        instanceVariableNames: ''
        classVariableNames: ''
```

Figure 31: Example code showing a *message aliasing* trying to define an aliasing of an *alias message*.

In this example, **ExampleClass** defines a *message aliasing mapping* chain from **m1** to **m2** and **m3**. This is done with the *message aliasing mappings* **m2 -> m1** and **m3 -> m2**. The *result* is **m2** and **m3** *messages* with the same **m1** *message* implementation. That result can be obtained by defining **m2 -> m1** and **m3 -> m1** *message aliasing mappings* instead of the actually defined ones.

3.2.14 Error Type: Unimplemented Self-Sent Message due Message Renaming

Category: Transformations.

Sub Category: Message Rename.

Error Type Definition: This *traits error* happens when a *method* available at a *trait composition client* **self-sends** a *message* provided by one of the *traits* in its *used trait composition*, but the **self-sent** *message* is renamed and is not provided by the *used trait composition*. As a result of the *message rename* the *trait composition client* **self-sends** an unimplemented *message* (the **self-sent** *message* is actually implemented, but not available under its original name). This *error* is more severe if the *trait composition client* is a *concrete class*, since an *abstract class* or a *trait* can be not complete.

Examples:

There are three different scenarios for this *error*:

1)

Trait named: #TExampleTrait uses: {}	Object subclass: #ExampleClass
	        uses: TExampleTrait @ {#m2->#m1} - {#m1}
	        instanceVariableNames: ''
	        classVariableNames: ''

m1 ^self aMessage.	testMessage ^self m1

Figure 32: Example code showing a *renamed message* self-sent under its original name from a *trait composition use clause client* defined *message*.

In this example, **ExampleClass** implements **testMessage** *message* which **self-sends m1** *message*. **TExampleTrait** implements **m1** *message*, but is not available for **ExampleClass,** since **ExampleClass** *trait composition use clause* renames it.

2)

Trait named: #TExampleTrait1 uses: {}	Object subclass: #ExampleClass
	        uses:
	        TExampleTrait1 @ {#m3->#m1} - {#m1} +
	        TExampleTrait2 - {#m1}
	        instanceVariableNames: ''
	        classVariableNames: ''

m1 ^self aMessage.	
Trait named: #TExampleTrait2 uses: {}	
m2 ^self m1.	
m1 self requiredMethod	testMessage ^doSomething

Figure 33: Example code showing a *renamed message* self-sent under its original name from another *trait* in the *trait composition use clause*.

In this example, **TExampleTrait2** implements **m2** *message* which **self-sends m1** *message.* **TExampleTrait1** implements **m1** *message*, but is not available at **ExampleClass** since its *trait composition use clause* renames it.

3)

```
Trait named: #TExampleTrait uses: {}
m1
        ^self aMessage.
m2
        ^self m1
```

```
Object subclass: #ExampleClass
        uses: TExampleTrait @ {#m3->#m1} -
        {#m1}
        instanceVariableNames: ''
        classVariableNames: ''
```

Figure 34: Example code showing a *renamed message* self-sent under its original name from the same *trait* which originally defined the *renamed message*.

In this example, **TExampleTrait** implements **m2** *message* which **self-sends m1** *message.* **TExampleTrait** also implements **m1** *message*, but is not available at **ExampleClass** since its *trait composition use clause* renames it.

Note that **m1** is not, and cannot be explicitly declared as a **TExampleTrait** *required message*, since it already defines a valid **m1** *message* implementation. This scenario is important in a recursive message implementation, since the **self-sending** *method* will be available under the *alias message*, but the **self-sent** *message* will still be the *excluded aliased message*. This scenario is the main reason to make different this *trait error type* form "*Unimplemented Self-Sent Message due Message Exclusion*" where the *error* is produced by message *exclusion* without a *message aliasing*, and because of that, the recursive method error will not happen.

3.2.15 Error Type: Unimplemented Self-Sent Message due Message Exclusion

Category: Transformations.

Sub Category: Message Exclusion.

Error Type Definition: This *traits error* happens when a *method* available at a *trait composition client* ***self-sends*** a *message* which is not available because a *message exclusion* excludes it from the *used trait composition provided messages*. This *trait error* is similar to "*Unimplemented Self-Sent Message due Message Renaming*". Because of this, the three possible scenarios and the severity consideration for "*Unimplemented Self-Sent Message due Message Renaming*" can be applied to this *trait error type*. The difference between both *trait error types* is that in this *trait error type* the *excluded message* is not available under the *alias message* name in the *trait composition client*, and because of it the recursive message renaming error cannot happen.

Example:

`Trait named: #TExampleTrait uses: {}`	`Object subclass: #ExampleClass`
	`        uses: TExampleTrait - {#m1}`
	`        instanceVariableNames: ''`
	`        classVariableNames: ''`

`m1`
`        ^self aMessage.`	

`m2`
`        self anotherMessage`
	`m3`
	`        ^self m1`

Figure 35: Example code showing a *message exclusion* of a self-sent *message* from a *method* defined in the *trait composition client*.

In this example, **TExampleTrait** defines **m1** *message*, and **ExampleClass>>m3 self-sends m1**, but **TExampleTrait** does not provide **m1** since **m1** was excluded in **ExampleClass** *trait composition use clause*.

3.2.16 Error Type: Excluded and Not Provided Explicit Required Message

Category: Transformations.

Sub Category: Message Exclusion.

Error Type Definition: This *traits error* happens when a concrete class trait composition use clause excludes an explicit required message declaration from one of its composed traits, but the required message is not provided by the rest of the *trait composition* nor by the *used trait composition client* (through direct definition or inherited). This *trait error* only applies when the *trait composition client* is a *concrete class*, because *traits* and *abstract classes* do not have to be complete.

As a best practice recommendation, an *explicit required message* declaration should not be excluded unless it were provided by the *trait composition* or its *client*.

Example:

```
Trait named: #TExampleTrait1 uses: {}
```

```
m1
       ^self requirement.
```

```
m2
       ^self m1
```

```
Object subclass: #ExampleClass
       uses: TExampleTrait1 - {#m1} +
       TExampleTrait2
       instanceVariableNames: ''
       classVariableNames: ''
```

```
Trait named: #TExampleTrait2 uses: {}
```

```
m3
       self anotherMessage
```

```
m4
       ^self m2
```

Figure 36: Example code showing the *message exclusion* of an *explicit required message* declaration, which is not provided by the *trait composition client class* nor its *used trait composition*.

In this example, **TExampleTrait** defines **m1** as an *explicit required message*. **ExampleClass** excludes **m1** *message* on its *used trait composition*, which is not provided by **TExampleTrait2** nor by **ExampleClass**. In this way, **m1** is still a *required message*, but the *explicit required message* declaration is missing at **ExampleClass** making the detection of the missing *required message* more difficult.

3.2.17 Error Type: Always Excluded Message

Category: Transformations.

Sub Category: Message Exclusion.

Error Type Definition: This *traits error* happens when a *trait* **ExampleTrait** provides a *message* **exampleMessage** which is excluded in every use of **ExampleTrait** in a *trait composition use clause*. If **exampleMessage** is always excluded, it could show that **exampleMessage** should not be included into **ExampleTrait** provided messages.

3.2.18 Error Type: Too Many Excluded Messages

Category: Transformations.

Sub Category: Message Exclusion.

Error Type Definition: This *traits error* happens when all or most of the messages provided by an **ExampleTrait** *trait* are excluded in a *used trait composition*. When this happen it is an evidence against using **ExampleTrait** in that *trait composition use clause*, or maybe a reason for refactoring **ExampleTrait** and obtain a new *trait* which provides just the needed *messages*.

Example:

```
Trait named: #TExampleTrait uses: {}
```

```
m1
       ^self aMessage.
```

```
m2
       self anotherMessage
```

```
m3: anObject
       ^anObject add: self m1
```

```
Object subclass: #ExampleClass
       uses: TExampleTrait - {#m1.#m2.#m3:}
       instanceVariableNames: ''
       classVariableNames: ''
```

Figure 37: Example code showing a *trait composition use clause* which applies *message exclusion* to all the *provided messages* of the transformed *trait*.

In this example, **ExampleClass** uses **TExampleTrait** on its *trait composition use clause*, but excludes all the **TExampleTrait** *provided messages*. It would be preferable to exclude **TExampleTrait** from the *used trait composition*.

3.2.19 Error Type: Misplaced Meta-Level Instance Message Aliasing

Category: Transformations.

Sub Category: Trait Transformation Meta-level Error.

Error Type Definition: This *traits error* happens when a *message aliasing* applied in the *class messages* side is invalid, but it would be valid if it were applied to the *instance messages* side.

More formally, A *meta-behavior* (can be a *metaClass* or *classTrait* i.e. the *class messages* side of a *behavior* definition) defines on its *used trait composition* a *message aliasing* applied to a *classTrait* which does not contain the *aliased message*, but the *classTrait's trait*, used in the corresponding *behavior trait composition use clause* (i.e. in the *instance messages* side) defines the *message* to be aliased.

Note that this *traits error* happens when there is an "*Undefined Aliased Message*" for a *classTrait* in a *class messages* side *trait composition use clause*, but the *message aliasing* actually applies to the corresponding *classTrait's trait* in the *instance messages* side *trait composition use clause.*

Also note that the *message aliasing* should apply to the corresponding *classTrait's trait* and it does not care if the *message aliasing* applies to another *trait*.

Example:

`Trait named: #TExampleTrait uses: {}`	`Object subclass: #ExampleClass`
	`    uses: TExampleTrait`
	`    instanceVariableNames: ''`
	`    classVariableNames: ''`

`m1`
`    ^self aMessage.`	

`TExampleTrait classTrait uses: {}`	`#ExampleClass class`
	`    uses: (TExampleTrait classTrait) @`
	`    {#m2 -> #m1}`
	`    instanceVariableNames: ''`

`mc1`
`    ^self somethingToDo`	

Figure 38: Example code showing a *message aliasing* of an *instance message* defined in the *metaClass* definition.

In this example, **ExampleClass** *trait composition use clause* includes **TExampleTrait**, and symmetrically, **ExampleClass class** *metaClass* uses **TExampleTrait classTrait** on its used *trait composition*. In **ExampleClass class** *metaclass trait composition use clause*, **m2 -> m1** *message aliasing mapping* is applied to **TExampleTrait classTrait**, which does not define **m1** *required message*. Nevertheless **m2 -> m1** would fit if it were applied to **TExampleTrait** in **ExampleClass** *trait composition use clause*.

3.2.20 Error type: Misplaced Meta-Level Instance Message Exclusion

Category: Transformations.

Sub Category: Trait Transformation Meta-level Error.

Error Type Definition: This *traits error* happens when a *message exclusion* applied in the *class messages* side is invalid, but it would be valid if it were applied to the *instance messages* side.

More formally, A *meta-behavior* (can be a *metaClass* or *classTrait* i.e. the *class messages* side of a *behavior* definition) defines on its *used trait composition* a *message exclusion* applied to a *classTrait* which does not contain the *excluded message*, but the *classTrait's trait*, used in the corresponding *behavior trait composition use clause* (i.e. in the *instance messages* side) defines the *message* to be excluded.

Note that this *traits error* happens when there is an "*Undefined Excluded Message*" for a *classTrait* in a *class messages* side *trait composition use clause*, but the *message exclusion* actually applies to the corresponding *classTrait's trait* in the *instance messages* side *trait composition use clause.* Also note that the *message exclusion* should apply to the corresponding *classTrait's trait* and it does not care if the *message exclusion* applies to another *trait*.

Example:

`Trait named: #TExampleTrait uses: {}`	`Object subclass: #ExampleClass`
	`       uses: TExampleTrait`
	`       instanceVariableNames: ''`
	`       classVariableNames: ''`

`m1`
`       ^self aMessage.`	

`TExampleTrait classTrait uses: {}`	`#ExampleClass class`
	`       uses: (TExampleTrait classTrait) -`
	`       {#m1}`
	`       instanceVariableNames: ''`

`mc1`
`       ^self somethingToDo`	

Figure 39: Example code showing a *message exclusion* of an *instance message* defined in the *metaClass* definition.

In this example, **ExampleClass** *trait composition use clause* includes **TExampleTrait**, and symmetrically, **ExampleClass class** *metaClass* uses **TExampleTrait classTrait** on its *used trait composition*. In **ExampleClass class** *metaClass trait composition use clause*, the *message exclusion* for **m1** is applied to **TExampleTrait classTrait**, which does not define the **m1** *required message*. Nevertheless the *message exclusion* for **m1** would fit if it were applied to **TExampleTrait** in **ExampleClass** *trait composition use clause*.

3.2.21 Error Type: Misplaced Meta-Level Class Message Aliasing

Category: Transformations.

Sub Category: Trait Transformation Meta-level Error.

Error Type Definition: This *traits error* happens when a *message aliasing* defined for a *trait* in a *behavior's used trait composition* is not valid because the transformed *trait* does not include the *message* to be aliased, but the *message* to be aliased is defined in the corresponding transformed *trait's classTrait* used in the *behavior's meta-behavior used trait composition* (the *meta-behavior* of a *behavior* is the corresponding *classTrait* of a *trait* of the *metaClass* of a *class*). In a simpler way, an undefined *message aliasing* defined in the *instance messages* side would be valid if it were defined in the *class messages* side.

Note this *traits error* happens when an undefined *message aliasing* in a *used trait composition* becomes valid if it were applied to the *trait classTrait*, which must be used in the *meta-behavior trait composition use clause*.

Example:

`Trait named: #TExampleTrait uses: {}`	`Object subclass: #ExampleClass`
	`uses: TExampleTrait@{#m2 -> #mc1}`
	`instanceVariableNames: ''`
	`classVariableNames: ''`

`m1`
`^self aMessage.`	

`TExampleTrait classTrait uses: {}`	`#ExampleClass class`
	`uses: TExampleTrait classTrait`
	`instanceVariableNames: ''`

`mc1`
`^self somethingToDo`	

Figure 40: Example code showing an invalid *message aliasing* in the *instance message* side. This *message aliasing* would be valid if it were applied to the *class messages side used trait composition*.

In this example, **ExampleClass** *used trait composition* includes **TExampleTrait**, and symmetrically, **ExampleClass class** *metaClass* uses **TExampleTrait classTrait** on its *used trait composition*. In **ExampleClass** *used trait composition*, **m2 -> mc1** *message aliasing mapping* is applied to **TExampleTrait**, which does not define the **mc1** *required message*. However **m2 -> mc1** would fit if it were applied to **TExampleTrait classTrait** in **ExampleClass class** *metaClass used trait composition*.

3.2.22 Error Type: Misplaced Meta-Level Class Message Exclusion

Category: Transformations.

Sub Category: Trait Transformation Meta-level Error.

Error Type Definition: This *traits error* happens when a *message exclusion* defined for a *trait* in a *used trait composition* is not valid because the transformed *trait* does not include the message to be excluded, but the *message* to be excluded is defined in the corresponding transformed *trait's classTrait* in the *trait composition client meta-behavior trait composition use clause* (the *meta-behavior* of a *behavior* is the corresponding *classTrait* of a *trait* or the *metaClass* of a *class*). In a simpler way, an undefined *message exclusion* defined in the *instance messages* side would be valid if it were defined in the *class messages* side.

Note this *traits error* happens when an undefined *message exclusion* in a *used trait composition* becomes valid if it were applied to the *trait classTrait*, which must be used in the *meta-behavior used trait composition*.

Example:

`Trait named: #TExampleTrait uses: {}`	`Object subclass: #ExampleClass uses: TExampleTrait-{#mc1} instanceVariableNames: '' classVariableNames: ''`
`m1 ^self aMessage.`	
`TExampleTrait classTrait uses: {}`	`#ExampleClass class uses: TExampleTrait classTrait instanceVariableNames: ''`
`mc1 ^self somethingToDo`	

Figure 41: Example code showing an invalid *message exclusion* in the *instance messages* side. This *message exclusion* would be valid if it were applied to the *class messages* side *used trait composition*.

In this example, **ExampleClass** *used trait composition* includes **TExampleTrait**, and symmetrically, **ExampleClass class** *metaClass* uses **TExampleTrait classTrait** on its *used trait composition*. In **ExampleClass** *trait composition use clause*, the **mc1** *message exclusion* is applied to **TExampleTrait**, which does not define the **mc1** *required message*. However the *message exclusion* for **mc1** would fit if it were applied to **TExampleTrait classTrait** in **ExampleClass class** *metaClass used trait composition.*

3.2.23 Error Type: Trait Composition Conflict Method

Category: Composition.

Sub Category: Trait Composition Conflict.

Error Type Definition: This *traits error* happens when a *trait composition client* has a *trait composition conflict method* as one of its *messages implementation.* This *method* is automatically generated when a *trait composition conflict* is defined on a *trait composition use clause* and the conflicting *message* is not redefined by the *trait composition client*. A *trait composition conflict method* can also appear with a conflict free *trait composition use clause* when a previously existing *trait composition conflict* has been solved, but the already generated *trait composition conflict method* was not removed from the *trait composition client*.

Example:

`Trait named: #TExampleTrait1 uses: {}`	`Object subclass: #ExampleClass`
	`uses: TExampleTrait1 + TExampleTrait2`
	`instanceVariableNames: ''`
	`classVariableNames: ''`

`m1`
`        ^self aMessage.`	

`m2`
`        ^self m1`	

`Trait named: #TExampleTrait2 uses: {}`	`m2`
	`        self traitConflict`

`m2`
`        self anotherMessage`
	`m4`
	`        self traitConflict`

Figure 42: Example code showing *trait composition conflict methods* not solved in the *trait composition client class*.

In this example, **ExampleClass** *trait composition use clause* composes **TExampleTrait1** and **TExampleTrait2** which conflict on **m2** *message.* This *composition conflict* generates **m2** tr*ait composition conflict method* available on **ExampleClass**. There is another *trait composition conflict method* on **ExampleClass>>m4**, but which is not related with any *trait composition conflict* at its *used trait composition*. This could be product of an already fixed *trait composition conflict* in the *trait composition use clause* or be a *method* explicitly defined by the programmer.

3.2.24 Error Type: Trait Composition Conflict due Aliasing

Category: Composition.

Sub Category: Trait Composition Conflict.

Error Type Definition: This *traits error* happens when, in a *trait composition conflict*, at least one of the conflicting *messages* is an *alias message*.

Example:

```
Trait named: #TExampleTrait1 uses: {}
m1
        ^self aMessage.
m2
        ^self m1
Trait named: #TExampleTrait2 uses: {}
m3
        self anotherMessage
```

```
Object subclass: #ExampleClass
        uses: TExampleTrait1@{#m3 -> #m1} +
        TExampleTrait2
        instanceVariableNames: ''
        classVariableNames: ''
```

Figure 43: Example code showing a *trait composition conflict* generated by a conflict between a *trait* defined *message* and an *alias message*.

In this example, **ExampleClass** composes **TExampleTrait2** which provides **m3** *message* and a *trait transformation* of **TExampleTrait1** which also provides **m3** *message* through *message aliasing*. Since **ExampleClass** does not redefine **m3,** a *trait composition conflict method* is defined on **ExampleClass>>m3** because of both **m3** *messages* provided at **ExampleClass** *used trait composition*.

3.2.25 Error Type: Unnecessary Message Exclusion

Category: Composition.

Sub Category: Unnecessary Trait Transformation.

Error Type Definition: This *traits error* happens when a *message exclusion* is defined in a *trait composition use clause* in a way that if it were not defined, no *trait composition conflict* would happen i.e. the *message exclusion* is unnecessary since its goal is to avoid *trait composition conflicts*, and it does not prevent any *composition conflict*.

Example:

```
Trait named: #TExampleTrait1 uses: {}
m1
       ^self aMessage.
m2
       ^self m1
Trait named: #TExampleTrait2 uses: {}
m3
       self anotherMessage
```

```
Object subclass: #ExampleClass
       uses: TExampleTrait1 - {#m1} +
       TExampleTrait2
       instanceVariableNames: ''
       classVariableNames: ''
```

Figure 44: Example code of a *message exclusion* that does not solve any *trait composition conflict*.

In this example, **ExampleClass** defines a *message exclusion* for **m1** on **TExampleTrait1**. If no *message exclusion* were applied, no *trait composition conflict* would happen since **TExampleTrait2** does not provide **m1** *message*.

3.2.26 Error Type: Unnecessary Message Aliasing

Category: Composition.

Sub Category: Unnecessary Trait Transformation.

Error Type Definition: This *traits error* happens when a *message aliasing* is defined in a *trait composition use clause*, but the *alias message* is not referenced by any *method* available at the *trait composition client*. An *alias message* not referenced in the *trait composition client* is valid when the programmer wants to make that *alias message* the *message* name for the *aliased message method* in the *trait composition client*. If it were the case, it would be preferable a message rename because it is confusing to have the same implementation for two different messages.

Example:

`Trait named: #TExampleTrait1 uses: {}`	`Object subclass: #ExampleClass uses: TExampleTrait1 - {#m4 -> #m1} + TExampleTrait2 instanceVariableNames: 'anObject' classVariableNames: ''`
`m1 ^self aMessage.`	
`m2 ^self m1`	
`Trait named: #TExampleTrait2 uses: {}`	`m5 anObject doSomethingWith: self m1`
`m3 self anotherMessage`	

Figure 45: Example code showing a *message aliasing* for which the *alias message* is not sent in any *trait composition client method*.

In this example, **ExampleClass** defines **m4** *message alias* for **TExampleTrait1>>m1**, but **m4** is not sent at any *method* available at **ExampleClass**.

3.2.27 Error Type: Override with Identical Method

Category: Traits Use.

Sub Category: Message Overriding.

Error Type Definition: This *traits error* happens when a *trait composition client* defines a *message* which overrides one of its *used trait composition provided messages,* but defining a *method* identical to the overridden one.

Example:

`Trait named: #TExampleTrait uses: {}`	`Object subclass: #ExampleClass`
	`uses: TExampleTrait`
	`instanceVariableNames: ''`
	`classVariableNames: ''`

`m1`
`^self aMessage.`	

`m3: anObject`
`^anObject add: self m1`
	`m3: anObject`
	`^anObject add: self m1`

Figure 46: Example code showing a *class* overriding with the same *method* one of its *used trait composition provided messages*.

In this example, **ExampleClass** implements **m3:** *message* with the same *method* implementation of **TExampleTrait>>m3:**. One possible reason for this is that, during a refactoring, the programmer found that the same *method* is defined on multiple places, and decides to move it to a *trait*, but forgets *to remove* the *method* from all the new *trait composition clients.*

3.2.28 Error Type: Overridden Aliasing

Category: Traits Use.

Sub Category: Message Overriding.

Error Type Definition: This *traits error* happens when a *message aliasing* is defined in a *trait composition use clause*, but its *trait composition client* overrides the *alias message* making the *alias message* implementation unavailable at the *trait composition client*.

Example:

`Trait named: #TExampleTrait uses: {}`	`Object subclass: #ExampleClass`
	`        uses: TExampleTrait@{#m2 -> #m1}`
	`        instanceVariableNames: 'anObject'`
	`        classVariableNames: ''`

`m1`
`        ^self aMessage.`	

`m3: anObject`
`        ^anObject add: self m1`
	`m2`
	`        ^anObject doSomething`

Figure 47: Example code showing a *class* overriding an *alias message* from its *used trait composition*.

In this example, **ExampleClass** defines a *message aliasing mapping* **m2 -> m1** on its *trait composition use clause*, but also locally defines **m2** which overrides **m2** *alias message*.

3.2.29 Error Type: Always Overridden Message

Category: Traits Use.

Sub Category: Message Overriding.

Error Type Definition: This *traits error* happens when an **ExampleTrait** *trait* provides an **exampleMessage** *message* which is overridden by all the *behavior* clients of every *trait composition* using **ExampleTrait**. If this happen, it could show that **exampleMessage** should not be included in **ExampleTrait** protocol.

3.2.30 Error Type: Too Many Overridden Messages

Category: Traits Use.

Sub Category: Message Overriding.

Error Type Definition: This *traits error* happens when an **ExampleTrait** *trait* is used in a *trait composition use clause* where all or most of the **ExampleTrait** *provided messages* are overridden by its *trait composition client*. When this happens, it is an indicator against using **ExampleTrait** in the *trait composition use clause*, or maybe for refactoring **ExampleTrait** to obtain a new *trait* which provides just the not overridden *messages*.

Example:

`Trait named: #TExampleTrait uses: {}`	`Object subclass: #ExampleClass`
	`uses: TExampleTrait`
	`instanceVariableNames: ''`
	`classVariableNames: ''`

`m1`
`^self aMessage.`
	`m1`
	`^doOtherThing`

`m2`
`self anotherMessage`
	`m2`
	`^doSomethingElse`

Figure 48: Example code showing a *class* overriding all the *provided messages* of a *trait* on its used *trait composition*.

In this example, **ExampleClass** uses **TExampleTrait** on its *trait composition use clause*, but overrides all the **TExampleTrait** *provided messages*. In this scenario, it would be preferable to remove **TExampleTrait** from the *trait composition use clause*.

3.2.31 Error Type: Unimplemented Required Message

Category: Traits Use.

Sub Category: Required Messages.

Error Type Definition: This *traits error* happens when a *concrete class* does not provide one of its *used trait composition required messages*. If the *trait composition client* is a *trait* or an *abstract class* it does not have to provide all its *used trait composition required messages* since it does not have to be complete.

Example:

`Trait named: #TExampleTrait uses: {}`	`Object subclass: #ExampleClass` `uses: TExampleTrait` `instanceVariableNames: 'anObject'` `classVariableNames: ''`
`m1` `^self requirement.`	
`m3: anObject` `^anObject add: self m1`	`m2` `^anObject doSomething`

Figure 49: Example code showing a *concrete class* which does not provide all its *used trait composition required messages*.

In this example, **TExampleTrait** defines **m1** as *required message*. The *concrete class* **ExampleClass** uses **TExampleTrait** as its *used trait composition*, but does not implement **m1** *message*. Because of this, if an **ExampleClass** instance receives an **m1** *message*, it would generate an error since **ExampleClass** does not provide a valid m1 *message* implementation.

3.2.32 Error Type: Hidden Implementation by Explicitly Required Message

Category: Traits Use.

Sub Category: Required Messages.

Error Type Definition: This *traits error* happens when a *trait* in a *class used trait composition* declares an *explicit required message* which overrides an implementation of the *explicit required message* provided by one of the *class' superclasses* via *inheritance*. Ideally an *explicit required message* declaration should not override *superclass provided messages* but in practice they behave as any other *trait composition provided messages*.

Example:

`Trait named: #TExampleTrait uses: {}`	`Object subclass #ExampleSuperClass`
	`        instanceVariableNames: ''`
	`        classVariableNames: ''`

`m1`
`        ^self requirement.`
	`m1`
	`        ^self doSomeStuff`

`m2`
`        ^self m1 add: anObject`
	`ExampleSuperClass subclass: #ExampleClass`
	`        uses: TExampleTrait`
	`        instanceVariableNames: 'anObject'`
	`        classVariableNames: ''`

Figure 50: Example code showing a *trait explicit required message* declaration which "*hides*" a valid *message* implementation defined at the *superclass* of the *trait composition client* class.

In this example, **ExampleClass** should inherit **ExampleSuperClass>>m1**, but it actually receives **m1** *explicit required message* declaration from **TExampleTrait** through its *used trait composition*.

3.2.33 Error Type: Not Explicitly Declared Required Message

Category: Traits Use.

Sub Category: Required Messages.

Error Type Definition: This *traits error* happens when a *trait* does not define an *explicit required message* declaration for some of its *required messages*. Despite it is valid to have *implicit required messages* it is clearer explicitly declaring the *required messages*.

Example:

```
Trait named: #TExampleTrait uses: {}

m1: anObject
       ^anObject add: self m2
```

Figure 51: Example code showing a *required message* not explicitly declared.

In this example, **TExampleTrait** requires **m2** message which is not declared as an *explicit required message*. For this example it would be preferable to have an *explicit required message* declaration for **m2** message.

3.2.34 Error Type: Unused Required Message

Category: Traits Use.

Sub Category: Required Messages.

Error Type Definition: This *traits error* happens when a *trait* defines an *explicit required message* declaration for a *message* that is not required i.e. a declared *explicit required message* is not **self-sent** in any *message* provided by the *trait*.

Example:

```
Trait named: #TExampleTrait uses: {}

m1: anObject
	^anObject hash

m2
	^self requirement
```

Figure 52: Example code showing an *explicit required message* which is not actually required.

In this example, **TExampleTrait** declares **m2** *message* as an *explicit required message* but since no other *method* **self-sends m2**, it is not a *required message*.

3.2.35 Error Type: Super-Sent Message Lookup Bypasses Used Trait Composition Provided Message

Category: Traits Use.

Sub Category: Super Sent Messages.

Error Type Definition: This *traits error* happens when a *trait composition client* **super-sends** a *message* which is provided by its *used trait composition*. If a *trait composition client* needs to use a *trait composition provided message*, it should do a **self-send** instead of a **super-send** since the semantics of a *trait composition provided message* is the same as it were defined in the *trait composition client*. If an implementation extension needs to use the same *message* name, the *trait composition provided message* should be referenced by an *alias message* since the *trait composition client* defined *message* will override the *trait composition provided message*.

Example:

`Trait named: #TExampleTrait uses: {}`	`Object subclass: #ExampleClass`
	`        uses: TExampleTrait`
	`        instanceVariableNames: 'anObject'`
	`        classVariableNames: ''`

`m1`
`        ^self doSomething.`
	`m3`
	`        ^anObject: super m1 doSomethingElse`

Figure 53: Example code showing a *class* super-sending a *message* for which the *method lookup* bypasses the *used trait composition provided message*.

In this example, **ExampleClass super-sends m1** *message* which is provided by **TExampleTrait**. Because of *flattening property*, **m1** provided to **ExampleClass** by using **TExampleTrait** is semantically equivalent to defining **m1** in **ExampleClass**. Because of this, **super-sending m1** from an **ExampleClass** instance will make *method lookup* to bypass **TExampleTrait>>m1** *provided message*.

3.2.36 Error Type: Trait Method Super-Sends a Message

Category: Traits Use.

Sub Category: Super Sent Messages.

Error Type Definition: This *traits error* happens when a *trait defined method* **super-sends** a *message*. Despite **super-sending** a *message* is valid; *Traits* model only defines a mechanism to declare *required messages* to be provided by the *trait composition client* and does not define any way to declare requirements to be satisfied by one of the *trait composition client superclasses*.

There are some scenarios where **super-sending** a *message* from a *trait defined method* is valid. One possible scenario for this is extending a *message* behavior **super-sending** the same *message*, which is useful to model a generic wrapper [SDNB/03]. Despite of it, **super-sending** a *message* from a *trait defined method* is not recommended because of the lack of mechanisms to declare requirements for a *trait composition client superclass*.

Example:

`Trait named: #TExampleTrait uses: {}`	`Object subclass #ExampleSuperClass`
	`instanceVariableNames: ''`
	`classVariableNames: ''`

`m1`
`^super m2.`
	`ExampleSuperClass subclass: #ExampleClass`
	`uses: TExampleTrait`
	`instanceVariableNames: 'anObject'`
	`classVariableNames: ''`

	`m2` `^doSomeStuff`

Figure 54: Example code showing TExampleTrait used by ExampleClass and defining a *method* which *super-sends* a *message*.

In this example, **TExampleTrait** implements **m1** *message*, which **super-sends m2**. **ExampleClass** uses **TExampleTrait** and implements **m2**, but, because the *flattening property*, the *method lookup* will start looking at a **m2** implementation at **ExampleSuperClass**, missing **ExampleClass>>m2** implementation.

3.2.37 Error Type: Unused Trait

Category: Best Practices.

Sub Category: Traits Definition and Use.

Error Type Definition: This *traits error* happens when, a *trait* is not included in any *class* or *trait used trait composition*. Since this *trait* is not used at all there is no reason to have it**.**

3.2.38 Error Type: Traits Names Don’t Start with T

Category: Best Practices.

Sub Category: Traits Definition and Use.

Error Type Definition: This *traits error* happens when a *trait* name does not start with “**T**”. As a convention for naming a *trait* and easily differentiating it from a *class*, it is an accepted practice to name *traits* with a “**T”** name prefix. Examples of this are **TDrawing** and **TCircle** *traits*, presented at “*Defining Traits*” section in this work.

4 Automatic Trait Error Detection Implementation

The chosen platform for implementing automatic *Traits error* detection is **Smalllint**, a *static code checking* tool inspired on **Lint**, which is available in several **Smalltalk** implementations like **Pharo**, **Squeak** and others. The original **Smalllint** implementation is only able to check *classes* and *methods*, and because of this, it had to be modified to analyze and detect *Traits related errors*.

This section will describe:

- **Smalllint** description and implementation.
- **Smalllint** limitations for *Traits error* detection.
- **Smalllint** extension implementation.
- **Smalllint** *Traits error* detection *rules* implementation.

The described **Smalllint** extension has been implemented modifying the **Smalllint** version available in **Pharo Smalltalk 1.0**. This implementation has been published as **Monticello** packages in **squeaksource** repository at *http://www.squeaksource.com/TesisTraitLint*. For convenience, the implementation is distributed in three packages:

- **OBLintTraitExtension:** It contains the **Smallint** extension implemented using **OmniBrowser** framework, including browsers, *environments* and other **Smalllint** related elements.
- **Tesis:** It contains all the *Traits* related *rule* implementations, including **Smallint** *rules*, *Traits* inspection framework, rule format adaptation and other rule related elements.
- **TesisTest:** It contains the *unit tests* that checks the behavior of the *classes* included in **Tesis** *package*.

OBLintTraitExtension and **Tesis** *packages* can be loaded independently and **TesisTest** depend on **Tesis** *package*. Despite **OBLintTraitExtension** and **Tesis** can be loaded independently, but to get the full **Smalllint** extension including **Traits** related *rules* both *packages* have to be loaded.

4.1 Smalllint Description and Implementation

As said previously, **Smalllint** is a *static code analysis* tool provided in various **Smalltalk** dialects like **Squeak** and **Pharo**. **Smalllint** is based on and works similarly to **Lint** tool.

To run **Smalllint** on some code, the programmer has to select a *refactoring scope*. A *refactoring scope* is a set of *classes* where **Smalllint** will search for *errors*. To detect *errors*, **Smalllint** defines a set of *rules* to evaluate on the defined *refactoring scope*. Each of these *rules* is defined to detect a specific *error type*. In a *rule* evaluation the *rule* receives a *class* or a *method*, checks it, and, in case of positive *error* detection, adds it to the *rule result* set.

After filling in the evaluation of all the *rules* on the selected *refactoring scope*, **Smalllint** shows for each *rule*, the *classes* or *methods* which have been added to the *rule's result set* (i.e. shows the *classes* and *methods* with *errors*).

This section will describe the current implementation of **Smalllint** and its components, including its model and **UI**. Later, we describe **Smalllint** limitations and problems to detect *Traits errors*, and the changes introduced to give **Smalllint** the ability for *Traits related error* detection.

4.1.1 Smalllint Model

Smalllint is included in the **Pharo** image, as part of the **Refactoring Browser**, a framework which includes several useful tools like refactoring, rewriting *rules*, etc. The **Smalllint** (or **Lint**) model consists of a set of *rules*, each of them associated with a specific *error type*, that check *classes* and/or *methods* finding *error* occurrences if appropriate. This set of rules will be extended with *traits* related *rules* as result of this thesis.
This chapter will present the **Smalllint** model in more detail (note that many model *classes* names are prefixed with "**RB**" which comes after **Refactoring Browser**).

Rules

In the **Smalllint** model, every *rule* is implemented by a *class*, and all the *rules* are organized in a *class* hierarchy with **RBLintRule** as the root.
There are several types of *rules*:

- **RBBasicLintRule** models the individual *rules*, for which there are two main types.
- **RBBlockLintRule** lets the programmer define an individual *rule* in a programmatic way; this means the programmer defines code to detect the *error*.
- **RBParseTreeLintRule** lets the programmer define an individual *rule* using a special syntax to analyze *methods parse tree* in a declarative way.
- **RBCompositeLintRule** groups other **Lint** *rules* and evaluates each of them when executed.
- **RBTransformationRule** that lets the programmer define code transformations (hopefully, changing one code chunk to another semantically equivalent chunk of code).

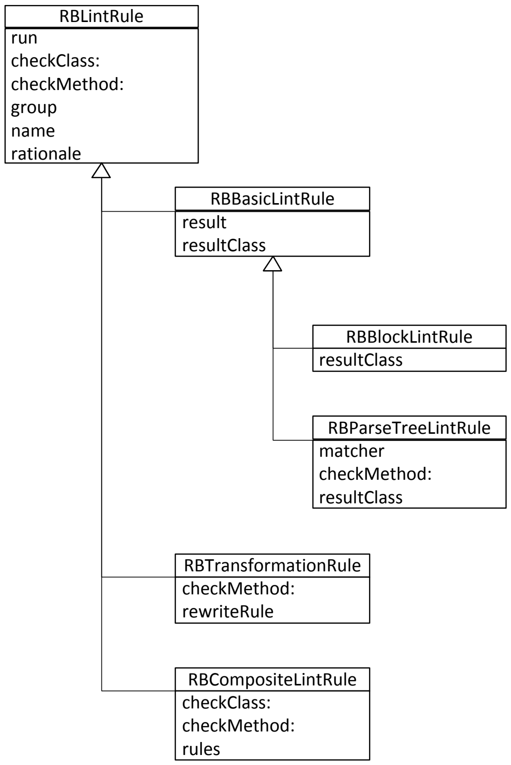


Figure 55: Smalllint rules *class hierarchy*.

Rule Evaluation

A **Lint** *rule* works as follows:

A *rule* implements at least one of the *messages* {**checkClass:**, **checkMethod:**}, depending on if the *rule* will verify *classes* or *methods*. The rule also defines a *result class*. The *result class* models a set of *rule results*; it can be a set of *classes*, a set of *method selectors* or any other kind of set. The different result *sets available* are implemented as **BrowserEnvironment** subclasses.

When a *rule* is evaluated, it receives a **checkClass:** or **checkMethod:** *message* with a *context* containing the entity (a *class* or a *message selector*/*compiled method*) to be analyzed. After receiving the *context*, the *rule* will take the entity to analyze, and check it looking for an occurrence of the *error type* for which the *rule* has been defined. After analyzing the received entity, the *rule*, in case of *positive error detection*, will add a *result* (usually the analyzed entity) into its *rule result set*.

Rule Checker

In **Smalllint**, *rules* have the responsibility of checking individual entities for a specific *error type*; while the responsibility of evaluating *rules* on a group of entities relies on **SmalllintChecker**. A **SmalllintChecker** includes a *rule*, usually a **RBCompositeLintRule** *instance* composed by a group of *rules*, and also an *environment*, which includes a set of *classes*. The *environment* is usually a **BrowserEnvironment** *instance*, which allows to iterate on a *class* set, using the **classesDo:** *message*. When a **SmalllintChecker** receives a **run** *message*, it resets its *rule result* and then iterates over the *classes* included in the checker *environment*.

For each iterated *class*, **SmalllintChecker** sends **checkClass:** and **checkMethod:** *messages* to its *rule*. **checkClass:** is sent with the currently iterated *class* and **checkMethod:** is sent once with each of the iterated *class* defined *messages*. In case of a *composite rule*, the composed *rule* is responsible of forwarding the received *messages* (**checkClass:**, **checkMethod:** and any other) to its composing *rules*. In this way, when a composed *rule* is evaluated, it is the same as evaluating all of its composing *rules*.

```
SmalllintChecker>>run
	rule resetResult.
	environment classesDo: [ :class |
		class isTrait ifFalse: [
			self checkClass: class.
			self checkMethodsForClass: class ] ]
```

Figure 56: SmalllintChecker>>run *message implementation.*

4.1.2 UI

Smalllint UI is implemented using **Omnibrowser** framework, which is a browser framework that supports the definition of browsers based on explicit *meta-models* definition [BDPW/07]. The basics of **Omnibrowser** and **Smalllint UI** implementation will be described next.

Omnibrowser Framework

Omnibrowser is a framework to write browser-based tools in **Smalltalk**. It is used to implement browsers that are typically included in a **Smalltalk IDE**, such as the **Refactoring Browser** (which includes **Smalllint**) that runs in **Squeak** and **Pharo**[1].

In the **Omnibrowser** model, a *meta-graph* defines the navigation structure for the displayed data; each *meta-node* defines a kind of "*real*" *node* with its behavior and a set of *transitions* to other *meta-nodes* i.e. other kind of *nodes* that can be reached from the current *node*. The real data is modeled with a *graph* which is an "*instance*" of its *meta-graph* and where each "*real*" *node is* an "*instance*" of a *meta-node* with the properties and *transitions* like the ones defined by its *meta-node*.

Each time an item in the browser is selected, a "*real*" *node* is selected in the model. And then, the browser automatically computes the following possible items to display, coming from the selected "*real*" *node*. To compute them, the browser selects the *meta-node* for the selected "*real*" *node*, and then from that *meta-node* all the possible *transitions* are retrieved. Following that, all the *transitions* are sent as *messages* to the selected *"real" node*, getting a set of "real" *nodes* as response for each "*transition*" *message*. Each "*real*" *node* coming from the computed *transitions* have as its *meta-node* the *meta-node* reached in the *meta-graph* taking that *transition*.

Example

An implementation of a file browser using **Omnibrowser** will be described next:

First, the file browser itself has to be defined

```
OBBrowser subclass: #FileBrowser
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'PBE-Omnibrowser'
```

Figure 57: FileBrowser browser *class* definition [OBWeb].

This browser has to implement a *meta-graph* describing the file system data structure to be navigated. Since a usual file system includes files and directories, the *meta-graph* includes a directory and file *meta-nodes*. To model the directories ability to contain other files and directories, two *transitions* are defined in the *meta-graph*, **directories** going from directory to directory, and **files** going from directory to file.

```
FileBrowser class»defaultMetaNode
"returns the directory metanode that acts as the root metanode"
| directory file |
        directory := OBMetaNode named: 'Directory'.
        file := OBMetaNode named: 'File'.
        directory
        childAt: #directories put: directory;
        childAt: #files put: file.
        ^directory
```

Figure 58: FileBrowser *method* defining the file browser *meta-graph* [OBWeb].

[1] The Omnibrowser code can be found at http://code.google.com/p/omnibrowser/

To implement this *meta-graph*, the **FileBrowser** implements **defaultMetaNode** *message*. **defaultMetaNode** *message* implementation defines the full *meta-graph* and returns its root *meta-node*.

The next step is to define the *concrete nodes*. These *nodes* are wrappers of the domain entities with the responsibility of adapting them to be handled by the browser. Two different *nodes* are defined in this case: **FileNode** and **DirectoryNode**.

```
OBNode subclass: #FileNode
       instanceVariableNames: 'path'
       classVariableNames: ''
       poolDictionaries: ''
       category: 'PBE-Omnibrowser'
```

```
FileNode subclass: #DirectoryNode
       instanceVariableNames: ''
       classVariableNames: ''
       poolDictionaries: ''
       category: 'PBE-Omnibrowser'
```

Figure 59: *Class* definitions for the *concrete nodes* representing **File** and **Directory** domain entities [OBWeb].

Instances of **FileNode** and **DirectoryNode** will represent concrete files and directories of the file system domain. The first step is to define **FileNode** *class* as a **OBNode** *subclass*. Instances of this *class* will represent *files*. The other entity in our model is *directory*, which can contain *files* and other *directories*. A directory can be modeled as a special kind of file. Because of this, **DirectoryNode** is defined as a *subclass* of **FileNode**.

After defining **FileNode** and **DirectoryNode**, the *meta-graph* defined *transitions* have to be defined. As said previously, there are two possible *transitions* from a directory *meta-node*, **directories** and **files**.

```
DirectoryNode»directories
| dir |
       dir := FileDirectory on: path.
       " dir directoryNames collect: [:each |
       DirectoryNode new path: (dir fullNameFor:
       each)]
```

```
DirectoryNode»files
| dir |
       dir := FileDirectory on: path.
       " dir fileNames collect: [:each |
       FileNode new path: (dir fullNameFor:
       each)]
```

Figure 60: DirectoryNode *messages* representing the *transitions* on the *meta-model* [OBWeb].

When a browsed item is selected, the possible *transitions* are obtained from its item *node's meta-node*, and then, those *transitions* are sent to the selected concrete *node* as messages. Because this, **directories** and **file** *messages* are defined at **DirectoryNode** *class*.

directories will return a set of directory *nodes* (**DirectoryNode** instances) and **files** will return a set of file *nodes* (**FileNode** instances), in both cases, corresponding to the files and directories contained in the *node's* directory. Also note that a directory *node* will have a directory *meta-node*, since it will be obtained after following a **directories** *transition*. The same will be applicable to file *nodes* with file *meta-nodes* after taking **files** *transition*.

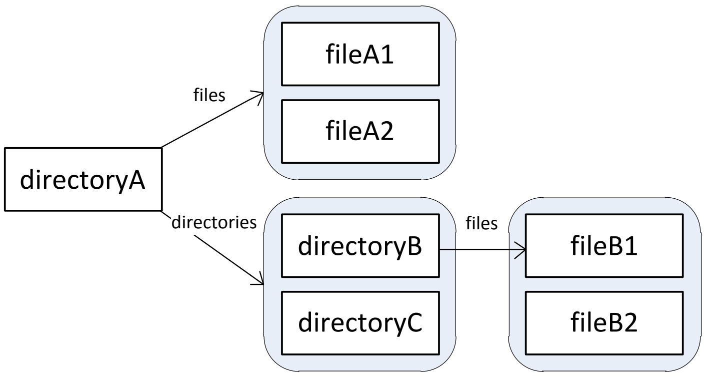


Figure 61: A file domain *concrete graph* example.

Figure 61 shows an example of a concrete file browser *graph* where concrete *nodes* are grouped by its taken transition, following its *meta-graph* defined structure and transitions. Files come from **files** *transitions* and directories come from **directories** *transitions*.
As a final step in the browser definition, the *concrete root node* is defined. The *concrete root node* specifies where the domain navigation will start. In this case the *concrete root node* will represent the file system *root* directory.

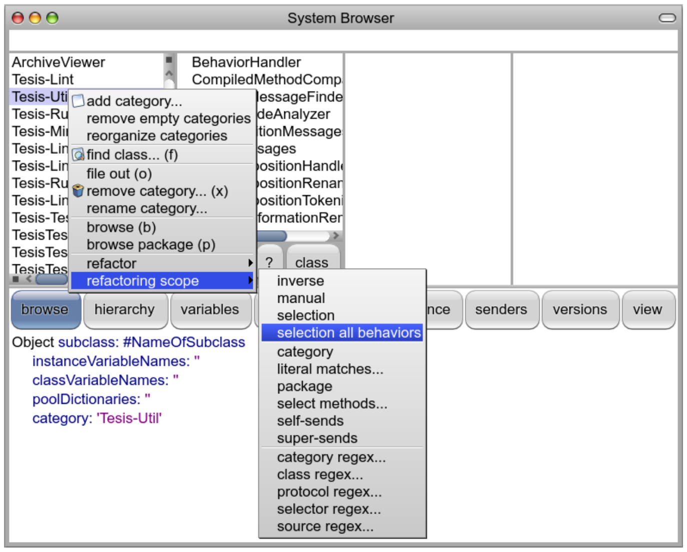
```
FileBrowser class»defaultRootNode
        ^DirectoryNode new path: '/'
```

Figure 62: Default *concrete root node* definition [OBWeb].

Note that **defaultRootNode** is a **DirectoryNode** instance, which matches with the **defaultMetaNode** i.e. the **Directory** *meta-node* [BDPW/07] [OBWeb].

Smalllint UI Implementation

Smalllint UI is a browser that shows the defined **Lint** *rules* and their *results* on the selected *environment* to be validated (after its execution finalization).



```
Object subclass: #NameOfSubclass
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Tesis-Util'
```

Figure 63: Menu option for selecting a refactoring scope from a **Class Browser**.

To run **Smalllint** the programmer must first define which set of *classes* will be validated. To do it, he must select any option under "*Refactoring Scope*" menu option in a **ClassBrowser**.

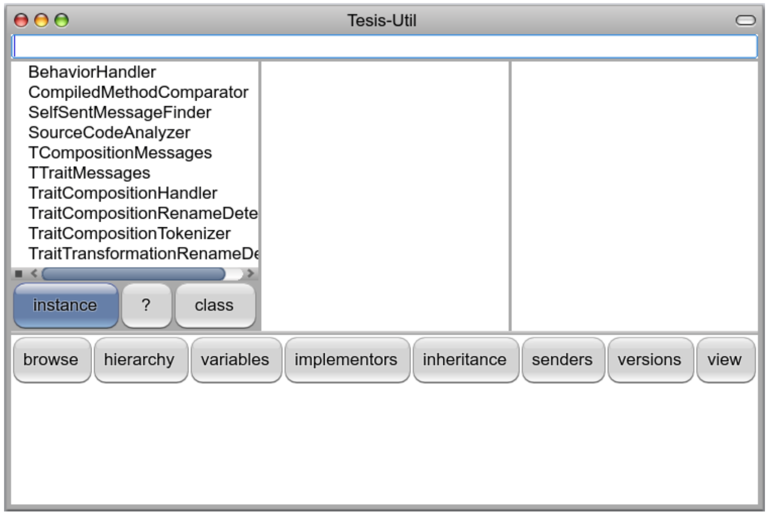
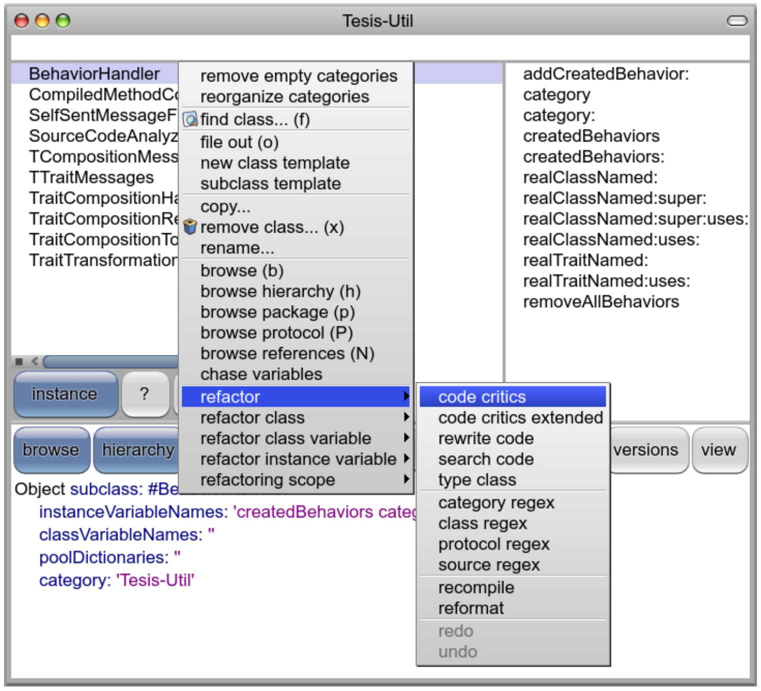


Figure 64: Browser presenting the selected "*Refactoring Scope*".

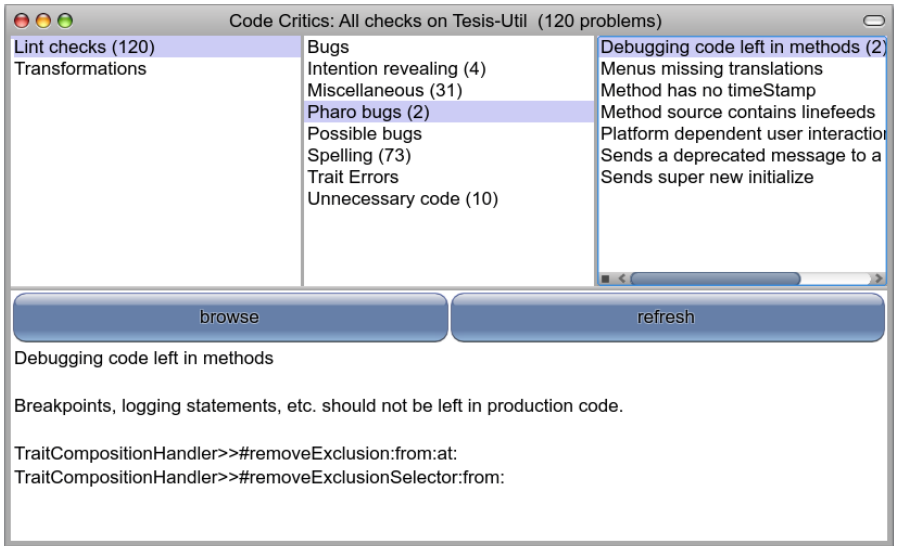
This action will open a browser showing all the selected *classes*.
After opening the browser displaying the selected "*Refactoring Scope*", the **Smalllint** tool can be run on the content of that scope, i.e. the selected "*Refactoring Scope*" will be the *environment* to be validated by the tool.



```
Object subclass: #Be
    instanceVariableNames: 'createdBehaviors categ
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Tesis-Util'
```

Figure 65: Browser menu options for running "*Code Critics*".

The **refactor>>code critics** menu option will open the **Smalllint** tool on the browsed *environment* (i.e. "*Refactoring Scope*").


Code Critics: All checks on Tesis-Util (120 problems)
Lint checks (120)
Transformations
Bugs
Intention revealing (4)
Miscellaneous (31)
Pharo bugs (2)
Possible bugs
Spelling (73)
Trait Errors
Unnecessary code (10)
Debugging code left in methods (2)
Menus missing translations
Method has no timeStamp
Method source contains linefeeds
Platform dependent user interactio
Sends a deprecated message to a
Sends super new initialize
browse
refresh

Debugging code left in methods

Breakpoints, logging statements, etc. should not be left in production code.

TraitCompositionHandler>>#removeExclusion:from:at:
TraitCompositionHandler>>#removeExclusionSelector:from:

Figure 66: "*Code Critics*" (**Smalllint**) browser.

The browser displays all the available *rules* grouped by *categories* and displays the *rule execution results* after all the *rules* completed the evaluation on the analyzed *environment*.
The browser itself is implemented by **ORLintBrowser** *class*. It defines a *meta-graph* with the *rules* data structure to navigate through the *rules* groups reaching individual *rules*.

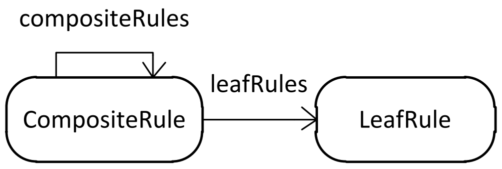


Figure 67: Smalllint browser *meta-graph*.

The browser's *meta-graph* has two different *node* types: *composite rule*, which represents *composite* **Lint** *rules*, and *leaf rule*, which represents *concrete lint rules*. The *meta-graph* can be navigated from a *composite rule node* to other *composite rule nodes* through a **compositeRules** *transition*, and a set of *leaf rule nodes* can be reached from a *composite node* through **leafRules** *transition* (the *meta-nodes* are instances of **OBMetaNode** *class*).
The "*concrete*" *graph* for the **Lint** browser is modeled with *nodes* wrapping the different kinds of **Lint** domain *rules* i.e. *composite* and *concrete* **Lint** *rules*. The *node class* for *composite* **Lint** *rules* is **ORCompositeLintNode**, and for *concrete* ones is **ORBasicLintNode**.
ORCompositeLintNode implements **compositeRules** and **leafRules** *messages* which return collections of *nodes* of the expected *rule type* (i.e. *composite* or *leaf*) for the chosen transition.
After ending the *rules* execution, the browser refreshes the displayed data, enabling the display of the computed *rule results*. The *rule results* are implemented by **BrowserEnvironment** or one of its *subclasses* and contain the entities that resulting positive in the *rule* evaluation.

4.2 Smalllint Extension for Traits Error Detection

Adapting current **Smalllint** implementation to detect *Traits related errors* implies identifying problems and limitations that do not allow **Smalllint** to check *Traits errors*, and defining change requirements for **Smalllint** to achieve the *Traits related error* checking functionality.

This section will list the identified **Smalllint** limitations and problems and the requirements to solve them.

Later, each limitation/requirement will be described as well as the change which satisfies the requirement.

4.2.1 Smalllint Limitations and Requirements

To implement *Traits automatic error checking*, there are some limitations and problems **Smalllint** must solve. These are the **Smalllint** limitations:

- *Environments* only list *classes*.
- *Refactoring scope* selection browser only shows *classes*.
- **Smalllint** *rule* checker only checks *classes* and *methods* defined by *classes*.
- **Smalllint** *rules* only checks *classes* and *methods* defined by *classes*.
- Poor *error result* information displayed (Only shows which *class* or *method* has an *error*).
- Lack of *Traits* aspects inspection framework.

Based on these limitations the requirements to change **Smalllint** are the following:

- *Environments* must be able to list *classes* and *traits*.
- *Refactoring scope* browser must show *classes* and *traits*.
- **Smalllint** *rule* checker must be able to check *classes*, *traits* and *methods* defined by both entities.
- **Smalllint** *rules* must be able to check *classes*, *traits* and *methods* defined by both entities.
- Implement an improved **Smalllint** *error result* presentation mechanism.
- Implement a *Traits* aspect inspection framework.

4.3 Smalllint Extension Implementation

This section describes the work done for satisfying each of the requirements.

4.3.1 Environments must be Able to List Classes and Traits

Environments are used by **Smalllint**, mainly to represent the selected "*Refactoring Scope*" to be analyzed. The environments on **Smalllint** (and **Smallltalk**, since they are used in several applications/frameworks) are usually implemented as *subclasses* of **BrowserEnvironment**. Basically, an *environment* is a kind of set which contains *classes*, and gives functionality to iterate on them. The *classes* iteration functionality is implemented by **classesDo:** which is similar to **do:** *message* which is available for collections, but in this case it is specific to iterate on the *classes* included in the *environment*. This was valid before *Traits* since *Classes* were the only behavior or structure defining entities in a **Smalltalk** image.

```
BrowserEnvironment>>classesDo: aBlock
	self allClassesDo: [ :each |
		(self includesClass: each)
			ifTrue: [ aBlock value: each ] ]
```
Figure 68: classesDo: evaluates **aBlock** on each *class* included in the **BrowserEnvironment**.

classesDo: *method* shows some other *class* related *messages* like **allClassesDo:** and **includesClass:** The objective of this change is to extend *environments* (**BrowserEnvironment** and its *subclasses*) with the ability of iterating on the included *classes* and/or *traits*. The shared aspect between *Traits* and *Classes* is that both define object behavior, and, after this, will be considered as *Behavior* any entity which defines behavior (i.e. defines behavior in a *meta-level*. It does not apply to a *compiled method*). Considering this definition, the message to iterate on the behaviors included in an *environment* will be named **behaviorsDo:**

```
BrowserEnvironment>>behaviorsDo: aBlock
	self allBehaviorsDo: [ :each |
		(self includesBehavior: each)
			ifTrue: [ aBlock value: each ] ]
```
Figure 69: behaviorsDo: *method* for iterating on an *environment's behaviors* (*classes* and *traits*).

Figure 69 shows **behaviorsDo:** implementation, where we can also see **allBehaviorsDo:** and **includesBehavior:** *messages*, which are similar to **allClassesDo:** and **includesClass:** *messages*, but applicable to any *behavior* instead of just *classes*.

4.3.2 Refactoring Scope Selection Browser Must Show Classes and Traits.

When a "*Refactoring Scope*" is selected, it is displayed on a browser. Currently, this browser is implemented by **ORClassBrowser** which defines a *meta-graph* as follows:

```
ORClassBrowser>>defaultMetaNode
	| root |
	root := OBMetaNode named: 'Environment'.
	^ self buildMetagraphOn: root
```
```
OBCodeBrowser>> buildMetagraphOn: root
   ^ self
	buildMetagraphOn: root
	class: #classes
	comment: #comments
	metaclass: #metaclasses
```
```
OREnvironmentNode>> classes
	^ self browserEnvironment allNonMetaClasses collect: [ :each | each asNode ]
```
Figure 70: ClassBrowser *meta-graph* definition and *concrete node* message transition implementation.

This *meta-graph* defines the **classes** transition between the browsed *environment* and the displayed items, implying that the browsed items will be only the *classes* included in the *environment*.

A new browser has been implemented at **OR2BehaviorBrowser**, which displays all the *behaviors* included in the *environment*.

```
ORBehaviorBrowser>>buildMetagraphOn: root
^ self
       buildMetagraphOn: root
       class: #behaviors
       comment: #comments
       metaclass: #metaclasses
```

```
OREnvironmentNode>>behaviors
       ^ self browserEnvironment classNames collect:
                              [ :each | (Smalltalk at: each) asNode ]
```

Figure 71: New **BehaviorBrowser** based on the original **ClassBrowser**, but displaying all the *environment's behaviors*.

The new browser *meta-graph* defines **behaviors** transition instead of **classes** transition. In this way the browser displays all the *behaviors* instead of only c*l*asses.
Note that **classNames** implemented by the *environments* returns all the *behaviors* names, not only *classes*, and because of this, **behaviors** *message* implementation works properly (anyway it should be renamed for more clarity).

4.3.3 Smalllint Rule Checker Must be Able to Check Classes, Traits and Methods Defined by Both Entities.

SmalllintChecker has the responsibility of running the *rules* on the selected "*Refactoring Scope*", but it runs the *rules* on the *environment's classes* and explicitly filters any *trait* included on it. Because of this, **SmalllintChecker** was extended by **FullEnvironmentSmalllintChecker**, which overrides **run** *message*, extending the *rule* evaluation to the *traits* included in the "*Refactoring Scope*".

```
FullEnvironmentSmalllintChecker >>run
       rule resetResult.
       environment
              behaviorsDo: [ :class |
                     class isTrait
                            ifTrue: [
                                   self checkTrait: class.
                                   self checkMethodsForTrait: class ]
                            ifFalse: [
                                   self checkClass: class.
                                   self checkMethodsForClass: class ] ]
```

Figure 72: Enhanced **SmalllintChecker** which evaluates all the *behaviors* in the "*Refactoring Scope*".

Figure 72 shows that the *environment* content is iterated using **behaviorsDo:** instead of **classesDo:** and also shows the introduction of **checkTrait:** and **checkMethodsForTrait:** which will send **checkTrait:** and **checkTraitMethod:** to each *rule*, in the same way that **checkClass:** and **checkMethodsForClass:** currently sends **checkClass:** and **checkMethod:** to the *rules*.

4.3.4 Smalllint Rules Must be Able to Check Classes, Traits and Methods Defined by Both Entities.

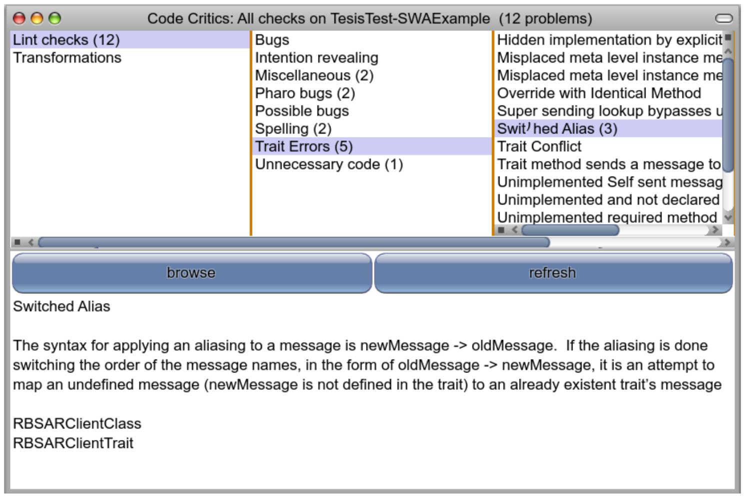
All the **Smalllint** *rules* protocol includes **checkClass:** and **checkMethod:** *messages* which are the *messages* sent by **SmalllintChecker** to evaluate each *rule*. If any *rule* wants to check *classes* or *methods, it* has to implement the corresponding *message*. Following with this schema, **Lint** *rules* protocol has been extended with **checkTrait:** and **checkTraitMethod:** *messages*. These *messages* will be sent by **FullEnvironmentSmalllintChecker** to each *trait* or *trait* defined *method* included in the *environment*.

4.3.5 Implement an Improved Smalllint Error Result Presentation Mechanism.

The original **Smalllint** *rule result* presentation consists of listing *classes* or *methods* which resulted positive on a *rule* evaluation.

Despite this approach seems to be good enough to check *classes*-based systems, it is not accurate to show exactly where the *error* happens (e.g. which sent *message* inside a *method* generates the *error*), and also does not consider the possibility of other constructions in the *class*, like a *traits composition use clause*.

The objective of the *rule result* presentation improvement is to let the user navigate through *rule results* in the same way that he navigates from *composite* to *leaf rules*. In this way, when a *rule* is selected, it will display in a new panel the positive *rule results* grouped by a common denominator, like the *class* or the *method* where the *errors* happen, (this can be defined by each *rule*). When one of these items is selected, another panel will appear and display all the *error* occurrences on that entity. This can be done iteratively until reaching the specific place where the *error* happens (i.e. it will present the *error* from the more general to the more specific location). It is also expected that all the previous *rules* will be able to present their results as they did before without any change.

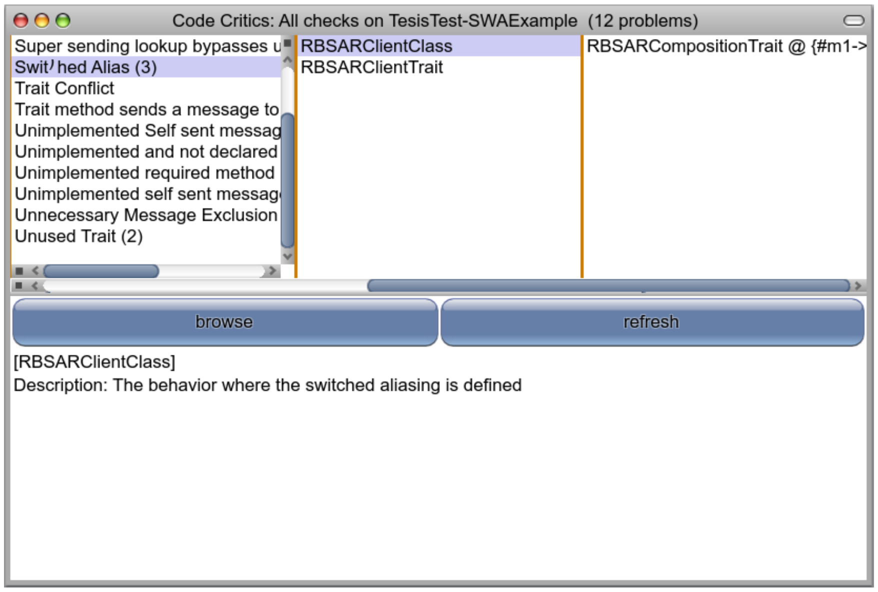

Code Critics: All checks on TesisTest-SWAExample (12 problems)
Lint checks (12)
Transformations
Bugs
Intention revealing
Miscellaneous (2)
Pharo bugs (2)
Possible bugs
Spelling (2)
Trait Errors (5)
Unnecessary code (1)
Hidden implementation by explicit
Misplaced meta level instance me
Misplaced meta level instance me
Override with Identical Method
Super sending lookup bypasses u
Switched Alias (3)
Trait Conflict
Trait method sends a message to
Unimplemented Self sent messag
Unimplemented and not declared
Unimplemented required method
browse
refresh

Switched Alias

The syntax for applying an aliasing to a message is newMessage -> oldMessage. If the aliasing is done switching the order of the message names, in the form of oldMessage -> newMessage, it is an attempt to map an undefined message (newMessage is not defined in the trait) to an already existent trait's message

RBSARClientClass
RBSARClientTrait

Figure 73: Enhanced **Smalllint** browser is compatible with the original *rule result* presentation schema.


Code Critics: All checks on TesisTest-SWAExample (12 problems)
browse
refresh

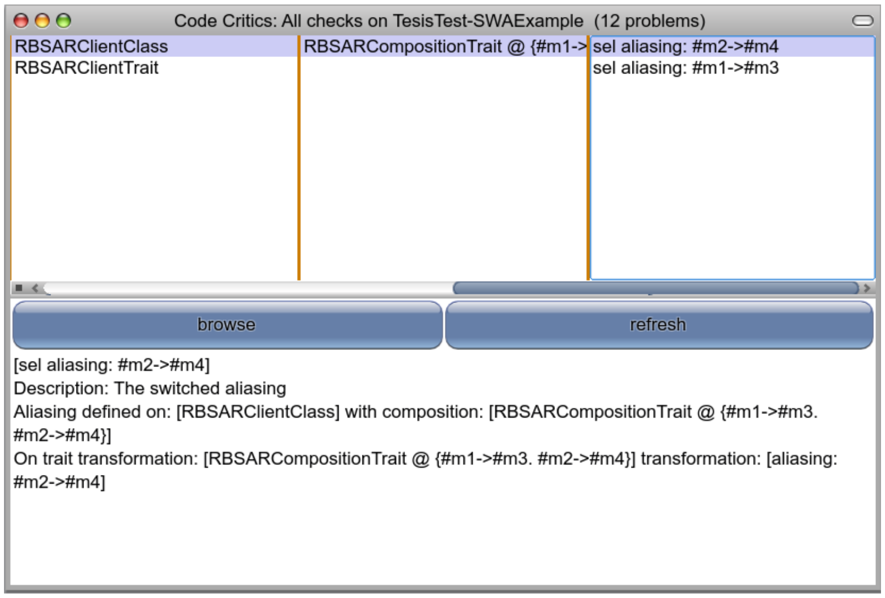
Super sending lookup bypasses u
Switʲhed Alias (3)
Trait Conflict
Trait method sends a message to
Unimplemented Self sent messag
Unimplemented and not declared
Unimplemented required method
Unimplemented self sent messag
Unnecessary Message Exclusion
Unused Trait (2)

RBSARClientClass
RBSARClientTrait

RBSARCompositionTrait @ {#m1->

[RBSARClientClass]
Description: The behavior where the switched aliasing is defined

Figure 74: Enhanced **Smalllint** browser *rule result* presentation.


Code Critics: All checks on TesisTest-SWAExample (12 problems)
browse
refresh

RBSARClientClass
RBSARClientTrait

RBSARCompositionTrait @ {#m1->

sel aliasing: #m2->#m4
sel aliasing: #m1->#m3

[sel aliasing: #m2->#m4]
Description: The switched aliasing
Aliasing defined on: [RBSARClientClass] with composition: [RBSARCompositionTrait @ {#m1->#m3. #m2->#m4}]
On trait transformation: [RBSARCompositionTrait @ {#m1->#m3. #m2->#m4}] transformation: [aliasing: #m2->#m4]

Figure 75: Detailed description for an individual *rule result* in the lower panel.

To complete the proposed **Smalllint** extension, it is convenient to divide it into two aspects:

- **Smalllint UI** extension.
- New *rule result schema* for the extended **UI**.

Smalllint UI Extension

ORExtendedLintBrowser class implements the extended **Smalllint** browser. It redefined the previous **Smalllint** browser *meta-graph* implemented by **ORLintBrowser**.

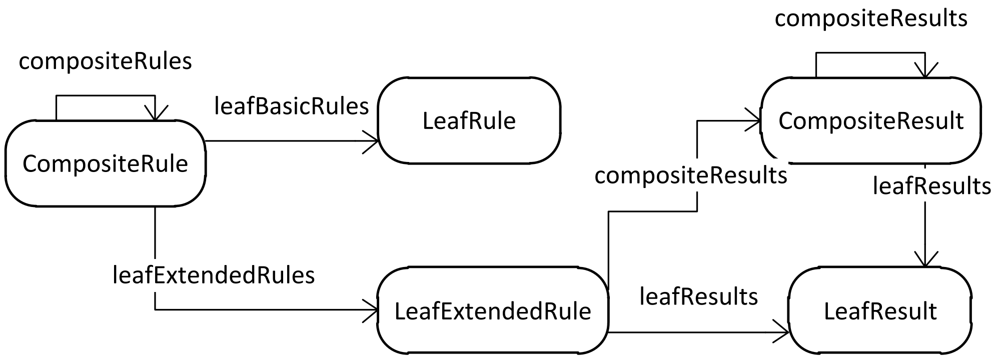


Figure 76: Extended **Smalllint** browser *meta-graph*.

This new *meta-graph* defines a **CompositeRule** *node* for the grouped **Lint** *rules*, and a **LeafRule** *node* for *rules* who does not implement the extended *error result* presentation (e.g. previously existing *rules*). **LeafExtendedRule** *node* models the *rules* which implement the extended *error* presentation. Both **LeafRule** and **LeafExtendedRule** *nodes* model *concrete rules* and not grouped ones. A tree structured *result* can be navigated from a **LeafExtendedRule** *node*. In this tree structured *result*, **CompositeResult** *node* represents the *internal nodes*, and **LeafResult** *node* represents the *leaf nodes*.

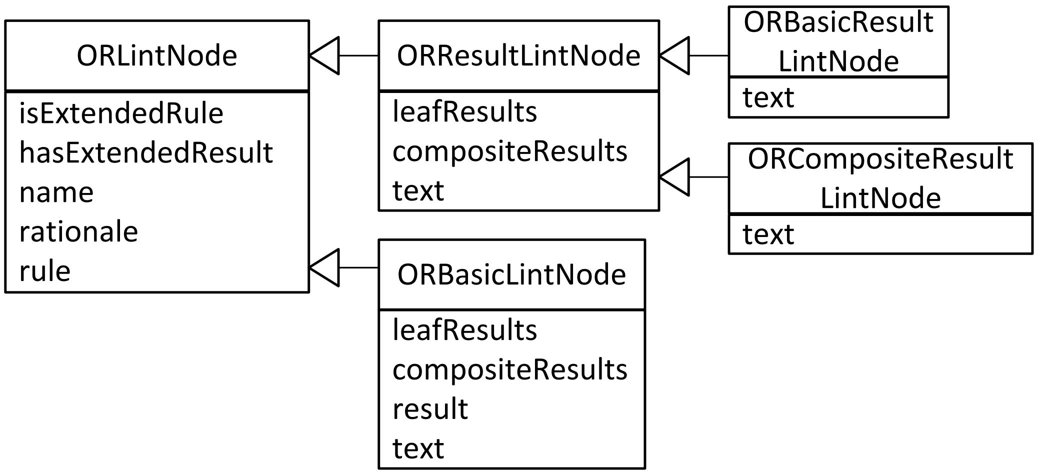


Figure 77: OmniBrowser concrete n*ode classes* representing each the **Smalllint** *rules* and *result nodes* in the **Smalllint** browser.

Figure 77 shows the *concrete graph* implementation that was also adapted to comply with the new *meta-graph*. The *concrete rule node class* **ORBasicLintNode** was extended with **leafResults** and **compositeResults** transition *messages* to enable the navigation from the *rule node* to the first *result node*. To model the *result* presentation *nodes*, **ORLintNode** was extended by **ORResultLintNode**, which also defines **leafResults** and **compositeResults** transition *messages* to implement the *meta-graph* defined *transitions*. The extended **Smalllint** *result nodes* also implements **text** *message* to present the current selected *node* on the lower panel in the same way the *rule result* was presented before. Since this *result* presentation is different for *leaf* and *non-leaf nodes* two **OR2ResultLintNode** *subclasses* were defined, **ORCompositeResultLintNode** and **ORBasicResultLintNode**, implementing different **text** versions.

New Rule Result Schema for the Extended UI

The original *rule result* schema does not fit with the new navigational rule result presentation schema and it also strongly couples the *rule result* with what **Smalllint UI** expects as *result*. In the old *rule result* schema, **Smalllint** expected *classes* or *methods* into the *rule result environment*, forcing the *rules* to produce *classes* or *methods* as *results* to add into the *rule result* set. An objective in the new *rules result* schema is that the *rules results* can present more information about the *errors* instead of just in what *class* or *method* is the *error* produced. To achieve this, the *rule* logic will be decoupled from the **Smalllint** framework and will produce an arbitrary object with all the information about the displayed *errors.* This *result* object will be adapted by the *rule* to a *rule result* structure compatible with the new *rule result* presentation schema.

Structured Smalllint Rule Result

When a *rule* is evaluated, it gets a collection of *rule result* objects. Each *rule result* is an occurrence of the analyzed *error* and is modeled by an **ExtendedLintRuleResult** *subclass*. Once a **Smalllint** *rule* computes the *rule result* collection, it has to adapt it to a tree structure compatible with the extended **Smalllint** browser *meta-graph*.
Every *rule result* can be considered as a path from a *behavior* (a *class* or a *trait*) to the entity which produces the *error*, where each *node* is closer to the *error* location. For example, a "*Switched Message Aliasing*" *error* in a *class used trait composition clause* would be represented by a path like:

Behavior -> trait transformation included in the used trait composition -> erroneous message aliasing mapping

Note that each path for the same *error type* will have the same length since there is not any involved recursive structure.

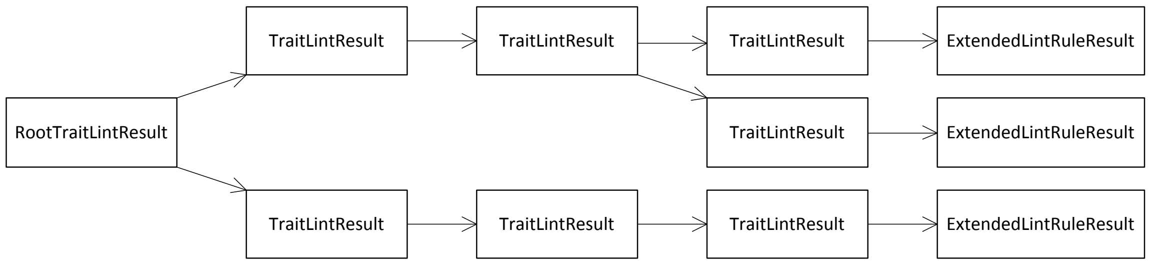


Figure 78: Smalllint *rule result* set adapted to a tree like structure.

Figure 78 shows a group of *rule results* transformed to a path like representation, combined in a way that each repeated *node* appears just one time, and with the addition of a *root node* which represents the entire *rule result* from where all the *rule result paths* start. The resulting tree like structure represents the adapted *rule result set* is compatible with the one described by the **Smalllint** browser *meta-graph*.

In this tree *rule result* representation, each *leaf node* on the tree represents a different *rule result* since all the *rule results* differ in at least one property value, defining each *rule result* a different path. Considering this, each *leaf node* includes its corresponding unique **ExtendedLintRuleResult** instance which will let the extended browser present a detailed *error* description when the *leaf node* where selected.

This tree-like structure is implemented by two *classes*: **RootTraitLintResult** and **TraitLintResult**. **RootTraitLintResult** is the root of the *result* tree, playing the role of the *rule result* set (similar to the role played by the *environments* as the *rule result* set in the previous schema). **TraitLintResult** implements each *node* of the *rule result* tree, containing all the information corresponding to the property assigned to the *node*. Each of these *nodes* will be adapted to be handled by **Smalllint** framework by **ORResultLintNode** instances which have been already described.

Rule Result Adaptation

The transformation from a set of **ExtendedTraitLintResult** instances to a tree structured *result* is done by an adaptable *rule result* builder schema, which is configured for each different **Smalllint** *rule*.

Each *rule* defines a *rule result* adapter by subclassing **LintResultBuilder**. This *rule result* adapter configures a chain of **LintResultNodeBuilder.** Each builder in the chain is responsible for building the *nodes* corresponding to a specific property, i.e. each level of the *rule result* tree.

Each **LintResultNodeBuilder** takes care of a property in the *rule results* to be transformed. The property value is obtained by sending a *message* to the adapted *rule result*. When the *node* builder receives a set with **ExtendedTraitLintResult** instances, all the *rule results* are grouped, having the *rule results* in each group their defining property value in common. Then, for each group, a **TraitLintResult** *node* is created with the information provided by the group defining property and all the *rule results* in the group are sent to the next **LintResultNodeBuilder** in the chain to be transformed. The next builder will return a collection of **TraitLintResult**, which will be the following *nodes* for the current group *node* in the tree structured *rule result*.

Example

The next "*Switched Message Aliasing*" *rule result* example is implemented by **SwitchedAliasRuleResult** *class* which is a *subclass* of **ExtendedTraitLintResult**.

The structure of this *rule result* is:

SwitchedAliasRuleResult

- *Error* defining *behavior.*
- *Erroneous trait transformation* at the *behavior's used trait composition.*
- Switched *message aliasing mapping.*

This *rule result* can be represented as a path like:

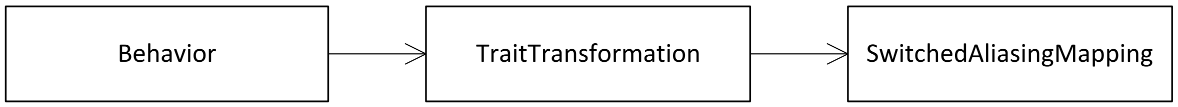


Figure 79: Path like representation of an individual "*Switched Message Aliasing*" *rule result*.

To obtain this path structure the **LintResultBuilder** will define a **LintResultNodeBuilder** chain with three *nodes*, one for each *rule result* property to be displayed.

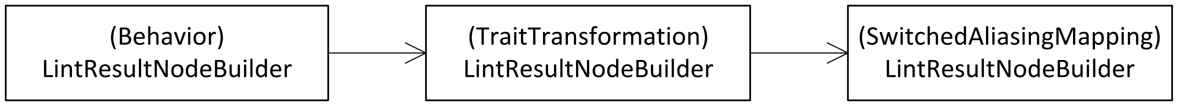

(Behavior) LintResultNodeBuilder
(TraitTransformation) LintResultNodeBuilder
(SwitchedAliasingMapping) LintResultNodeBuilder

Figure 80: LintResultNodeBuilder chain for building adapted representations of *rule* **ExtendedTraitLintResult** *results*.

Considering this example scenario

Object subclass: #ClientClass uses: CompositionTrait @ {#m1->#m3. #m2->#m4}	Trait named: #CompositionTrait uses: {}
Trait named: #ClientTrait uses: CompositionTrait @ {#m2->#m3}	m1 ^doSomething m2 ^doSomethingElse

Figure 81: Example code of a *class* containing positive results of "*Switched Message Aliasing*" *error*.

The "*Switched Message Aliasing*" **Smalllint** *rule* will return three positive *rule results*:

SwitchedAliasRuleResult • definingBehavior:ClientClass • traitTranformation: CompositionTrait@ {#m1->#m3. #m2->#m4} • switchedAliasMapping: (#m2->#m4)	SwitchedAliasRuleResult • definingBehavior:ClientClass • traitTranformation: CompositionTrait@ {#m1->#m3. #m2->#m4} • switchedAliasMapping: (#m1->#m3)	SwitchedAliasRuleResult • definingBehavior:ClientTrait • traitTranformation: CompositionTrait@ {#m2->#m3} • switchedAliasMapping: (#m2->#m3)

Figure 82: Positive "*Switched Message Aliasing*" *error rule results* from example at figure 81.

After adapting the retrieved *rule results*, the **Smalllint** tree structured *rule result* will be like:

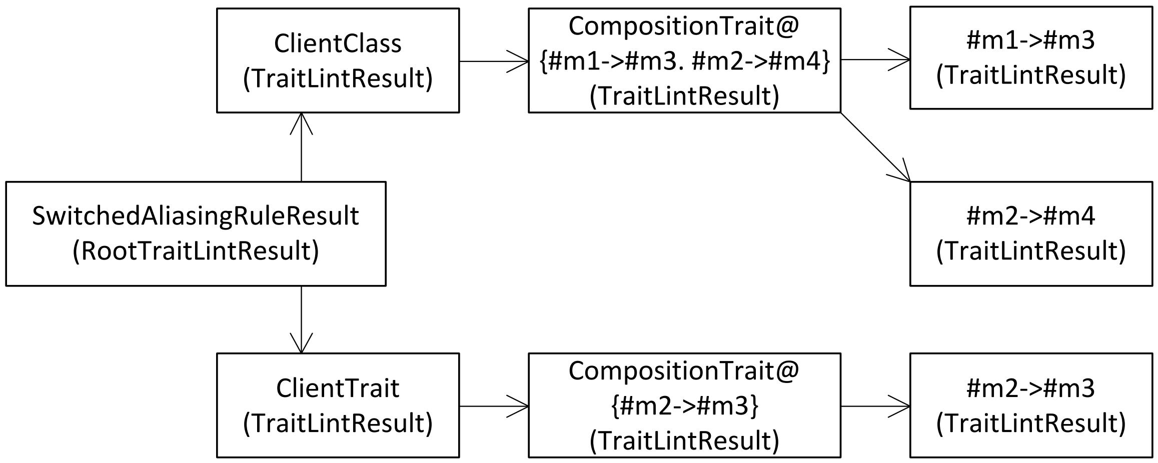

ClientClass (TraitLintResult)
CompositionTrait@ {#m1->#m3. #m2->#m4} (TraitLintResult)
#m1->#m3 (TraitLintResult)
SwitchedAliasingRuleResult (RootTraitLintResult)
#m2->#m4 (TraitLintResult)
ClientTrait (TraitLintResult)
CompositionTrait@ {#m2->#m3} (TraitLintResult)
#m2->#m3 (TraitLintResult)

Figure 83: Positive "*Switched Message Aliasing*" *error rule result set* adapted to be displayed by the extended **Smalllint** browser.

4.3.6 Implement a Traits Aspect Inspection Framework

In the *Traits* model, there are two main entities capable of defining behavior: *Class* and *Trait*, both define *method dictionaries* and can be considered *behavior* entities. In fact, both *inherit* from a *behavior class*, **Behavior** and **TraitBehavior**. They do not *inherit* from a common *class* but both use **TPureBehavior** *trait*. There are other two entities with similar capabilities to *Traits*: **TraitTransformation** and **TraitComposition**, both of them can be used as argument of a *trait composition use clause*, but do not share *behavior* with ***Trait*** *class*. All these four entities have some similar responsibilities, and some specific ones, but since some of them can play the same role, would be useful to have a common *protocol* to simplify the access to some properties depending on which role is performed.

Instead of modifying the mentioned entities protocol, it has been decided to define a set of new entities which will give access to the entities properties through a unified *protocol*. These new entities act like *inspectors* or *mirrors* of the original entities. Since *inspector* is a central concept with a specific meaning on **Smalltalk**, these new entities are going to be called mirrors.

Several entity aspects have been identified, each of them with an associated *protocol*:

- **Provided Messages:** Is the set of *messages* or *protocol* the entity can provide to other entities, using *Traits* use or *Class inheritance* mechanisms. In the case of *Class* and *Trait*, it is the entity *protocol* itself. In the case of *trait transformation* or *trait composition*, it is the set of *messages* provided to a *behavior* which uses the *trait transformation* or *trait composition*, considering the *excluded* and *aliased messages*. The conflicting *messages* in a *trait composition* are not included as *provided messages*.
    - **Example Aspect Accessing Messages: providedMessages**, **localProvidedMessages**
- **Compiled Method accessing:** Are the *compiled methods* associated to the entity's *provided messages*. For *Class* and *Trait*, this mapping is done using the *method dictionary*. For *trait transformations* the *message aliasings* or *exclusions* are considered. For *trait composition* the *conflicting messages* are also excluded.

    - **Example Aspect Accessing Messages: compiledMethodProvidedAt:**, **selectedMethodAt:**
- **Required Messages:** It is the set of *messages* the entity needs, and should acquire via implementation or trait use. There are two different kinds of *required messages*, *explicitly required message*s and *implicitly required messages*, depending on if there is a *message* implementation declaring the requirement (*explicit*), or if it is just a **self-sent** *message* which has not been implemented nor provided by other entity (*implicit*). The *required messages* are originated by *Traits*, but can propagate to any entity with a *trait composition* use clause.
    - **Example Aspect Accessing Messages: requiredMessages**, **implicitlyRequiredMessages**, **explicitlyRequiredMessages**
- **Conflicting Messages:** It is the set of *messages* which are defined by multiple sources in a *trait composition*. The *conflicting messages* can be also present in a *Class* or *Trait* if they do not resolve the conflict generated on its *used trait composition*.
    - **Example Aspect Accessing Messages: conflictingMessages**
- **Overriden Messages:** It is the set of messages provided to a *Class* or *Trait*, by *inheritance* or *trait use*, but for which the entity has provided its own implementation.
    - **Example Aspect Accessing Messages: overridenMessages**
- **Transformations:** Are the *message aliasings*, *message exclusions* and *message renamings* defined by an entity that can be used as a *trait composition* in a *trait composition use clause* i.e. *trait*, *trait transformation* and *trait composition*.
    - **Example Aspect Accessing Messages: transformations**, **aliases**, **exclusions**, **renames**

Four *mirrors* were defined grouping the identified different aspects:

Mirror	**Protocols implemented**
BehaviorMirror	Provided Messages, Compiled Methods, Overriden Messages, Conflicting Messages
TraitMirror	Provided Messages, Compiled Methods, Overriden Messages, Conflicting Messages, Transformations
TraitTransformationMirror	Provided Messages, Compiled Methods, Required Messages, Transformations
TraitCompositionMirror	Provided Messages, Compiled Methods, Required Messages, Conflicting Messages, Transformations

Figure 84: Each *trait* entity *mirror* and the aspects *protocols* which each of them have to implement.

These mirrors unify the protocols for different entities depending on the role they are playing. A situation where mirrors are useful is when a *trait transformation* has to be handled. In this situation, the *trait transformation mirror* has a single protocol, does not matter if the *trait transformation* is a *message aliasing* or a *message exclusion*. Another situation were *mirrors* are useful is when a *used trait composition* is analyzed, since a *trait*, a *trait transformation* or a *trait composition* can play a *trait composition* role. This unified protocol simplifies the access to the entities properties and helps on the rules implementation.

4.4 Smalllint Traits Error Detection Rules Implementation

After the implementation of the described changes, **Smalllint** is capable of checking *Traits related errors*, without losing the ability of checking all the previously defined *rules*. Fourteen *rules*, at least one from each *traits error category* have been implemented using the framework extension as a mode of example. The implemented *rules* are listed next, including a brief description of the input, output, *rule* implementation details, *rule* algorithm used and *rule result* presentation for the new **Smalllint** schema.

4.4.1 Switched Message Aliasing

Input: A *behavior* (can be a *class* or *trait*).

Output: A collection including all the *switched message aliasing* occurrences defined in the *behavior's used trait composition*. In case the received *behavior* does not use any *trait composition*, it will return an empty collection.

Each switched *message aliasing* occurrence is defined as follows:

SwitchedAliasing:

1. **Behavior:** It is the *behavior* which uses the *trait composition* that defines the *switched message aliasing*.
2. **SelectedAliasing:** It is *message aliasing* which defines the switched *message aliasing mapping*.
   2.1. **TraitTransformation:** It is the *trait transformation* from the *behavior's used trait composition* which includes the switched *message aliasing mapping*.
   2.2. **Aliasing association:** It is the switched *message aliasing mapping* itself.

Algorithm:

1. Get the received *behavior's used trait composition* **TC**.
2. For each *trait transformation* **TT** defined in **TC**.
   2.1. Let **TR** be **TT's** transformed *trait*.
   2.2. For each *message aliasing mapping* **ATT** defined in **TT**.
      2.2.1. Let **oldM->newM** be **ATT's** message mapping.
      2.2.2. Select **ATT** as a switched *message aliasing mapping* if **oldM** is included in **TR** *protocol* and **newM** is not included in **TR** *protocol*.

Main objects for this rule:

1. **BehaviorSwitchedAliasingDetector**: It detects the switched *message aliasings* in a *behavior's used trait composition* and creates the *rule* **SwitchedAliasing** *results*.
2. **TraitCompositionSwitchedAliasingDetector**: It retrieves the switched *message aliasings* defined in the *trait transformations* of a *trait composition*.

UI:

The detected switched *message aliasing results* are grouped and presented in three levels:

1. The *behavior* where the switched *message aliasing* is defined.
2. The *trait transformation* from the *behavior used trait composition*, where the switched *message aliasing mapping* is defined.
3. The switched *message aliasing mapping* itself (includes a detailed description of the individual *result*).

4.4.2 Unimplemented Self-Sent Message due Message Renaming

Input: A *behavior* (can be a *class* or *trait*).

Output: A collection including a behavior's **self-sent** *messages* not implemented because a valid *message* implementation provided by the *behavior's used trait composition* has been renamed.

Each **self-sent** *message* not available due *message rename* is defined as follows:

RenamedAndSentMessage:

1. **SentMessage**: It is the unimplemented **self-sent** *message*, including the *behavior* and the *method* from where it is sent.
   - 1.1. **Behavior**: It is the *behavior* where the **self-sending** *message method* is available.
   - 1.2. **SelectedMethod**: It is the *behavior's* **self-sending** *message method* and its associated *message* name.
   - 1.3. **SentMessage**: It is the unimplemented **self-sent** *message* which is sent from **SelectedMethod**.
2. **SelectedTransformationBehavior**: It is the *message rename* that makes the **self-sent** *message* to be undefined in the *behavior's* **self-sending** *method*.
   - 2.1. **Behavior**: It is the *behavior* for which its *used trait composition* includes the *trait transformation* that defines the *message rename*.
   - 2.2. **SelectedRename**: It is the *message rename* that makes the **self-sent** *message* to be undefined in the *behavior's* **self-sending** *method*.
     - 2.2.1. **TraitTransformation**: *It is the trait transformation* that defines the *message rename*.
     - 2.2.2. **newMessage**: It is the new *message* name for the renamed *message*.
     - 2.2.3. **oldMessage**: It is the old *message* name for the renamed *message*.

Algorithm:

1. Get all original message names of the renamed *messages* defined in the *behavior's used trait composition*.
2. Get all the **self-sent** *messages* sent from any of the *behavior's* available *methods* (the *behavior's* available *methods* can be defined in the *behavior* itself or acquired from a *superclass* or from its *used trait composition*).
3. Get all the *behaviour protocol messages*.
4. Get the *behavior's* unimplemented **self-sent** *messages* (set(2) - set(3)).
5. Get the *behavior's* unimplemented **self-sent** *messages* due *message renaming* (set(1) ∩ set(4)).

Main objects for this rule:

1. **UnimplementedSelfSentMessageDueRenamingDetector**: It implements the *rule* algorithm and creates the *rule's* **RenamedAndSentMessage** *results*.
2. **BehaviorRenameDetector**: It detects the *message renames* defined in the analyzed *behavior's used trait composition*.
3. **SelfSentMessageFinder**: It detects **self-sent** *messages* from the analyzed *behavior's* available *methods*.

4. **BehaviorMirror**: It analyses various aspects a *behavior*, in this case is used to get all the analyzed *behavior* protocol, including the *inherited* messages and their defining classes.

UI:

The renamed and sent *message results* are grouped and presented in three levels:

1. The *behavior* for which it's *used trait composition* defines the *message rename*.
2. The *behavior* where the **self-sent** *message* is originally defined.
3. The *method* where the unimplemented **self-sent** *message* is sent (includes a detailed description of the individual unimplemented **self-sent** *message* due *message renaming result*).

4.4.3 Unimplemented Self-Sent Message due Message Exclusion

Input: A *behavior* (can be a *class* or *trait*).

Output: A collection including all the **self-sent** *messages* from the *methods* available at the received *behavior*, for which there is no implementation available because a valid *message* implementation provided by a *trait* in the *behavior's used trait composition* has been excluded.

Each **self-sent** *message* not available due *message exclusion* is defined as follows:

RemovedAndSentMessage:

1. **SentMessage:** It is the unimplemented **self-sent** *message*, including the *behavior* and the method from where it is sent.
   - 1.1. **Behavior:** It is the analyzed *behavior* where the **self-sending** *message method* is available.
   - 1.2. **SelectedMethod:** It is the *behavior's* **self-sending** *method* and its associated *message* name.
   - 1.3. **SentMessage**: It is the unimplemented **self-sent** *message* which is sent from **SelectedMethod**.
2. **SelectedTransformationBehavior:** It is the *message exclusion* that makes the **self-sent** *message* to be undefined in the *behavior's* **self-sending** *method*.
   - 2.1. **Behavior:** It is the *behavior* for which its *used trait composition* includes the *trait transformation* that defines the *message exclusion*.
   - 2.2. **SelectedExclusion:** It is the *message exclusion* that makes the **self-sent** *message* to be undefined in the *behavior's* **self-sending** *method*.
     - 2.2.1. **TraitTransformation:** It is the *trait transformation* that defines the *message exclusion*.
     - 2.2.2. **Exclusion:** It is the *excluded message* which makes the **self-sent** *message* to be undefined.

Algorithm:

1. Get all the *excluded messages* defined in the *behavior's used trait composition*.
2. Get all the **self-sent** *messages* from any of the *behavior's* available *methods* (the *behavior's* available *methods* can be defined in the *behavior* itself or acquired from a *superclass* or from its *used trait composition*).
3. Get all the *behaviour protocol messages*.
4. Get the *behavior's* unimplemented **self-sent** *messages* (set(2) - set(3)).
5. Get the *behavior's* unimplemented **self-sent** *messages* due *message exclusion* (set(1) ∩ set(4)).

Main objects for this rule:

1. **UnimplementedSelfSentMessageDueExclusionDetector:** It implements the *rule* algorithm and creates the *rule's* **RemovedAndSentMessage** *results*.
2. **SelfSentMessageFinder**: It detects the **self-sent** *messages* from the *behavior's* available *methods*.
3. **BehaviorMirror**: It analyses various aspects a *behavior*, in this case is used to get all the analysed *behavior's protocol*, including the *inherited messages* and their defining messages.

4. **TraitCompositionMirror**: It analyses various aspects of a *trait composition*, in this case is used to get the *message exclusions* defined in the analyzed *behavior's used trait composition*.

UI:

The excluded and sent *message results* are grouped and presented in three levels:

1. The *behavior* for which it's *used trait composition* defines the *message exclusion*.
2. The *behavior* where the **self-sent** *message* is originally defined.
3. The *method* where the unimplemented **self-sent** *message* is sent (includes a detailed description of the individual unimplemented **self-sent** *message* due *message exclusion result*).

4.4.4 Misplaced Meta-Level Class Message Aliasing

Input: A *behavior* (can be a *class* or *trait*).

Output: A collection including all the *message aliasings* defined in the *behavior's instance message side used trait composition*, but which are applicable to the *behavior's class side used trait composition*.

Each misplaced *meta-level instance message aliasing* is defined as follows:

WrongMetalevelTransformation:

1. **SelectedTransformationBehavior:** It is the *message aliasing* defined in *behavior's instance message side used trait composition*.
   1.1. **Behavior**: It is the *behavior* where the misplaced *meta-level message aliasing* is defined.
      1.1.1. **SelectedAliasing:** It is the misplaced *meta-level message aliasing* and its defining *trait transformation*.
         1.1.1.1. **traitTransformation:** It is the *trait transformation* which defines the misplaced *message aliasing*.
         1.1.1.2. **Aliasing:** It is the misplaced *meta-level message aliasing mapping*.

Algorithm:

1. Get all the *instance message* side *behavior's used trait composition message aliasings*.
2. Get from set(1) all the *message aliasings* for which its defining *trait transformation* does not define the *aliased message*.
3. For each *message aliasing* **MA** from set(2).
   3.1. Get *trait* **T** from the *message aliasing* **MA**.
   3.2. Get from **T** its *classTrait* **CT**.
   3.3. If there a *trait transformation* **TT** in the *behavior's class message side* which transforms **CT**.
      3.3.1. If **MA** applies to **TT** then add **TT** to the *rule result* set.

Main objects for this rule:

1. **MisplacedInstanceMethodAliasingDetector:** It implements the *rule* algorithm. Verifies if a *message aliasing mapping* applies to a *trait transformation* in a *class* or *instance message side* of a *behavior's used trait composition*.
2. **TraitCompositionMirror:** It analyses different aspects of a *trait composition*. In this case it is used to get *instance* and *class meta-level behavior's used trait composition* provided protocols and defined *message aliasings*.

UI:

The misplaced *meta-level instance message aliasing results* are grouped and presented in two levels:

1. The *behavior* where the misplaced *meta-level* message aliasing is defined.
2. The misplaced *meta-level message aliasing* (includes a detailed description of the individual misplaced *meta-level instance message aliasing result*).

4.4.5 Misplaced Meta-Level Class Message Exclusion

Input: A *behavior* (can be a *class* or *trait*).

Output: A collection including all the *message exclusions* defined in the *behavior's instance message side used trait composition*, but which are applicable to the *behavior's class side used trait composition*.

Each misplaced *meta-level instance message exclusion* is defined as follows:

WrongMetalevelTransformation:

1. **SelectedTransformationBehavior:** It is the *message exclusion* defined in *behavior's instance message side used trait composition*.
   - 1.1. **Behavior**: It is the *behavior* where the misplaced *meta-level message exclusion* is defined.
     - 1.1.1. **SelectedExclusion:** It is the misplaced *meta-level message exclusion* and its defining *trait transformation*.
       - 1.1.1.1. **traitTransformation:** It is the *trait transformation* which defines the misplaced *message exclusion*.
       - 1.1.1.2. **Exclusion:** It is the misplaced *meta-level* excluded *message*.

Algorithm:

1. Get all the *instance message* side *behavior's used trait composition message exclusions*.
2. Get from set(1) all the *message exclusion* for which its defining *trait transformation* does not define the *excluded message*.
3. For each *message exclusion* **ME** from set(2).
   - 3.1. Get *trait* **T** from the *message exclusion* **ME**.
   - 3.2. Get from **T** its *classTrait* **CT**.
   - 3.3. If there a *trait transformation* **TT** in the *behavior's class message side* which transforms **CT**.
     - 3.3.1. If **ME** applies to **TT** then add **TT** to the *rule result* set.

Main objects for this rule:

1. **MisplacedInstanceMethodExclusionDetector:** It implements the *rule* algorithm. It verifies if a *message exclusion* applies to a *trait transformation* in a *class* or *instance message side* of a *behavior's used trait composition*.
2. **TraitCompositionMirror:** It analyses different aspects of a *trait composition*. In this case it is used to get *instance* and *class meta-level behavior's used trait composition* provided protocols and defined *message exclusions*.

UI:

The misplaced **meta-level** *instance message exclusion results* are grouped and presented in two levels:

1. The *behavior* where the misplaced *meta-level message exclusion* is defined.
2. The specific misplaced *meta-level message exclusion* (includes a detailed description of the individual misplaced *meta-level instance message exclusion result*).

4.4.6 Trait Composition Conflict Method

Input: A *behavior* (can be a *class* or *trait*).

Output: A collection including all the *behavior's trait composition conflict methods* (those that *self-send* **traitConflict** *message*) and their defining *trait transformations* from the *behavior's used trait composition* (the defining *trait transformations* can be an empty collection in case the *trait composition conflict method* where directly defined in the *behavior*).

Each *trait conflict* is defined as follows:

TraitConflict:

1. **SelectedMethod:** It is the *trait composition conflict* marked *method*.
   1.1. **Behavior**: It is the *behavior* that includes the *trait composition conflict* marked *method* as part of its *protocol*.
   1.2. **Message:** It is the *message* name for the *trait composition conflict* marked *method* included in the *behavior protocol*.
2. **ConflictingTraitTransformations:** It is the collection of the *trait transformations* included in the *behavior's used trait composition* that provides the conflicting *messages* that generates the *trait composition conflict method*.

Algorithm:

1. Get all the *behavior's* protocol.
2. Select all the *trait composition conflict* marked *methods messages* from set(1).
3. For each *message* **M** at set(2), select from the *behavior's used trait composition* the *trait transformations* that defines the conflicting *message* **M**.

Main objects for this rule:

1. **TraitConflictDetector:** It implements the *rule* algorithm, searches the *behaviour's trait composition conflict methods* and selects the *behavior's used trait composition* defined *trait transformations* that provides the conflicting *messages*.
2. **SourceCodeAnalyzer:** It analyses *compiled method* source code aspects. In this case it is used to find if a specific *method* **self-sends** a **traitConflict** *message*, indicating a *trait composition conflict method*.

UI:

The trait conflict *results* are grouped and presented in two levels:

1. The *behavior* where the *trait composition conflict* marked *method* is defined.
2. The *behavior's trait composition conflict* marked method (includes a detailed description of the individual *trait composition conflict result*).

4.4.7 Unnecessary Message Exclusion

Input: A *behavior* (can be a *class* or *trait*). It is required that the *behavior's used trait composition* is free of any *trait composition conflict*, since the *rule* algorithm detects the positive *results* through the *trait composition conflicts.*

Output: A collection including the unnecessary *message exclusions* defined at the *trait transformations* in the *behavior's used trait composition*. A *message exclusion* is not necessary if it can be removed from its defining *trait transformation* at the *behavior's used trait composition* without producing any *trait composition conflict*. In case that the *behavior's used trait composition* defines a *trait composition conflict*, the *rule* will return an empty *result set*.

Each unnecessary *message exclusion* is defined as follows:

UnnecesaryMessageExclusion:

1. **SelectedTransformationBehavior:** It is the *message exclusion* defined in *behavior's instance message side used trait composition*.
   1.1. **Behavior**: It is the *behavior* where the unnecessary *message exclusion* is defined.
      1.1.1. **SelectedExclusion:** It is the unnecessary *message exclusion* and its defining *trait transformation*.
         1.1.1.1. **traitTransformation:** It is the *trait transformation* which defines the unnecessary *message exclusion*.
         1.1.1.2. **Exclusion:** It is the unnecessarily excluded *message*.

Algorithm:

1. Get all the *message exclusion trait transformations* from the *behavior's used trait composition*.
2. For each *message exclusion* **me** at set(1).
   2.1. Create a new *trait composition* copying the *behavior's used trait composition* but removing **me** from it.
   2.2. Check if the new *trait composition* defines any *trait composition conflict*.

Main objects for this rule:

1. **UnnecesaryExclusionDetector:** It implements the *rule* algorithm. Creates new *trait compositions* stripping individual *message exclusions* from the original *trait composition* and check for any *trait composition conflict*.
2. **TraitCompositionHandler:** It manipulates *trait composition* components and allows modifying some of them to create a new *trait composition*. In this case it is used to remove individual *message exclusions* from the *trait transformations* in the handled *trait composition*.
3. **TraitCompositionConflictDetector:** It detects if there is any *message* provided to the *trait composition* by more than one *trait transformation*, defining a *trait composition conflict*.
4. **BehaviorTraitCompositionConflictDetector:** It gets the *trait composition conflict* from a *behavior's used trait composition* and check which of them are resolved overriding the conflicting *message* at the *behavior*.

UI:

The unnecessary *message exclusion results* are grouped and presented in two levels:

1. The *behavior* where the unnecessary *message exclusion trait transformation* is defined at its *used trait composition*.
2. The unnecessary *message exclusion* that could be removed without generating any *trait composition conflict* (includes a detailed description of the individual unnecessary *message exclusion result*).

4.4.8 Override with Identical Method

Input: A *behavior* (can be a *class* or *trait*).

Output: A collection including the *behavior's* locally defined *methods* that overrides an identical *message implementation* provided by the *behavior's used trait composition*.
Each one overrides with identical *method* is defined as follows:
OverridenWithIdenticalMethod:

1. **OriginalSelectedMethod:** It is the *message* implementation *method* provided by the *behavior's used trait composition*.
   1.1. **definingEntity:** It is the *trait transformation* that provides the overridden *message* to the *behavior's used trait composition*.
   1.2. **message:** It is the *message* name for the overridden *message* implementation *method*.
2. **OverridingSelectedMethod:** It is the locally defined *behavior's method* that overrides a *behavior's used trait composition provided message* with an identical *method* implementation.
   2.1. **definingEntity:** It is the *behavior* that defines the *method* which overrides a *used trait composition provided message* with an identical *method*.
   2.2. **message:** It is the *message* name for the overriding *method*.

Algorithm:

1. Get all the *behavior's used trait composition* provided *messages* that are overridden by the *behavior*.
2. For each message **M** from set(1) which does not define a *trait composition conflict* (If the overridden *message* defines a *trait composition conflict*, the *message* override fix the conflict).
   2.1. Get **M** If *behavior's used trait composition provided* **M** *message* implementation is equals to *behavior's* locally defined **M** *message* implementation.

Main objects for this rule:

1. **OverrideWithIdenticalMethodDetector:** It implements the *rule* algorithm. It gets the *trait composition* messages overridden by its defining *behavior*, check if the overrides fixes a *trait composition conflict* and gets the *compiled methods* from the *trait composition* and from the *behavior* to compare them.
2. **BehaviorMirror:** It analyzes different aspects of a *behavior*. In this case it is used to find the *behavior's used trait composition provided messages* that are overridden by the *behavior*. It also gets the overriding *message implementation method* defined by the *behavior*.
3. **TraitCompositionMirror:** It analyzes different aspects of a *trait composition*. In this case it is used to get the *behavior's used trait composition provided message* implementation *method* overridden by the *behavior*.
4. **CompiledMethodComparator:** It compares the parse trees of two *compiled methods*. In this case, it decides if the *trait composition provided message* implementation *method* and the overriding *behavior's* locally defined *method* are equivalent or not.

UI:

The override with identical *method results* are grouped and presented in three levels:

1. The *behavior* that overrides a *message* provided by its *used trait composition*.
2. The *trait transformation* from the *behavior used trait composition* that defines the overridden *message*.
3. The overridden *message* (includes a detailed description of the individual override with identical *method result*).

4.4.9 Unimplemented Required Message

Input: A *concrete class* (it should not declare any **subclassResponsibility** *message*).

Output: A collection including the class' *used trait composition required messages* not implemented at the client *class*. The *required messages* can be *explicitly required* (the *trait composition* defines the *message* and **self-sends requirement** *message* on the associated *method*) or *implicitly required* (the *trait composition* defines a *message* that **self-sends** a *message* for which there is not any implementation at the *trait composition* nor in its client *behavior*)
Each unimplemented *required message* is defined as follows:

UnimplementedRequiredMessage:

1. **Behavior:** It is the *class* which does not provide an implementation for one or more of its *used trait composition required messages*.
   1.1. **RequiredMessage:** It is the *required message* and all its definition details. A required *message* can be *implicitly* or *explicitly required*, created by a *message aliasing* or directly required by a *trait*. In this case this *required message* is not implemented at the *behavior*.
      1.1.1. **requiredMessage:** It is the *required message* with no implementation provided by the *used trait composition client concrete class*.
      1.1.2. **requiringMessage:** It is the *method* where the *message* requirement is defined through **self-sending requirement** *message* (*explicitly required*) or **self-sending** the *required unimplemented message* (*implicitly required*).

Algorithm:

1. Get all the *class' used trait composition required messages.*
2. Get all the *class' used trait composition explicitly required messages.*
3. Get all the *class provided message* (they can be locally defined, inherited or provided by its *used trait composition*).
4. Get all the locally defined *class provided messages.*
5. Get set(1) - set(3) the *class' used trait composition required messages* not implemented at the *client class*.
6. Get set(2) - set(4) the *class' used trait composition explicitly required messages* for which the requirement declaration *method* is the *required message* implementation available at the *client class*. i.e. the *explicit required message* is not provided by the *client class* or the *explicit required message* declaration is "hiding" a *superclass provided required message* implementation.
7. Get set(5) U set(6) all the *explicit* and *implicit required messages* not implemented at the client *class.*

Main objects for this rule:

1. **UnimplementedRequiredMethodDetector:** It implements the *rule* algorithm. It gets the *implicitly* and *explicitly required messages* and check on the *concrete class* if there is any valid implementation for them.
2. **BehaviorMirror:** It analyses aspects of a *behavior*. In this case is used to get the locally defined and the full *behavior protocol* and its implementation.

3. **TraitCompositionMirror:** It analyses aspects of a *trait composition*. In this case is used to get the *class' used trait composition explicitly* and *implicitly required messages*.

UI:

The unimplemented *required message results* are grouped and presented in three levels:

1. The *class* where some of its *used trait composition required messages* are not implemented.
2. The *required message* declaring *trait transformation* at the *class' used trait composition*.
3. The unimplemented *required message* (includes a detailed description of the individual *unimplemented required message result*).

4.4.10 Hidden Implementation by Explicitly Required Message

Input: A *class*.

Output: A collection including the *class' used trait composition explicitly required messages* that hide a *superclass' provided explicit required message* implementation.

Each hidden implementation by an *explicitly required message* is defined as follows:

HiddenImplementationByRequiredMessage:

1. **Behavior:** It is the *class* where some of its *used trait composition explicitly required messages* hide *superclass provided required message* implementations.
   1.1. **RequiredMessage:** It is the explicit *required message* that "*hides*" a *superclass provided* implementation for itself.
      1.1.1. **requiredMessage:** It is the *class' used trait composition required message*.
      1.1.2. **requiringMessage:** It is the *class' used trait composition required message* declaring *method* (since it is an *explicit required message*, **requiredMessage** and **requiringMessage** are equals).
   1.2. **SelectedMethod:** It is the *superclass provided required message* implementation hidden by the *class'* us*ed trait composition explicitly required message* declaration.
      1.2.1. **Behavior:** It is the *behavior* where the hidden *required message* implementation is defined.
      1.2.2. **Message:** It is the *message* name for the hidden *required message* implementation.

Algorithm:

1. Get all the *class' used trait composition required messages* not implemented at the *class*.
2. Get all the *class' superclass provided messages*.
3. Get set(1) ∩ set(2) the *messages* implemented at the *class' superclass* but overridden by a *class' used trait composition explicitly required message* declaration.

Main objects for this rule:

1. **HiddenByTraitRequiredMessageDetector:** It implements the *rule* algorithm. It gets the *class* and *superclass provided messages*, the class' *used trait composition explicitly required messages* and looks for the overridden messages.
2. **BehaviorMirror:** It analyses aspects of a *behavior*. In this case is used to get the *class* and *superclass provided messages*.
3. **TraitCompositionMirror:** It analyses aspects of a *trait composition*. In this case is used to get the *class' used trait composition explicitly required messages*.

UI:

The hidden implementation by *explicitly required message results* are grouped and presented in three levels:

1. The *class* where some of its *used trait composition explicitly required messages* hide valid *superclass provided required message* implementations.
2. The *trait transformation* from the *class' used trait composition* that defines the *explicit required message*.

3. The *explicitly required message* that hides a *superclass provided required message* implementation (includes a detailed description of the individual unimplemented *required message result*).

4.4.11 Not Explicitly Declared Required Message

Input: A *trait*

Output: A collection including the not explicitly declared *trait required messages* (i.e. the *implicit required messages*).

Each not explicitly declared *required message* is defined as follows:

ImplicitRequiredMessage:

1. **Behavior:** It is the *trait* where the not explicitly declared *required message* is sent.
2. **SentMessage:** It is the **self-sent** *message* not implemented by the *trait*.
3. **SelectedMethod:** It is the *trait* defined *method* which **self-sends** the *not* explicitly declared *required message*.

Algorithm:

1. Get all the *trait provided messages*.
2. For each *trait provided message* implementation get the **self-sent** *messages* not included in set(1).

Main objects for this rule:

1. **TraitMirror:** It analyses different aspects of a *trait*. In this case is used by get the *trait's implicitly required messages*.

UI:

The not explicitly declared *required message results* are grouped and presented in two levels:

1. The *trait* where the not explicitly declared *required messages* are **self-sent**.
2. The *trait* defined *method* where *implicitly required* message is sent (includes a detailed description of the individual not explicitly declared *required message result*).

4.4.12 Super-Sent Message Lookup Bypasses Used Trait Composition Provided Message

Input: A *class*.

Output: A collection including the *class'* available *methods* (can be not locally defined) that **super-sends** *messages* provided by the *class' used trait composition*. The *method lookup* bypasses the **super-sent** *messages* implemented at the *class' used trait composition* because they are considered as if they were defined in the *trait composition client* itself (*flattening property*).

Each **super-sending** *lookup* bypasses *used trait composition provided message* is defined as follows:

SuperSentTraitCompositionBypassedMethod:

1. **superSentMessage:** It is the **super-sent** *message* that bypasses the *class' used trait composition provided message*.
   - 1.1. **Behavior:** It is the *behavior* from where the **super-sent** *message* is sent, bypassing one of its *used trait composition provided* messages.
   - 1.2. **SentMessage:** It is the **super-sent** *message* for which the *method lookup* bypasses the *behavior's used trait composition provided message* implementation.
   - 1.3. **SelectedMethod:** It is the *behavior* defined *method* from where the bypassed *message* is **super-sent**.
     - 1.3.1. **Behavior:** It is the *behavior* where the **super-sending** *method* is defined.
     - 1.3.2. **Message:** It is the *message* name for the **super-sending** *method*.
2. **traitCompositionBypassedMessage:** It is the bypassed *class' used trait composition provided message*.
   - 2.1. **TraitTransformation:** It is the *trait transformation* included in the *class' used trait composition* which defines the bypassed *message*.
   - 2.2. **Message:** It is the bypassed *class' used trait composition* defined *message*.

Algorithm:

1. Get all the *class provided message* implementation *methods*.
2. For each *message* from set(1), get all the **super-sent** *messages*.
3. Get the *class' used trait composition provided messages*.
4. Get set(2) ∩ set(3) the set of all the **super-sent** *messages* that are also provided by the *class used trait composition* (since the *message* is **super-sent**, the *method lookup* will bypass the *class used trait composition provided message* implementation).

NOTE: The *class' used trait composition conflicting provided messages* ignored because there is no way to decide which implementation of the conflicting *trait composition provided messages* is meant to be extended.

Main objects for this rule:

1. **SuperSendLookupTraitCompositionBypassDetector:** It implements the *rule* algorithm. It gets the *class' used trait composition provided messages*, the *class* available *message*

implementations, theirs **super-sent** *messages* and checks if the **super-sent** *messages* bypass any *trait composition provided message*.

2. **BehaviorMirror:** It analyses different aspects of a *behavior*. In this case is used to get the *class'* available *message implementation methods*.
3. **TraitCompositionMirror:** It analyses different aspects of a *trait composition*. In this case is used to get the *class used trait composition provided messages*.
4. **SourceCodeAnalyzer:** It analyses *compiled methods* source code aspects. In this case is used to get the **super-sent** *messages* from *class'* available *message implementation methods*.

UI:

The **super-sending** lookup bypasses *used trait composition messages results* are grouped and presented in three levels:

1. The *class* that implements the *method* which **super-sends** the bypassed *message*.
2. The *message* name for the *method* which **super-sends** the bypassed *message*.
3. The bypassed *message* provided by the *class' used trait composition* (includes a detailed description of the individual **super-sending** lookup bypasses *used trait composition method result*).

4.4.13 Trait Method Super-Sends a Messages

Input: A *trait*.

Output: A collection including the *trait's* defined *methods* that **super-sends** *messages*.

Each *trait method* **super-sends** *messages* is defined as follows:

SentMessage:

1. **Behavior:** It is the *trait* which *defines* the *method* from where a *message* is **super-sent**.
2. **SentMessage:** It is the *message* **super-sent** from the *trait's* defined *method*.
3. **SelectedMethod:** It is the *trait's* defined *method* that **super-sends** a *message*.
   - 3.1. **Behavior:** It is the *trait* which defines the *method* from where a *message* is **super-sent**.
   - 3.2. **Message:** It is the **super-sending** *trait's* defined *method message* name.

Algorithm:

1. Get the *trait* defined *methods*.
2. For each *method* at set(1) get all its **super-sent** *messages*.

Main objects for this rule:

1. **TraitSuperMessageSendingDetector:** It implements the *rule* algorithm. Gets the *trait's* defined *methods* and search all the **super-sent** *messages* from each of them.
2. **TraitInspector:** It Analyses different aspects of a *trait*. In this case is used to get the *trait* defined *methods*.
3. **SourceCodeAnalyzer:** It analyses *compiled methods* source code aspects. In this case is used to get the *messages* **super-sent** from a *trait* defined *method*.

UI:

The *trait method* **super-sends** *message results* are grouped and presented in three levels:

1. The *trait* which defines the *method* from where a *message* is **super-sent**.
2. The *trait's* defined **super-sending** *method*.
3. The message **super-sent** from a *trait* defined *method* (includes a detailed description of the individual *trait method* **super-sends** *message result*).

4.4.14 Unused Trait

Input: A *trait.*

Output: A collection including the analyzed *trait* if it is not included in any *behavior's used trait composition*, an empty collection otherwise.

Each unused *trait* is defined as follows:

UnusedTrait:

1. **Trait:** Is the *trait* that is not included in any *behavior's used trait composition*.

Algorithm:

1. Get all the *behavior's used trait compositions* available in the *image*.
2. Get the analysed *trait* if it is not included in any *trait composition* on set(1).

Main objects for this rule:

1. **UnusedTraitDetector:** It implements the *rule* algorithm. It iterates the entire *image* looking for the analysed *trait* on every defined *used trait composition*.
2. **TraitCompositionMirror:** It analyses different aspects of a *trait composition*. In this case it is used to check if the analyzed *trait* is part of a *trait composition*.
3. **SystemNavigation:** It supports the navigation of the system. In this case it is used to go through all the defined *behaviors* in the **Smalltalk** *image* and analyze each *used trait composition*.

UI:

The unused *trait results* are grouped and presented in one level:

1. The *trait* that is not part of any *behavior's used trait composition* (includes a detailed description of the individual unused *trait result*).

5 Smalllint Traits Error Rules Use Results Analysis

After completing the *Traits error* typification and the **Smalllint** tool extension, including *Traits error rules* implementation, the next step is testing the extended **Smalllint** on real *Traits* using code. The intention of this testing is to verify how effective the extended tool is and the incidence of the typified *traits error types* on real scenarios.

The selected *Traits* using code to run the proposed test is a group of *packages* included in **Pharo** 1.0 *image* and *Traits* related *Thesis* implementations followed by this computer science department:

- "*Reingeniería de Jerarquías Polimórficas Utilizando Traits*" Acciaresi, Claudio;Buttarelli, Nicolás Martín [AB/07].
- "*Análisis de Lenguajes con Traits y sin Clasificación*" Campodonico, Diego [C/11].
- **Kernel-Classes** included at **Pharo 1.0** *image package*.
- **Traits (Traits-xxx)** included at **Pharo 1.0** *image package*.

Positive Smalllint Trait Related Results found					total
Package Rule	[AB/07]	[C/11]	Kernel-Classes	Traits-xxx	
Switched Message Aliasing	0	0	0	0	0
Unimplemented Self-Sent Message due Message Renaming	0	0	0	0	0
Unimplemented Self-Sent Message due Message Exclusion	0	0	0	0	0
Misplaced Meta-Level Class Message Aliasing	0	0	0	0	0
Misplaced Meta-Level Class Message Exclusion	0	0	0	0	0
Trait Composition Conflict Method	0	1	0	0	1
Unnecessary Message Exclusion	36	1	0	0	37
Override with Identical Method	45	816	4	1	866
Unimplemented Required Message	5	6	0	8	19

Hidden Implementation by Explicitly Required Message	3	0	0	1	4
Not Explicitly Declared Required Message	686	255	0	103	1044
Super-Sent Message Lookup Bypasses Used Trait Composition Provided Message	5	2	3	3	13
Trait Method Super-Sends a Message	4	1	0	3	8
Unused trait	0	6	0	0	6
Total	784	1088	7	119	1998

Figure 85: Smalllint *Traits error rules results* chart.

Figure 85 shows the **Smalllint** *Trait* related *rule results* evaluated on the proposed *packages*. The *results* show that the tool could find problems in all the analyzed packages. *Packages* included in **Pharo** image present less *traits errors*, possibly because their maturity, but anyway they still contain some *traits errors*. Other issue to note is that most *traits error* found belong to *types* that do not always mean problems in the object behavior.

The *trait errors* with more occurrences are:

- ***"Not Explicitly Declared Required Message"***: The lack of an *explicit required message* declaration makes more difficult to detect a *trait* requirements, but it does not affect its *behavior*.
- ***"Override with Identical Method"***: It does not affect the *behavior* in any aspect, but implies a problem since one of *Traits* objectives is to avoid code duplication, even more in this case where duplicated code can be removed without any concecuence.
- ***"Unnecessary Message Exclusion"***: It removes a *message* in a *trait composition*, without avoiding any *trait composition conflict*, but losing *behavior* provided the excluded *message*.
- ***"Unimplemented Required Message"***: It shows when a *concrete class* does not provide all the *required messages* from its *used trait composition*. One reason for this high *error* occurrence can be that the client *classes* are actually *abstract classes* but they have not been declared (no *subclass* responsibility *message* are defined in the *class*).

The more frequent *traits errors* with impact in object's behavior are:

- ***"Super-Sent Message Lookup Bypasses Used Trait Composition Provided Message"***: The code in a *trait composition client* bypasses message implementations provided by its *used traits*.

- “***Trait Composition Conflict Method***”: The *Trait composition client* shows unresolved *traits composition errors*.

It can be noted that most of the *traits errors* found are *errors* that do not explicitly impact on the *object's* behavior and that some *errors* with a few or no occurrences are more likely to happen in an early stage of a development, or with programmers not familiar with *Traits*. Examples of these *error types* are “***Switched Message Aliasing***”, “***Misplaced Meta-Level Instance Message Aliasing***”, “***Misplaced Meta-Level Instance message exclusion***” or “***Trait Composition Conflict Method***”, but since they have impact in the behavior, they can be found through usual testing.

6 Conclusions

This work identifies and typifies a set of *errors* produced by the use of *Traits*. It also describes and implements tools to detect the identified *error types*.

Traits error types and their categorization add knowledge to *Traits* using language domain, describing *Traits* domain elements like *trait composition*, *trait transformation*, etc. During this thesis it has been also identified other domain related concepts not previously identified, like *Behavior* for describing the shared role of *Class* and *Trait* as *behavior* defining entities.

The identified *Traits error types* and *Traits error categories* also provides an organizational frame which eases the understanding of *Traits* domain, and describes possible problems that can arise from *Traits* use.

Different methods have been applied to discover *Traits errors*, some *errors* have been discovered by plain *Traits* use, while others have been discovered by domain and implementation analysis, defining possible scenarios which do not follow the model preconditions, somehow similar to a **QA** testing process (this technique was more effective on finding *errors* than only *Traits* use).

Other important step on *Traits error* discovery process was the categorization of the *Trait errors*, partitioning *Traits* domain by their characteristics. This decision simplified the *traits error* discovery process, since the partition of the domain to analyze into smaller sub-domains reduced the domain complexity and the size of the elements to analyze looking for possible problem. *Traits error* categorization also avoided *Traits error types* overlapping, since each *error* can belong to a single category.

Smalllint use for checking *Traits related errors* on already working software have been more effective than the expectations, finding an important number of *errors*, which are difficult to find using usual testing techniques. It is also important to note that many *errors* with few occurrences in the tests are the ones that happen in early stages of development, or with inexperienced *Traits* programmers, like syntax or composition *errors*. In this case the use of the extended **Smalllint** helps on their early detection. Because of extended **Smalllint** effectiveness, its use on every software development stage will aid on Improving development time and quality on *Traits* using software.

Smalllint adaptation implied overcoming several **Smalllint** tool and language *environment* problems and limitations.

Smalllint was initially designed to work on an "*only classes*" *environment*, and had to be adapted for using it on an *environment* which includes *classes* and *traits*.

Original **Smalllint** *error* presentation was found not good enough, especially for *Traits errors* where the *error* is a combination of problems spread along several entities like *classes*, *superclasses* and *traits*. Because of these limitations an alternative *error* presentation schema has been proposed. This alternative *error* presentation schema adds information about *errors* and their locations, improving in this aspect the previous schema. Despite of this improvement, the alternative schema also implies a more complex process for **Smalllint** *rule* definition. This aspect can be subject of study for future works.

The language *environment* also suffered limitations on *Traits* handling. Despite the amount of *Traits* related works, some basic tools like several *browsers* and *environment* objects had problem on handling *Traits*. Some of them were adapted for *Traits* handling, but have been evident the lack of a unifying concept covering *Classes* and *Traits*. In this thesis the idea of *Behavior* is proposed. A *Behavior* has been defined as any entity able to define a behavior or *protocol* (i.e. it defines a *method dictionary*). Considering this, the already existing tools should be adapted to handle *Behaviors* instead of only *Classes* whenever is reasonable.
The lack of some unifying abstractions showed the need for defining *Mirrors* framework. It was usual that different entities were playing the same role under a defined scenario, without having a common *protocol* for that role. *Mirrors* framework was created trying to provide that unifying *protocols*, but avoiding changes to the current *Traits* framework. It would be useful to do a *Traits* framework refactoring to obtain a more consistent *Traits* model considering the experience using mirrors.

7 Future Work

There are several fields related to the work presented in this thesis that are worth mentioning

Traits model refactoring: This thesis shows the need of having a better type definition on the *Traits* model implementation, because the lack of a common *protocol* for the shared role or responsibility on different scenarios where different *Traits* related entities are used. The implemented *mirror* framework can be taken as a starting point on this subject. It could also be interesting to unify *Class* and *Trait* concepts under the unifying *Behavior* concept proposed in this thesis. Another item to consider is to add a *required message* declaration mechanism for **super-sent** *messages*.

Traits implementation improvement: We have found several problems on the current *Traits* implementation. It has some problems like letting the programmer do invalid *trait composition* definition, *explicit message requirement* declaration hiding inherited implementations, etc. *Traits* implementation should be improved trying to avoid preventable problems, providing in this way a more reliable and mature framework implementation to the programmer. Also the *environment* tools like browser and others should be adapted/extended to handle *Traits* properly.

Smalllint tool improvement: **Smalllint** tool is able to be improved, especially in its *error result* presentation mechanism. This improvement can be focused on letting the programmer define *rules* with more information about the *result*, without adding much complexity to the current *rule* definition methodology. Some possible topics to consider are:

- Add source code highlight to spot errors on the code.
- Extend *rule* analysis coverage to *Class* and *Trait* definition clauses and *used trait composition clause*.
- Extend *parse tree* analysis language to cover *Class* and *Traits* declaration and *trait composition use clause*.

Improve Class/Trait extension/versioning mechanism: During this thesis we have problems on adding *packages* or extending *classes* because incompatible *class* versions or missing *packages*. Other problem was that it was not possible to add *classes* and *traits* to test only purposes without polluting the image. A more flexible *image* management model is needed. It should allow the applications to have different *classes* or *traits* in the image without affecting other applications. In this way, it would be possible to avoid *classes* or *traits* version conflicts during package loads, and to create *classes* or *trait* for testing purposes without actually add them into the global image.

8 Bibliography and References

[AB/07]	Claudio Acciaresi, Nicolás Martín Buttarelli. *"Reingeniería de Jerarquías Polimorfitas Utilizando Traits"*. Universidad de Buenos Aires 2007.
[BDPW/07]	Alexandre Bergel, Stéphane Ducasse, Colin Putney and Roel Wuyts, "Meta Driven Browsers" Advances in Smalltalk, Proceedings of 14th International Smalltalk Conference (ISC 2006),LNCS, vol. 4406, Springer, 2007, pp. 134-156
[BLAC/04]	A. P. Black and N. Schärli. **Traits: Tools and Methodology**. In Proceedings ICSE 2004, pages 676–686. ACM Press, Mai 2004.
[BSD/02]	Andrew Black, Nathanael Schärli, and Stéphane Ducasse. **Applying traitsto the Smalltalk collection hierarchy**. Technical Report IAM-02-007, Institutf¨ur Informatik, Universit¨at Bern, Switzerland, November 2002. Also availableas Technical Report CSE-02-014, OGI School of Science & Engineering,Beaverton, Oregon, USA
[CDW/07]	Damien Cassou, Stéphane Ducasse, and Roel Wuyts. "**Redesigning with Traits: the Nile Stream trait-based Library**". 2007
[CodAnWeb]	http://en.wikipedia.org/wiki/Program_analysis_(computer_science)
[DiProblWeb]	http://en.wikipedia.org/wiki/Diamond_problem
[DNSWB/06]	Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts and Andrew Black, "Traits: A Mechanism for fine-grained Reuse" ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 28, no. 2, March 2006, pp. 331-388
[DynCodAnWeb]	http://en.wikipedia.org/wiki/Dynamic_program_analysis
[G/07]	Alejandro González. *"Trait refactorings: mejorando la utilidad de las herramientas de refactoring"*. Universidad de Buenos Aires 2007.
[J/77]	Stephen Johnson. "*Lint, a C program checker"*. Computer Science Technical Report 65, Bell Laboratories, December 1977.
[LintWeb]	http://st-www.cs.uiuc.edu/users/brant/Refactory/LintChecks.html
[MIWeb]	http://en.wikipedia.org/wiki/Multiple_inheritance. Multiple inheritance definition at Wikipedia.com
[MixWeb]	http://c2.com/cgi/wiki?MixIn. Mixins definition at Cunningham & Cunningham, Inc.
[NDRS/05]	Oscar Nierstrasz, Stéphane Ducasse, Stefan Reichhart and Nathanael Schärli. ***"Adding Traits to (Statically Typed) Languages"***. Technical Report, no. IAM-05-006, Institut für Informatik, December 2005, Technical Report, Universität Bern, Switzerland.

[OBWeb]	https://gforge.inria.fr/frs/download.php/27418/Omnibrowser.pdf
[PharoWeb]	http://www.pharo-project.org
[S/05]	Nathanael Schärli. *"Traits — Composing Classes from Behavioral Building Blocks"*, Ph.D. thesis, University of Berne, February 2005.
[SDNB/03]	Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz and Andrew Black, ***"Traits: Composable Units of Behavior,"*** Proceedings of European Conference on Object-Oriented Programming (ECOOP'03), LNCS, vol. 2743, Springer Verlag, July 2003, pp. 248-274.
[StAnWeb]	http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
[StCodAnWeb]	http://en.wikipedia.org/wiki/Static_code_analysis