Preprint Version
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.2160

Talents: An Environment for Dynamically Composing Units of Reuse*

Jorge Ressia, Tudor Gîrba, Oscar Nierstrasz, Fabrizio Perin, Lukas Renggli

SUMMARY

Reuse in object-oriented languages typically focuses on inheritance. Numerous techniques have been developed to provide finer-grained reuse of methods, such as flavors, mixins and traits. These techniques, however, only deal with reuse at the level of classes.
Class-based reuse is inherently static. Increasing use of reflection and meta-programming techniques in real world applications underlines the need for more dynamic approaches. New approaches have shifted to object-specific reuse. However, these techniques fail to provide a complete solution to the composition issues arising during reuse.
We propose a new approach that deals with reuse at the object level and that supports behavioral and state composition. We introduce a new abstraction called a *talent* which models features that are shared between objects of different class hierarchies. Talents provide a composition mechanism that is as flexible as that of traits but which is dynamic.

KEY WORDS: Reflection, Traits, Mixins, Object-specific behavior, Object adaption, Smalltalk

1 Introduction

Classes in object-oriented languages define the behavior of their instances. Inheritance is the principle mechanism for sharing common features between classes. Single inheritance is not expressive enough to model common features shared by classes in a complex hierarchy. Several forms of multiple inheritance have consequently been proposed [1, 2, 3, 4, 5]. However, multiple inheritance introduces problems that are difficult to resolve [6, 7]. One can argue that these problems arise due to the conflict between the two separate roles of a class, namely that of serving as a factory for instances, as well as serving as a repository for shared behaviour for all instances. As a consequence, finer-grained reuse mechanisms, such as flavors [8] and mixins [9], were introduced to compose classes from various features.

Although mixins succeed in offering a separate mechanism for reuse they must be composed linearly, thus introducing new difficulties in resolving conflicts at composition time. Traits [10, 11] overcome some of these limitations by eliminating the need for linear ordering. Instead dedicated operators are used to resolve conflicts. Nevertheless, both mixins and traits are inherently static, since they can only be used to define new classes, not to adapt existing objects.

Ruby [12] relaxes this limitation by allowing mixins to be applied to individual objects. Object-specific mixins however still suffer from the same compositional limitations of class-based mixins, since they must still be applied linearly to resolve conflicts.

In this paper we introduce *talents*, object-specific units of reuse that model features an object can acquire at run-time. Like a trait, a talent represents a set of methods that constitute part of the behavior of an object. Unlike traits, talents can be acquired (or lost) dynamically. When a talent is

*In Software: Practice and Experience, 2012. Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.2160

applied to an object, no other instance of the object's class are affected. Talents may be composed of other talents, however, as with traits, the composition order is irrelevant. Conflicts must be explicitly resolved.

Like traits, talents can be flattened, either by incorporating the talent into an existing class, or by introducing a new class with the new methods. However, flattening is purely static and results in the loss of the dynamic description of the talent on the object. Flattening is not mandatory, on the contrary, it is just a convenience feature which shows how traits are a subset of talents.

The contributions of this paper are:

- We identify static problems associated with multiple inheritance, mixins and traits.
- We introduce *talents*, an object-specific behavior composition model that removes the limitations of static approaches.
- We describe *stateful talents*, an object-specific state composition mechanism.
- We describe a Smalltalk implementation of our approach.

Outline. In Section 2 we motivate the problem. Section 3 explains the talent approach, its composition operations and a solution to the motivating problem. In Section 4 we present the internal implementation of our solution in the context of Smalltalk. In Section 5 we discuss related work. Section 6 discusses about features of talents such as scoping and flattening. In Section 7 we present examples to illustrate the various uses of talents. Section 8 presents a dedicated user interface for managing and defining talents. Section 9 summarizes the paper and discusses future work.

2 Motivating Examples

In this section we analyze two examples that demonstrate the need for a dynamic reuse mechanism.

Moose is a platform for software and data analysis that provides facilities to model, query, visualize and interact with data [13, 14]. Moose represents source code in a model described by FAMIX, a language-independent meta-model [15]. The model of a given software system consists of entities representing various software artifacts such as methods (through instances of `FAMIXMethod`) or classes (through instances of `FAMIXClass`). Each type of entity offers a set of dedicated analysis actions. For example, a `FAMIXClass` offers the possibility of visualizing its internal structure, and a `FAMIXMethod` offers the ability to browse its source code. Selecting the needed features for an entity is awkward within the constraints of a fixed class hierarchy.

In a second example, we consider various kinds of streams, whose features can be combined at run time, rather than requiring that a class be created for every conceivable combination of features.

2.1 Moose Meta-model

Moose can model applications written in different programming languages, including Smalltalk, Java, and C++. These models are built with the language independent FAMIX meta-model. However, each language has its own particularities which are introduced as methods in the different entities of the meta-model. There are different extensions which model these particularities for each language. For example, the Java extension adds the method `isSessionBean` to the `FAMIXClass`, while the Smalltalk extension adds the method `isExtended`. Smalltalk however does not support namespaces, and Java does not support class extensions. Additionally, to identify test classes Java and Smalltalk require different implementations of the method `isTestClass` in `FAMIXClass`.

Another problem with the extensions for particular languages is that the user has to deal with classes that have far more methods than the model instances actually support. Dealing with unused code reduces developer productivity and it is error prone.

A possible solution is to create subclasses for each supported language. However, there are some situations in which the model requires a combination of extensions: Moose JEE [16, 17] — a Moose

extension to analyze Java Enterprise Applications (JEAs) — requires a combination of Java and Enterprise Application specific extensions. This leads to an impractical explosion of the number of subclasses. Moreover, possible combinations are hard to predict in advance.

Multiple inheritance can be used to compose the different behaviors a particular Moose entity requires. However, this approach has been demonstrated to suffer from the "diamond problem" [18, 9] (also known as "fork-join inheritance" [19]), which occurs when a class inherits from the same base class via multiple paths. When common features are defined in different paths then conflicts arise. This problem makes it difficult to handle the situation where two languages to be analyzed require the addition of a method of the same name.

Mixins address the composition problem by applying a composition order, this however might lead to fragile code and subtle bugs. Traits offer a solution that is neutral to composition order, but traits neither solve the problem of the explosion in the number of classes to be defined, nor do they address the problem of dynamically selecting the behavior. Traits are composed statically into classes before instances can benefit from them.

We need a mechanism capable of dynamically composing various behaviors for different Moose entities. We should be able to add, remove, and change methods. This new Moose entity definition should not interfere with the behavior of other entities in other models used concurrently. We would like to be able to have coexisting models of different languages, formed by Moose entities with specialized behavior.

2.2 Streams

Streams are used to iterate over sequences of elements such as sequenced collections, files, and network streams. Streams offer a better way than collections to incrementally read and write a sequence of elements.

Streams may be either readable, writeable or both readable and writeable. They can also be binary or character-based. Furthermore, streams can have different backends, such as memory streams, socket streams, database streams, or file streams.

The potential combination of all these various types of streams leads to an explosion in the number of classes.

Similar solutions to the Moose meta-model problem can be provided, however they present the same shortcomings. Multiple inheritance can be used to compose the different behaviors of a particular stream. However, the diamond problem again makes it difficult to handle the situation where two streams want to add a method of the same name. Mixins address the composition problem by applying a composition order, this however might lead to fragile code and subtle bugs. Traits offer a solution that is neutral to composition order, but traits neither solve the problem of the explosion in the number of classes to be defined, nor do they address the problem of dynamically selecting the behavior. Traits are composed statically into classes before instances can benefit from them.

We need a mechanism capable of dynamically composing the right combination of streams required for each particular occasion. The key objective is to avoid an exponential increase in the number of classes which need to provide all the different combinations.

3 Talents in a Nutshell

In this section we present our approach. We propose composable units of behavior for objects, called *talents*. These abstractions solve the issues present in other approaches.

The Talents* system and the examples presented in this paper are implemented in Pharo Smalltalk†, an open-source Smalltalk [20] implementation. Readers unfamiliar with the syntax of Smalltalk might want to read the code examples aloud and interpret them as normal sentences: An

*http://scg.unibe.ch/research/bifrost/talents
†http://www.pharo-project.org/

invocation of a method named `method:with:`, using two arguments looks like: `receiver method: arg1 with: arg2`. Other syntactic elements of Smalltalk are: the dot to separate statements: `statement1. statement2`; square brackets to denote code blocks or anonymous functions: `[ statements ]`; and single quotes to delimit strings: `'a string'`. The caret ^ returns the result of the following expression.

3.1 Defining Talents

A talent specifies a set of methods which may be added to, or removed from, the behavior of an object. Although the methods of a talent may directly access the state of an object, it is recommended to use accessor methods instead.

We will illustrate the use of talents with the Moose extension example introduced in the previous section.

A talent is an object that specifies methods that can be added to an existing object. A talent can be assigned to any object in the system to add or remove behavior.

```
1 aTalent := Talent new.
2 aTalent
3    defineMethod: #isTestClass
4    do: '^ self inheritsFromClassNamed: #TestCase'.
5 aClass := FAMIXClass new.
6 aClass acquire: aTalent.
```

We can observe that first a generic talent is instantiated and then a method is defined. The method `isTestClass` is used to test if a class inherits from `TestCase`. In lines 5–6 we can see that a FAMIX class is instantiated acquiring the previous talent. When the method `acquire:` is called, the object — in this case the FAMIX class — is adapted. Only this `FAMIXClass` instance is affected, no other instance is modified by the talent. No adaptation will be triggered if an object tries to acquire the same talent several times.

Talents can also remove methods from the object that acquires them.

```
1 aTalent := Talent new.
2 aTalent excludeMethod: #duplicatingClasses.
3 aClass := FAMIXClass new.
4 aClass acquire: aTalent.
```

In this case the existing method `duplicatingClasses` is removed from this particular class instance. Sending this message will now trigger the standard `doesNotUnderstand:` error of Smalltalk.

3.2 Composing Objects from Talents

Talent composition order is irrelevant, so conflicting talent methods must be explicitly disambiguated. Contrary to traits, the talent definition of a method takes precedence if the object acquiring the talent already has the same method. This is because we want behavior that is specific to objects, and as such the object-specific behavior must take precedence over the statically defined one. Once an object is bound to a talent then it is clear that this object needs to specialize its behavior. This precedence can be overridden if it is explicitly stated during the composition by removing the definition of the methods from the talent.

In the next example we will compose a group with two talents. One expresses the fact that a Java class is in a namespace, the other that a JEE class is a test class.

```
1  javaClassTalent := Talent new.
2  javaClassTalent
3     defineMethod: #namespace
4     do: '^ self container'.
5  jeeClassTalent := Talent new.
6  jeeClassTalent
7     defineMethod: #isTestClass
8     do: '^ self methods anySatisfy: [ :each | each isTestMethod ]'.
9  aClass := FAMIXClass new.
10 aClass acquire: javaClassTalent , jeeClassTalent.
```

In line 10 we can observe that the composition of talents is achieved by sending the comma message (,). The composed talents will allow the FAMIX class instance to dynamically reuse the behavior expressed in both talents.

3.3 Conflict Resolution

A conflict arises if and only if two talents being composed provide different implementations for the same method. Conflicting talents cannot be composed, so the conflict has to be resolved to enable the composition.

To gain access to the different implementations of conflicting methods, talents support an alias operation. An alias makes a conflicting talent method available by using another name. Traits aliasing is different to renaming, both the alias and the original method are accessible. As stated by Ducasse *et al.* [11]: "Aliases allow the programmer to make a trait method available under another name, and are very useful if the original name is excluded by a conict." Due to the dynamic nature of talents aliasing is implmented as a renaming. An alias in talents is a composition of a rename followed by an exclusion of the original method.

Formally, defining an alias `y` for a method `x` in the talent `T` establishes an alternative name `y`. In particular, all references to the original name `x` in the used talent `T` are changed (*i.e.,*, they refer to the new name y).

The main disadvantage of renaming is that it violates the traits flattening property. Since talents are targeted towards dynamic state and behavior compositions and adaptation the flattening behavior is of no advantage.

Talent composition also supports exclusion, which allows one to avoid a conflict before it occurs. The composition clause allows the user to exclude methods from a talent when it is composed. This suppresses these methods and allows the composite entity to acquire the otherwise conflicting implementation provided by another talent.

We would like models originating from JEE applications to support both Java and JEE extensions. Composing these two talents however generates a conflict for the methods `isTestClass` for a FAMIX class entity. The next example produces a conflict on line 10 since both talents define a different implementation of the `isTestClass` method.

```
1  javaClassTalent := Talent new.
2  javaClassTalent
3     defineMethod: #isTestClass
4     do: '^ self methods anySatisfy: [ :m | m isAnnotatedWith: #Test ]'.
5  jeeClassTalent := Talent new.
6  jeeClassTalent
7     defineMethod: #isTestClass
8     do: '^ self inheritsFromClassNamed: #TestCase'.
9  aClass := FAMIXClass new.
10 aClass acquire: javaClassTalent , jeeClassTalent.
```

There are different ways to resolve this situation. The first is to define aliases, like in traits, to avoid the name collision. Aliases are used to avoid collisions, rather than to resolve collisions by, say, using a priority mechanism:

```
10 aClass acquire: javaClassTalent , (jeeClassTalent @ {#isJeeTestClass -> #isTestClass}).
```

When the talent is acquired the method `isJeeTestClass` is installed instead of `isTestClass`, thus avoiding the conflict. Any other method or another talent can then make use of this aliasing.

Another option is to remove those methods that do not make sense for the specific object being adapted.

```
10 aClass acquire: javaClassTalent , (jeeClassTalent - #isTestClass).
```

By removing the definition of the JEE class talent the Java class talent method is correctly composed.

Each FAMIX extension can be defined as a set of talents, each for a single entity, *i.e.,* class, method, annotation, *etc.* For example, we have the Java class talent which models the methods

required by the Java extension to FAMIX class entity. We also have a Smalltalk class talent as well as a JEE talent that model further extensions.

3.4 Stateful Talents

In the original traits model, state can only be accessed within stateless traits by accessors, which become required methods of the trait. As demonstrated by Bergel *et al.* [21], traits are artificially incomplete since classes that use such traits may contain significant amounts of boilerplate glue code. Talents also provide a mechanism for dynamically defining state which is similar to its static counterpart, stateful traits.

```
1 aTalent := Talent new.
2 aTalent defineState: #testClass.
3 aClass := FAMIXClass new.
4 aClass acquire: aTalent.
```

We can observe that first a generic talent is instantiated and then a state called `testClass` is defined. This instance variable is a boolean attribute/field used to test if a class is a test case. In lines 3–4 we can see that a FAMIX class is instantiated acquiring the previous talent. As with behavioral talents when the method `acquire:` is called, the object — in this case the FAMIX class — is adapted. Only this `FAMIXClass` instance is affected, no other instance is modified by the talent. No adaptation will be triggered if an object tries to acquire the same talent several times.

Since this state is introduced on a live object, we also provide a mechanism for managing the initialization. When no default value is provided then the new talent-defined state is set to `nil`, the usual default value for uninitialized attributes in regular Smalltalk code. The developer can use the state definition with default behavior to control state initialization values.

```
1 aTalent := Talent new.
2 aTalent
3    defineState: #testClass
4    defaultValue: true.
5 aClass := FAMIXClass new.
6 aClass acquire: aTalent.
```

The method `defineState:defaultValue:` adds a state definition to a talent which when acquired by an object will have a default value. In the example the state `testClass` has the value `true` by default. To avoid sharing the default values between objects the message `defineState: aSymbol providedBy: aBlock` allows the user to provide a block which will dynamically defines the value of the state.

```
1 aTalent := Talent new.
2 aTalent
3    defineState: #testClass
4    providedBy: [ :class | class name includesSubString: 'Test' ].
5 aClass := FAMIXClass new.
6 aClass acquire: aTalent.
```

In this case the newly defined state is initialized lazily based on the object. In the example the state `testClass` is lazily initialized with `true` if the class-name includes the sub-string `'Test'`.

Methods defined afterwards for the talent can refer to the newly created state without the need for accessors.

```
1 aTalent := Talent new.
2 aTalent defineState: #testClass.
3 aTalent
4    defineMethod: #isTestClass
5    do: '^ testClass'.
6 aClass := FAMIXClass new.
7 aClass acquire: aTalent.
```

In lines 3–5 we can see the definition of the method `isTestClass` which returns the boolean value `testClass` state. No definition of accessors is required and talents can define methods directly accessing the state.

The user can define state accessors by various other helper methods:

```
aTalent := Talent new.
aTalent defineStateWithAccessors: #testClass.
aClass := FAMIXClass new.
aClass acquire: aTalent.
```

By using `defineStateWithAccessors:` the talent definition also adds the two accessors for reading and writing on the `testClass` state. The user can also use `defineStateWithReadAccessor:` and `defineStateWithWriteAccessor:` which are self-explanatory.

4 Implementation

In this section we describe how talents are implemented.

4.1 Bifröst

Talents are built on top of the Bifröst reflection framework [22]. Bifröst offers fine-grained unanticipated dynamic structural and behavioral reflection through meta-objects. Instead of providing reflective capabilities as an external mechanism we integrate them deeply into the environment. Explicit meta-objects allow us to provide a range of reflective features and thereby evolve both application models and the host language at run-time. Meta-objects provide a sound basis for building different coexisting *meta-level architectures* by bringing traditional object-oriented techniques to the meta-level. Each talent is modeled with a structural meta-object.

In recent years researchers have worked on applying traditional object-oriented techniques to the meta-level while attempting to solve various practical problems motivated by applications [23]. These approaches, however, offer specialized solutions arising from the perspective of particular use cases.

The Bifröst model solves the main problems of previous approaches while providing the main reflection requirements.

Partial Reflection. Bifröst allows meta-objects to be bound to any object in the system thus reflecting selected parts of an application.

Selective Reification. When and where a particular reification should be reified is managed by the different meta-objects.

Unanticipated Changes. At any point in time a meta-object can be bound to any object thus supporting unanticipated changes.

Meta-level Composition. Composable meta-objects provide the mean for bringing together different adaptations.

Runtime Integration. Bifröst's reflective model lives entirely in the language model, so there is no VM modification or low level adaptation required.

In Talents, we particularly use partially reflection on specific objects, talents are applied without anticipation, and they are composed dynamically.

Bifröst's adaptation mechanism is built on top of lower-level meta-objects. In the Bifröst Smalltalk implementation we bind meta-objects to abstract syntax tree (AST) nodes. A meta-object can be associated to a single AST node or to multiple ones. The next time the method is compiled the system automatically generates new bytecodes that take the meta-object into account. This behavior allows Bifröst to adapt the predefined behavior of objects. AST meta-objects can reify AST-related information depending on the AST node. For example, a message send node can reify the sender, the receiver and the arguments at runtime. The meta-level behavior specified in the meta-object can be executed before, after or instead of the AST node the meta-object is adapting.

Bifröst exploits Pharo's *reflective method* abstraction. A reflective method knows the AST of the method it represents. In Pharo classes are first-class objects that are accessible and changeable at run time. Classes have an internal collaborator which holds an instance of `MethodDictionary`, a special subclass of Dictionary. All methods of a class are stored in its method dictionary. The VM directly accesses class objects and method dictionaries when evaluating message sends. Normally, only instances of `CompiledMethod` are stored in the method dictionary of a class but Pharo allows us to replace it with any other object that obeys the right protocol. When such an object is used in place of a regular compiled method, the VM sends it the message `run:with:in:`, encoding the message, its arguments and the recipient. When a reflective method receives this message it processes the adaptations specified by the meta-object on the AST and generates a new compiled method that is eventually executed. If no adaptation is present the reflective method caches the compiled method to improve performance. In the current Talents implementation the user does not manage the lower level details of the adaptations. The talent interface allows the user to abstract from the complexities of the lower level.

4.2 Talents

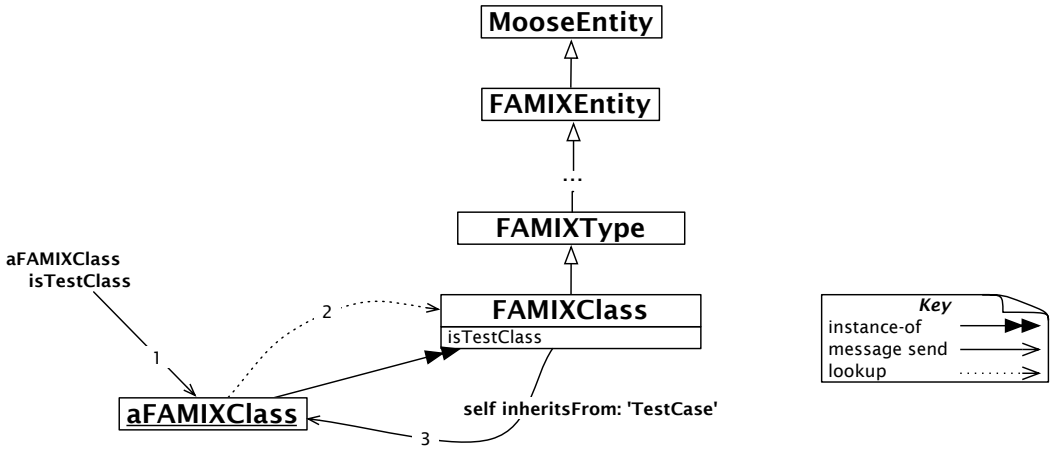

Figure 1: Default message send and method look up resolution.

Figure 1 shows the normal message send of `isTestClass` to an instance of `FAMIXClass`. The method lookup starts on the class finding the definition of the method and then executing it for the message receiver.

However, if we would like to factor the `FAMIXClass` JEE behavior out we can define a talent that models this. Each talent is modeled with a structural meta-object. A structural meta-object abstraction provides the means to define meta-objects like classes and prototypes. New structural abstractions can be defined to fulfill some specific requirement. These meta-object responsibilities are: adding and removing methods, and adding and removing state to an object. A composed meta-object is used to model composed talents. The specific behavior for defining and removing methods is delegated to the addition and removal of behavior in the structural meta-object.

In Figure 2 we can observe the object diagram for a FAMIX class which has acquired a talent that models JEE behavior. The method lookup starts in the class of the receiver. Originally, the `FAMIXClass` class did not define a method `isTestClass`, however, the application of the talent defined this method. This method is responsible for delegating the execution of the message to the receiver's talent. If the object does not have a talent, the normal method lookup is executed, thus talents do not affect other instances' behavior of the class. In this case, `aFAMIXClass` has a talent that defines the method `isTestClass`, which is executed for the message receiver.

Bifröst's structural meta-objects provide features for adding state to a single object and removing it. Talents can provide something that traits cannot, namely state. Moreover, talents can provide

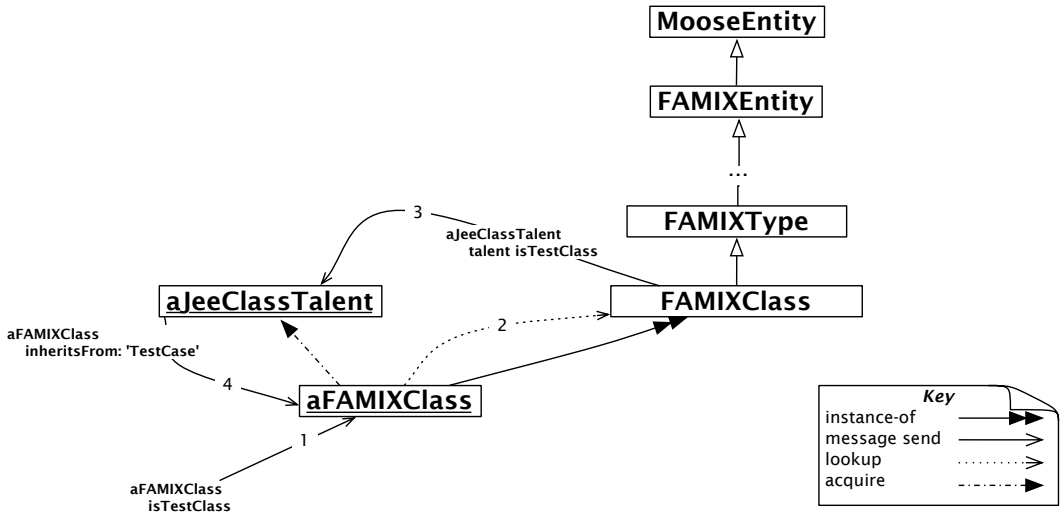

Figure 2: Talent modeling the Moose FAMIX class behavior for the method `isTestClass`.

operators for composing state adaptations. This composition is not present in object-specific techniques like mixins and Newspeak [24] modules.

5 Related Work

In this section we compare talents to other approaches to share behavior.

Mixins

Flavors [8] was the first attempt to address the problem of reuse across a class hierarchy. Flavors are small, incomplete implementations of classes, that can be "mixed in" at arbitrary places in the class hierarchy. More sophisticated notions of mixins were subsequently developed by Bracha and Cook [9], Mens and van Limberghen [25], Flatt, Krishnamurthi and Felleisen [26], and Ancona, Lagorio and Zucca [27].

Mixins present drawbacks when dealing with composition. Mixins use single inheritance for composing features and extending classes. Inheritance requires that mixins be composed linearly; this severely restricts one's ability to specify the glue code that is necessary to adapt the mixins so that they fit together [10]. However, although this inheritance operator is well-suited for deriving new classes from existing ones, it is not appropriate for composing reusable building blocks.

Bracha developed Jigsaw [28], a modularity framework which defines module composition operators merge, override, copy as and restrict. These operators inspired the sum, override, alias and exclusion operators on traits. Jigsaw models a complete framework for module manipulation providing namespaces, declared types and requirements, full renaming, and semantically meaningful nesting.

Ruby [12] introduced mixins as a building block of reusability, called modules. Moreover, modules can be applied to specific objects without modifying other instances of the class. However, object-specific modules suffer from the same composition limitation as modules applied to classes: they have to be applied linearly. Aliasing of methods is possible for avoiding name collisions, as well as removing method in the target object. However, objects or classes methods cannot be removed if they are not already implemented. This follows the concept of linearization of mixins. Talents can be applied without an order. Moreover, a talent composition delivers a new talent that can be reused and applied to other objects. Filters in Ruby provide a mechanism for composing behavior

into preexisting methods. However, they do not provide support for defining how modules defined methods should be composed for a single object.

CLOS

CLOS [29] is an object-oriented extension of Lisp. Multiple inheritance in CLOS [30, 31] imposes a linear order on the superclasses. This linearization often leads to unexpected behavior because it is not always clear how a complex multiple inheritance hierarchy should be linearized [32]. CLOS also provides a mechanism for modifying the behavior of specific instances by changing the class of an instance using the generic function `change-class`. However, these modifications do not provide any composition mechanisms, rendering this technique dependent on custom code provided by the user.

Traits

Traits [10, 11] overcome the limitations of previous approaches. A trait is a set of methods that can be reused by different classes. The main advantage of traits is that their composition does not depend on a linear ordering. Traits are composed using a set of operators — symmetric combination, exclusion, and aliasing — allowing a fair amount of composition flexibility. Traits are purely static since their semantics specify that traits can always be "flattened" to an equivalent class hierarchy without traits, but possibly with duplicated code. As a consequence traits can neither be added nor removed at run-time. Moreover, traits were not conceived to model object-specific behavior reuse.

Smith and Drossopoulou [33] proposed a mechanism for applying traits at runtime in the context of Java. However, only pre-defined behavior defined in a trait can be added at runtime. It is not possible to define and add new behavior at runtime.

Bettini *et al.* [34] proposed a mechanism for flexible dynamic trait replacement where traits can be applied at runtime. However, this technique can only change existing behavior, not add new behavior.

Object Extensions

Self [35] is a prototype-based language which follows the concepts introduced by Lieberman [36]. In Self there is no notion of class; each object conceptually defines its own format, methods, and inheritance relations. Objects are derived from other objects by cloning and modification. Objects can have one or more prototypes, and any object can be the prototype of any other object. If the method for a message send is not found in the receiving object then it is delegated to the parent of that object. In addition, Self also has the notion of trait objects that serve as repositories for sharing behavior and state among multiple objects. One or more trait objects can be dynamically selected as the parent(s) of any object. Selector lookups unresolved in the child are passed to the parents; it is an error for a selector to be found in more than one parent. Self traits do not provide a mechanism to fine tune the method composition. Let us assume that two objects are dynamically defined as parents of an object. If the both parents object define the same method there is not a simple way of managing the conflict.

Object extension [37, 38] provides a mechanism for self-inflicted object changes. Since there is no template serving as the object's class, only the object's methods can access the newly introduced method or data members. Ghelli *et al.* [37] suggested a calculus in which conflicting changes cannot occur, by letting the same object assume different roles in different contexts.

Drossopoulou proposed Fickle [39], a language for dynamic object re-classification. Re-classification changes at run-time the class membership of an object while retaining its identity. This approach proposes language features for object re-classification to extend an imperative, typed, class-based, object-oriented language. Even though objects may be re-classified across classes with different members, they will never attempt to access non-existing members.

Cohen and Gil introduced the concept of object evolution [40]. This approach proposes three variants of evolution, relying on inheritance, mixins and shakeins [41]. The authors

introduce the notion of evolvers, a mechanism for maintaining class invariants in the course of reclassification [39]. This approach is oriented towards dynamic reuse in languages with types. Shakeins provide a type-free abstraction, however, there are no composition operators to aid the developer in solving more complex scenarios.

Bracha *et al.* [24] proposed a new implementation of nested classes for managing modularity in Newspeak. Newspeak is class-based language with virtual classes. Class references are dynamically determined at runtime; all names in Newspeak are method invocations thus all classes are virtual. Nested classes were first introduced in Beta [42]. In Newspeak Classes declarations can be nested to an arbitrarily depth. Since all references to names are treated as method invocations any object member declaration can be overridden. The references in an object to nested classes are going to be solved when these classes are late bound to the classes definition in the active module the object it is in. Talents model a similar abstraction to modules, for dynamically composing the behavior of objects. However, Newspeak modules do not provide composition operators similar to talents. Composed talents can remove, alias, or override method definitions. Removing method definitions is not a feature provided by Newspeak modules. In Newspeak composition would be done in the module or in the nested classes explicitly.

Context-oriented Programming

Context-oriented programming (COP) was introduced by Costanza *et al.* [43]. The behavior of an object is split into layers that define the object's subjective behavior. Layers can be activated and deactivated to represent the actual contextual state. When a message is sent, the active context determines the behavior of the object receiving the message.

Subjective Programming

Subjective behavior is essential for applications that must adapt their behavior to changing circumstances. Many different solutions have been proposed in the past, based, for example, on perspectives [44], roles [45], contextual layers [43], and force trees [46]. Depending on the active context, an object might answer a message differently or provide a modified interface to its users. These approaches mainly concentrate on dynamically modifying an object's behavior, however, there is no support for behavior reuse between object as it exists in traits or mixins.

Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [47] modularizes cross cutting concerns. Join points define all locations in a program that can possibly trigger the execution of additional cross-cutting code (advice). Pointcuts define at run-time if an advice is executed. Both aspects and talents can add new methods to existing classes. Most implementations of AOP such as AspectJ [48] support weaving code at more fine-grained join points such as field accesses, which is not supported by talents. Although AOP is used to introduce changes into software systems, the focus is on cross-cutting concerns, rather than on reflecting on the system.

Aspects are concerns that cannot be cleanly encapsulated in a generalized abstraction (i.e., object, method, mixin). This means that in contrast to talents, aspects are neither designed nor used to build dynamic abstraction and components from scratch, but rather to alter the performance or semantics of the components in systemic ways.

6 Discussion

In this section we discuss other benefits that talents bring to a programming language.

6.1 Scoping

Scoping talents dynamically is of key importance because it allows us to reflect in which context the added features should be active and also to control the extent of the system that is modified. An object might require to have certain features in one context while having others features in a different context. Let us analyze an example to understand the motivation for scoping talents.

A bank financial system is divided in two main layers: the domain and the persistency layer. The domain layer models the financial system requirements and features. The persistency layer deals with the requirements of persisting the domain abstraction in a relational database. When testing the domain behavior of this application we do not want to trigger database-related behavior. Normally, this is solved through mocking or dependency injection [49]. However, these solutions are not simple to implement in large and legacy systems which are not fully understood, and where any change can bring undesired side effects. Scoped talents can solve this situation by defining a scope around the test cases. When the tests are executed the database access objects are modified by a talent which mocks the execution of database related actions. In a highly-available system which cannot be stopped, like a financial trading operation, scoped talents can help in actions like: auditing for the central financial authority, introducing lazy persistency for updating the database, logging. This is similar to the idea of modules in Newspeak.

COP solutions can provide an implementation solution to bringing talents to other languages. Lincke *et al.* [50] presented a mechanism for composing layers in ContextJS, a JavaScript COP implementation. Talents offer a form of COP based on object-specific meta-objects rather than layers.

6.2 Flattening

Flattening is the technique that folds into a class all the behavior that has been added to an object. There are two types of flattening in talents:

Flattening on the original class. Once an object has been composed with multiple talents it has a particular behavior. The developer can analyze this added behavior and from a modeling point of view realize that all instances of the object's class should have these changes. This kind of flattening applies the talent composition to the object's class.

Flattening on a new class. On the other hand the developer might realize that the new responsibilities of the object is relevant enough to be modeled with a separate abstraction. Thus a new class has to be created cloning the composed object behavior. This new class will inherit from the previous object class. Deleted methods will be added with a `shouldNotCallMethod` exception to avoid inheriting the implementation.

6.3 Performance

Using meta-level programming techniques on a runtime system can have a significant performance impact. Evidence from practical applications of Bifröst, like Subjectopia[51], a subjective system, MetaSpy[52], a domain-specific profiler, and Object-centric debugging [53], showed that the performance impact does not affect the usability of these tools.

We have performed a micro-benchmark to assess the maximal performance impact of our Smalltalk prototype of object-centric debugging. We assume that the behavior required to fulfill the profiling requirements is constant to any instrumentation strategy.

All benchmarks were performed on an Apple MacBook Pro, 2.8 GHz Intel Core i7 in Pharo 1.1.1 with the jitted Cog VM.

Consider a benchmark in which a test method is invoked one million times from within a loop. We measure the execution time of the benchmark with Bifröst reifying all 10^6 method activations of the test method. This shows that in the reflective case the code runs about 35 times slower than in the non-reflective case. However, for a real-world application with only few reifications the performance impact is significantly lower. Bifröst's meta-objects provide a way of adapting selected objects thus

allowing reflection to be applied within a fine-grained scope only. This provides a natural way of controlling the performance impact of reflective changes.

Let us consider the Mondrian profiling problem presented by Bergel *et al.* in MetaSpy *et al.* [52]. The main source of performance degradation is from the execution of the method `displayOn:`, *i.e.,* whenever a node is redisplayed. We developed a benchmark where the user interaction is simulated to prevent human interaction from polluting the measurements. In this benchmark we redraw one thousand times the nodes in the Mondrian visualization. This implies that the method `displayOn:` is called extensively. The results showed that the profiler-oriented instrumentation produces on average a 20% performance impact. The user of this Mondrian visualization can hardly detect the delay in the drawing process. Note that our implementation has not been aggressively optimized.

Another important detail of our implementation is that instrumentations are removed once they have interrupted the execution. The impact on performance is consequently temporal and local to specific objects of the application.

We believe that the simplicity of Bifröst interfaces and its capacity to be loaded on a running system justify this performance penalty. However, a specially modified VM can reduce this penalty while impacting negatively on the tool portability.

6.4 Talents in a statically typed language

A highly dynamic construct such as talents is possible in a statically typed language, even if the language implementation ensures that the type interface remains consistent. This can only be achieved by disallowing changes to the signatures of existing methods. Talents can safely replace existing methods as long as they do not alter their signatures.

In a statically typed language like Java we could declare talents with the help of a marker interface `Talent<T>` [54], where `T` is generic type variable specifying the interface of the talent. These interfaces can then be modeled with talents. Particular combinations of talents can deliver different combinations of an object's interfaces.

```
interface Talent<T> implements T {
   // marker interface for a talent with the interface T
}
```

Classes that want to support a specific talent need to implement the marker interface `Talent` that they parametrize with the interface of the talent `T`. This forces the class to provide a default implementation of all the methods in the interface of the talent. The example with `FAMIXClass` in Section 2.1 would look in Java as follows:

```
interface TestTalent {
   boolean isTestClass();
}

class FAMIXClass implements Talent<TestTalent> {
   boolean isTestTalent() {
      return false;
   }
   ...
}
```

Finally, a generic static helper method must be provided to let objects acquire talents:

```
<O extends Talent<T>, T> void acquire(O object, T talent)
```

The outlined approach would enable talents in Java without weakening the existing type system. We imagine that the talents themselves could be implemented using bytecode rewriting of the methods in classes that implement the marker interface and the state pattern outlined in Section 7.3.

As we demonstrated in this section, talents are possible even in the context of a statically typed language. The implementation however will be limited to the predeclared talent interfaces only.

6.5 Traits on Talents

Conceptually, talents are a generalization of traits. Traits can only be applied to a specific set of objects, classes. Talents can be applied to any object in the system, including classes.

Traits can be implemented on top of the talent infrastructure by having a talent `TraitTalent` with a modified method `basicNew`. This talent is applied to the class in which we would like to have traits. The modified `basicNew` method has the extra behavior of applying the set of composed traits for the given class. This set of composed traits can be contained in an added state to the class defined by the `TraitTalent`. Each trait is defined as a talent and added to the modified class. A trait can also be defined as a wrapper on a talent which adds traits related method.

7 Examples

In this section we present a number of example applications of talents. These examples are selected to exercise the various facets of the talents mechanism, and as such, act as validation of the expressiveness of our approach.

7.1 Mocking

Let us assume that we need to test a class which models a solvency analysis of the assets of a financial institution customer. The method we need to test is `SolvencyAnalysis>>isSolvent: aCustomer`. This method delegates to `SolvencyAnalysis>>assetsOf: aCustomer` which executes a complex calculation of the various assets and portfolios of the customer. We are only interested in isolating the behavior of `isSolvent:`, not in the complexities of `assetsOf:`

When testing such a use case we need to modify the assets of a particular customer by increasing or decreasing the financial instruments deposits. This implies that we need to interact and create objects that are unessential in relation to the objective of the test case. Introducing more objects in a test case increases the chances of making the test fail for other reasons than the test objective. Talents provide a mechanism to modify the behavior of particular objects to modify this interaction, providing an object-specific mocking mechanism.

Let us analyze a talent solution to this use case:

```
1 SolvencyAnalysisTest>>testIsSolvent
2    | aCustomer anAnalysis |
3    aCustomer := Customer named: 'test'.
4    anAnalysis := SolvencyAnalysis new.
5    anAnalysis method: #assetsOf: shouldReturn: 1.
6    self assert: (anAnalysis isSolvent: aCustomer).
7    anAnalysis method: #assetsOf: shouldReturn: -1.
8    self deny: (anAnalysis isSolvent: aCustomer).
```

We added the method `method:shouldReturn:` to the class `Object` which creates a talent with a method named as the first argument and with the body provided by the second argument. In line 5 and 7 you can see the use of this behavior. If the method `assetsOf:` return a positive amount then the customer is solvent otherwise not.

Talents can ease the testing of monolithic legacy applications built in a manner that does not easily support mocking.

7.2 Compiler Internal Abstractions

In traditional compiler design, the compilation of source code is a multi-step process: lexical analysis (scanning) is followed by syntactic analysis (parsing), which is followed by semantic analysis. This is followed by code generation, which itself may be split into multiple optimization and generation steps. Traditionally each of these steps has its own model to work on model of the code, *i.e.,* the lexical analysis uses tokens, the syntactic analysis uses an abstract syntax tree, the semantic analysis uses an intermediate representation, and so on.

The problem with this approach is that it brings a significant overhead of performance and memory. At each step a plethora of new nodes need to be instantiated, as each step accumulates new state and requires different behavior. However, old state of previous steps cannot be thrown away. At any time in the compilation chain the compiler needs to be able to navigate back to the state of the previous steps, for example to pinpoint errors in the source file or to query the lexical structure. Compiler designers can address this requirement in two possible ways, but neither of them is very attractive: Either they copy and accumulate state along the compilation chain, which is error prone and slow; or they keep references to the nodes of the previous step, which can be expensive if the paths have to be navigated often.

With talents we have an elegant solution to this problem. Imagine the scanner reads a variable assignment such as `a := 12`. This results in 3 tokens to be created: `a`, `:=`, and `12`. These tokens not only contain the value they represent, but also know the source file and location in that file. In the syntactic analysis a parser detects that these 3 tokens form an assignment, built from the variable `a`, the assignment operator `:=`, and the value `12`. With talents we let the assignment token acquire an `AssignmentNodeTalent` that has — besides the node specific behaivor — also additional state: an assignment always consists of a variable node and an expression node. In this particular case the token `a` acquires the `VariableNodeTalent` and the value `12` acquires the `ValueNodeTalent`. In the next step, the semantic analysis, the `a` is further refined with the type of variable it represents. In this particular example the compiler could for example let it acquire the `InstanceVariableTalent`.

With each step in the compilation chain new talents are attached. The talents not only introduce new node specific behavior, but also add new state. The added state allows the objects to reorganize themselves in new ways. While the tokens are organized in a sequence of tokens, the syntactic nodes form an abstract syntax tree, and the semantic analysis forms a graph of references.

The approach with talents avoids the drawbacks of existing solutions. The same objects are passed through the complete compilation chain. Each step augments the objects with new state and behavior relevant for this step. Unnecessary copying of state and navigation between long object chains is avoided.

Moreover, the different talents can add their own visitor methods, *e.g.,*`acceptTokenVisitor:`, `acceptNodeVisitor`, `acceptTypeVisitor:`, *etc.*. This does not result in conflicting methods and it allows different visitors to have their own strategy to walk over the trees.

7.3 State Pattern

The state pattern [55] models the different states a domain object might have. When this object needs to do something then it delegates the decision of what to do to its state. A class per object state is created with the required behavior. Sometimes, multiple instances of each state are created and sometimes a singleton pattern is used.

Instead of having a state abstract class and then concrete subclasses for each of the more specific states we could use talents. We will have a single state class and then create as many instances as different states are. We can model each specific state with a different talent that is applied to the state's instances, thus avoiding the creation of multiple state specific classes.

The talents based solution is simpler than a traditional state pattern, since it avoids the additional redirection from the object to its state. The developer simply has an object whose behavior changes, still having the state specific behavior but one indirection is eliminated.

7.4 Streams

Streams can be writable, readable, or both; depending on what talents are added. `WriteStreamTalent` adds the methods for writing to a stream, *i.e.,* `nextPut:` and `nextPutAll:`. `ReadStreamTalent` adds the methods to read from a stream, *i.e.,* `next` and `next:`. Streams can be binary or textual. Talents add the necessary supported methods, *i.e.,* `BinaryReadStreamTalent` adds `nextInt32`; and `TextualWriteStreamTalent` adds the methods `cr` and `space`. Furthermore streams are typically implemented on top of different backends, *i.e.,* collections, sockets, or files. Again we use talents

that provide the necessary primitives for the read and write talents to actually perform the desired tasks.

All possible combinations of read, write, or read-write; binary or textual; memory, sockets, or files are possible. No unnecessary methods outside the requested capabilities are present. Furthermore, the talent composition avoids additional dispatching cost. The resulting streams are as efficient as if all 18 combinations would have been implemented manually.

Traditional stream implementations check in every method if the underlying stream is still opened. With talents we can avoid such cumbersome checks and dynamically acquire a `ClosedStreamTalent` when a stream is closed. This talent either removes all modifying stream methods, or alternatively replaces them with one that throws an exception. This approach not only simplifies the implementation, but it is also more efficient as unnecessary tests are avoided altogether.

7.5 Class Extensions

Class extensions are a means to add required behavior to classes that belong to other packages outside our control. For example, when we load Moose there are several methods that are added or modified, in core classes like `Collection` hierarchy, `Object`, *etc.* This mechanism allows Moose developers to extend the system with Moose specific additions, *e.g.,* utility methods like *asMooseGroup* have been added to the core class `Collection` so to transform any collection in a MooseGroup. The implementation of the extended method `asMooseGorup` is shown in the snippet below:

```
Collection>>asMooseGroup
   ^ MooseGroup withAll: self
```

Class extensions are also largely used within Moose to add functionalities from newly added packages to core packages. For example, when the Moose extension to analyze relational databases [56] is loaded, new methods like `maps` are added to the element `FAMIXNamedEntity` in the Moose core to keep track of the relations between relational elements and source code entities. The implementation of the extended method `maps` is shown in the snippet below:

```
FAMIXNamedEntity>>maps
   <MSEProperty: #maps type: #FAMIXMapping opposite: #mapSource> <multivalued>  <derived>
   <MSEComment: 'Map relationship.'>

   ^self privateState attributeAt: #maps ifAbsentPut: [FMMultivalueLink on: self opposite:
    #mapSource:].
```

A key drawback of this approach is that extensions do not support the definition of state, but only behavior. Moose developers need to implement more complex models since they cannot add state to classes outside Moose packages. Talents can be used to address this issue. The extension mechanism can be improved to provide state facilities by applying talents to all live instances of a particular class when an extension is loaded. When a state extension is defined for a particular class a talent with state definition is used. When the packages with extensions are loaded all instances of the extended classes are gathered and the predefined stateful talents are applied to them. Moreover, some class instances can be left out of the talent adaptation by providing conditions on various criteria. For example, only instances not reachable from core classes should be adapted.

8 User Interface

The talents browser is responsible for organizing, managing and defining talents. This browser is built using Glamour [13, 57], an engine for scripting browsers.

In Figure 1 we can observe an instance of the talents browser for the FAMIX class case study. The browser is vertically divided into two panes, the upper navigation section and the lower source code section.

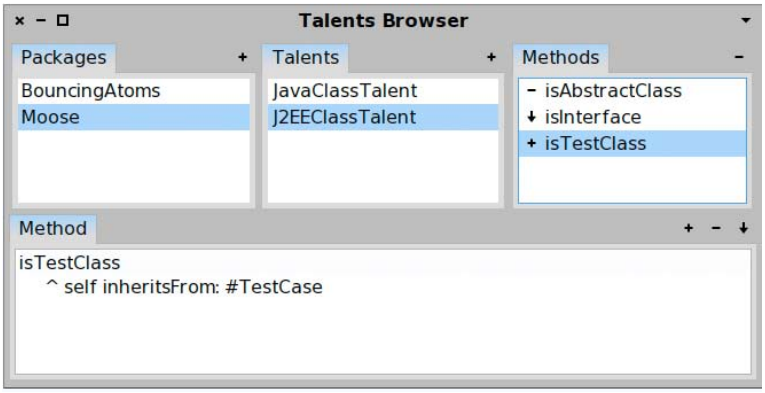

Figure 3: Talents Browser overview.

The navigation section is divided into three panes following traditional Smalltalk browsers. The first pane shows a list of talents packages. Packages group related talents together. In this example we can see two talents packages: `BouncingAtoms` and `Moose`. Once a package is selected the talents pane is populated with the talents belonging to that package. In this example the `Moose` package is composed of two talents: `JavaClassTalent` and `J2EEClassTalent`. When a talent is selected the Methods pane is populated with the talent's method definitions. The icons before the name of the method represent the definition behavior. The — icon is used to signal that this method should be excluded when the talent is applied to an object. The ↓ icon indicates that a method should be replaced with the defined behavior. The + icon represents that the method should be added to the adapted object.

These panes provide contextual menus for removing, renaming and adding packages, talents and methods. The methods panel only provides a remove menu item. The talents panel provides a remove and a rename menu items.

The source code pane displays the source code defined for the selected method in the Methods pane. The three icons in the upper right corner represent the actions possible when saving a method definition. The + icon accepts the source code and adds a method definition to the selected talent. The — icon prompts the user to provide the name of the method which should be excluded by the selected talent. The ↓ icon accepts the source code and a method replacement definition to the selected talent.

Talents defined in the browser are registered to `TalentsRegistry`. When a developer needs to use a particular talent he can access the registry by name.

```
aTalent := TalentsRegistry registry talentNamed: 'J2EEClassTalent'.
```

The talents browser is useful for managing the creation and structure of talents but it does not provide a way to manage the association of talents to objects. To fulfill this we modified the default object inspector, one of the main instruments used during development, to open the talent browser directly on the inspected object. Figure 4 shows the work flow to attach a talent to an object. Once we have opened the system inspector on the object we want to enrich with a talent, we can open the talent browser directly from the contextual menu of the object. As a result the object we were inspecting is passed along to the browser. The talents browser allows the developer to find the right talents and if need be to modify it for fulfilling new requirements. The developer can then select a talent and associate it to the inspected object using the option `Apply Talent` on the contextual menu that open on the talent classes.

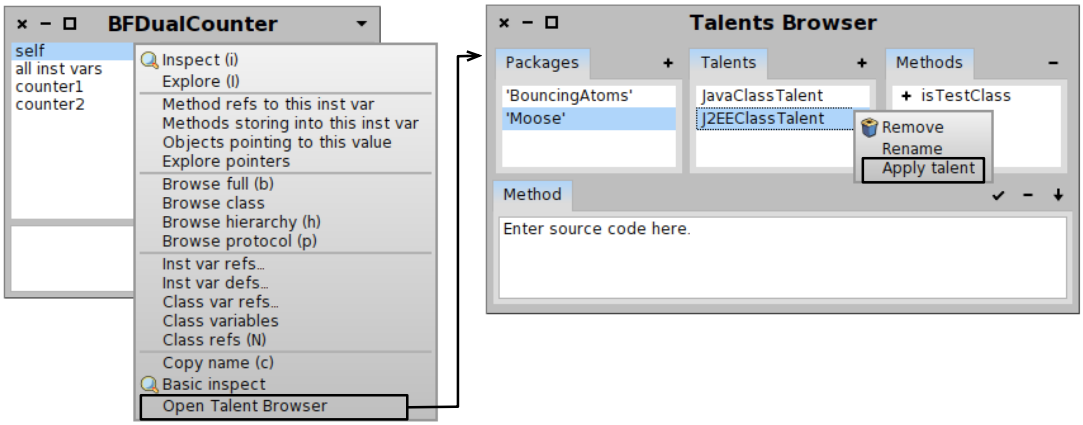

Figure 4: Modified inspector and Talents Browser Insteraction.

9 Conclusion and Future Work

This paper presented talents, a dynamic compositional model for reusing behavior. Talents are composed using a set of operations: composition, exclusion and aliasing. These operations provide a flexible composition mechanism while avoiding the problems found in mixins and traits.

Talents are most useful in composing behavior for different extensions that have to be applied to the same base classes, thus dynamically adapting the behavior of the instances of these classes seems natural to obtaining a different protocol.

Managing talents can currently be complicated since the classic development tools are unaware of them. Our talents user interface solves the problem of managing and defining talents. However, it does not provide features for composing talents nor does it help in visualizing these compositions. We plan on extending the talents user interface to deal with composition requirements.

We plan on providing a more mature implementation of the talents scoping facilities. This technique shows great potential for the requirements of modern applications, such as dynamic adaptation and dependency injection for testing, database accesses, profiling, and so on.

Acknowledgements. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Synchronizing Models and Code" (SNF Project No. 200020-131827, Oct. 2010 – Sept. 2012). We also like to thank Simon Denier for his feedback on earlier drafts of this paper.

References

[1] Borning AH, Ingalls DH. Multiple inheritance in Smalltalk-80. *Proceedings at the National Conference on AI*, Pittsburgh, PA, 1982; 234–237.

[2] Keene SE. *Object-Oriented Programming in Common-Lisp*. Addison Wesley: 75 Arlington Street Suite 300, Boston MA, 02116 USA, 1989.

[3] Meyer B. *Object-Oriented Software Construction*. Second edn., Prentice-Hall, 1997.

[4] Schaffert C, Cooper T, Bullis B, Killian M, Wilpolt C. An Introduction to Trellis/Owl. *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, vol. 21, 1986; 9–16.

[5] Stroustrup B. *The C++ Programming Language*. Addison Wesley: Reading, Mass., 1986.

[6] Dixon R, McKee T, Vaughan M, Schweizer P. A fast method dispatcher for compiled languages with multiple inheritance. *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, vol. 24, 1989; 211–214.

[7] Sweeney PF, Gil JY. Space and time-efficient memory layout for multiple inheritance. *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, ACM: New York, NY, USA, 1999; 256–275, doi: 320384.320408. URL http://doi.acm.org/10.1145/320384.320408.

[8] Moon DA. Object-oriented programming with Flavors. *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, vol. 21, 1986; 1–8.

[9] Bracha G, Cook W. Mixin-based inheritance. *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, vol. 25, 1990; 303–311.

[10] Schärli N, Ducasse S, Nierstrasz O, Black AP. Traits: Composable units of behavior. *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, *LNCS*, vol. 2743, Springer Verlag: Berlin Heidelberg, 2003; 248–274, doi:10.1007/b11832. URL http://scg.unibe.ch/archive/papers/Scha03aTraits.pdf.

[11] Ducasse S, Nierstrasz O, Schärli N, Wuyts R, Black AP. Traits: A mechanism for fine-grained reuse. *TOPLAS: ACM Transactions on Programming Languages and Systems* Mar 2006; **28**(2):331–388, doi:10.1145/1119479.1119483. URL http://scg.unibe.ch/archive/papers/Duca06bTOPLASTraits.pdf.

[12] Matsumoto Y. *Ruby in a Nutshell*. O'Reilly: 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2001.

[13] Gîrba T. The moose book 2010. URL http://www.themoosebook.org/book.

[14] Nierstrasz O, Ducasse S, Gîrba T. The story of Moose: an agile reengineering environment. *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, ACM Press: New York, NY, USA, Sep. 2005; 1–10, doi:10.1145/1095430.1081707. URL http://scg.unibe.ch/archive/papers/Nier05cStoryOfMoose.pdf, invited paper.

[15] Tichelaar S, Ducasse S, Demeyer S, Nierstrasz O. A meta-model for language-independent refactoring. *Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00)*, IEEE Computer Society Press: Los Alamitos, CA, 2000; 157–167, doi:10.1109/ISPSE.2000.913233. URL http://scg.unibe.ch/archive/papers/Tich00bRefactoringMetamodel.pdf.

[16] Perin F. MooseJEE: A Moose extension to enable the assessment of JEAs. *Proceedings of the 26th International Conference on Software Maintenance (ICSM 2010) (Tool Demonstration)*, Timişoara, Romania, Sep. 2010; 1–4, doi:10.1109/ICSM.2010.5609569. URL http://scg.unibe.ch/archive/papers/Peri10cMooseExtension.pdf.

[17] Perin F, Gîrba T, Nierstrasz O. Recovery and analysis of transaction scope from scattered information in Java enterprise applications. *Proceedings of International Conference on Software Maintenance 2010*, Timişoara, Romania, Sep. 2010; 1–10, doi:10.1109/ICSM.2010.5609572. URL http://scg.unibe.ch/archive/papers/Peri10aTransactionRecovery.pdf.

[18] Snyder A. Encapsulation and inheritance in object-oriented programming languages. *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, vol. 21, 1986; 38–45, doi:10.1145/28697.28702.

[19] Sakkinen M. Disciplined inheritance. *Proceedings ECOOP '89*, Cook S (ed.), Cambridge University Press: Nottingham, 1989; 39–56.

[20] Goldberg A, Robson D. *Smalltalk 80: the Language and its Implementation*. Addison Wesley: Reading, Mass., May 1983. URL http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf.

[21] Bergel A, Ducasse S, Nierstrasz O, Wuyts R. Stateful traits. *Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006)*, *LNCS*, vol. 4406, Springer: Berlin Heidelberg, 2007; 66–90, doi:10.1007/978-3-540-71836-9_3. URL http://scg.unibe.ch/archive/papers/Berg07aStatefulTraits.pdf.

[22] Ressia J, Renggli L, Gîrba T, Nierstrasz O. Run-time evolution through explicit meta-objects. *Proceedings of the 5th Workshop on Models@run.time at the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)*, Oslo, Norway, Oct. 2010; 37–48. URL http://scg.unibe.ch/archive/papers/Ress10a-RuntimeEvolution.pdf, http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-641/.

[23] McAffer J. Engineering the meta level. *Proceedings of the 1st International Conference on Metalevel Architectures and Reflection (Reflection 96)*, Kiczales G (ed.), San Francisco, USA, 1996.

[24] Bracha G, von der Ahé P, Bykov V, Kashai Y, Maddox W, Miranda E. Modules as objects in Newspeak. *ECOOP'10: Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP'10, Springer-Verlag: Berlin, Heidelberg, Jun. 2010; 405–428, doi:10.1007/978-3-642-14107-2_20. URL http://bracha.org/newspeak-modules.pdf.

[25] Mens T, van Limberghen M. Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems* 1996; **3**(1):1–30.

[26] Flatt M, Krishnamurthi S, Felleisen M. Classes and mixins. *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press: New York, NY, USA, 1998; 171–183, doi:10.1145/268946.268961. URL http://www.cs.brown.edu/~sk/Publications/Papers/Published/fkf-classes-mixins/.

[27] Ancona D, Lagorio G, Zucca E. Jam - a smooth extension of java with mixins. *ECOOP 2000 Object-Oriented Programming*, *Lecture Notes in Computer Science*, vol. 1850, Bertino E (ed.). Springer Berlin, Heidelberg, Jun. 2000; 154–178, doi:10.1007/3-540-45102-1_8. URL http://dx.doi.org/10.1007/3-540-45102-1_8.

[28] Bracha G. The programming language Jigsaw: Mixins, modularity and multiple inheritance. PhD Thesis, Dept. of Computer Science, University of Utah Mar 1992.

[29] DeMichiel LG, Gabriel RP. The Common Lisp object system: An overview. *Proceedings ECOOP '87*, *LNCS*, vol. 276, Bézivin J, Hullot JM, Cointe P, Lieberman H (eds.), Springer-Verlag: Paris, France, 1987; 151–170.

[30] Lawless JA, Milner MM. *Understanding Clos the Common Lisp Object System*. Digital Press: 225 Wildwood St., Woburn, MA 01801, 1989.

[31] Paepcke A. User-level language crafting. *Object-Oriented Programming: the CLOS perspective*. MIT Press: Cambridge, MA, USA, 1993; 66–99.

[32] Ducournau R, Habib M, Huchard M, Mugnier M. Monotonic conflict resolution mechanisms for inheritance. *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, vol. 27, 1992; 16–24.

[33] Smith C, Drossopoulou S. Chai: Typed traits in Java. *Proceedings ECOOP 2005*, Springer Verlag: Berlin Heidelberg, Glasgow, Scotland, Jul. 2005; 453–468.

[34] Bettini L, Capecchi S, Venneri B. Featherweight java with dynamic and static overloading. *Science of Computer Programming* Mar Mar 2009; **74**:261–278, doi:10.1016/j.scico.2009.01.007. URL http://portal.acm.org/citation.cfm?id=1518341.1518655.

[35] Ungar D, Smith RB. Self: The power of simplicity. *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, vol. 22, 1987; 227–242, doi:10.1145/38765.38828.

[36] Lieberman H. Using prototypical objects to implement shared behavior in object oriented systems. *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, vol. 21, Nov. 1986; 214–223, doi:10.1145/960112.28718. URL http://web.media.mit.edu/~lieber/Lieberary/OOP/Delegation/Delegation.htmlhttp://reference.kfupm.edu.sa/content/u/s/using_prototypical_objects_to_implement__76339.pdf.

[37] Ghelli G. Foundations for extensible objects with roles. *Inf. Comput.* 2002; **175**(1):50–75.

[38] Di Gianantonio P, Honsell F, Liquori L. A lambda calculus of objects with self-inflicted extension. *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, ACM: New York, NY, USA, 1998; 166–178, doi:10.1145/286936.286955.

[39] Drossopoulou S, Damiani F, Dezani-Ciancaglini M, Giannini P. Fickle: Dynamic object re-classification. *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, Springer-Verlag: London, UK, UK, Jun. 2001; 130–149. URL http://portal.acm.org/citation.cfm?id=646158.680007.

[40] Cohen T, Gil JY. Three approaches to object evolution. *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, ACM: New York, NY, USA, 2009; 57–66, doi:10.1145/1596655.1596665.

[41] Rashid A, Aksit M ((eds.)). *Transactions on Aspect-Oriented Software Development II*, *Lecture Notes in Computer Science*, vol. 4242, Springer: Berlin, Heidelberg, 2006.

[42] Madsen OL, Møller-Pedersen B, Nygaard K. *Object-Oriented Programming in the Beta Programming Language*. Addison Wesley: Reading, Mass., 1993.

[43] Costanza P, Hirschfeld R. Language constructs for context-oriented programming: An overview of ContextL. *Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05*, ACM: New York, NY, USA, Oct. 2005; 1–10, doi:10.1145/1146841.1146842. URL http://p-cos.net/documents/contextl-overview.pdf.

[44] Smith RB, Ungar D. A simple and unifying approach to subjective objects. *TAPOS special issue on Subjectivity in Object-Oriented Systems* Dec 1996; **2**(3):161–178, doi:10.1002/(SICI)1096-9942(1996)2:3%3C161::AID-TAPO3%3E3.0.CO;2-Z. URL http://www.mip.sdu.dk/~bnj/library/Us_Ungar.pdf.

[45] Kristensen BB. Object-oriented modeling with roles. *Proceedings of the 2nd International Conference on Object-Oriented Information Systems*, Murphy J, Stone B (eds.), Springer-Verlag: London , UK, 1995; 57–71.

[46] Darderes B, Prieto M. Subjective behavior: a general dynamic method dispatch. *OOPSLA Workshop on Revival of Dynamic Languages*, British Columbia, Canada, Oct. 2004; 1–6. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.7749&rep=rep1&type=pdf.

[47] Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier JM, Irwin J. Aspect-oriented programming. *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming*, *LNCS*, vol. 1241, Aksit M, Matsuoka S (eds.), Springer-Verlag: Jyvaskyla, Finland, 1997; 220–242, doi:10.1007/BFb0053381.

[48] Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An overview of AspectJ. *Proceedings ECOOP 2001*, no. 2072 in LNCS, Springer Verlag: London, UK, 2001; 327–353.

[49] Fowler M. *Patterns of Enterprise Application Architecture*. Addison Wesley: 75 Arlington Street Suite 300, Boston MA, 02116 USA, 2005.

[50] Lincke J, Appeltauer M, Steinert B, Hirschfeld R. An open implementation for context-oriented layer composition in contextjs. *Science of Computer Programming* Dec Dec 2011; **76**:1194–1209, doi:10.1016/j.scico.2010.11.013. URL http://dx.doi.org/10.1016/j.scico.2010.11.013.

[51] Langone D, Ressia J, Nierstrasz O. Unifying subjectivity. *Proceedings of the 49th International Conference on Objects, Models, Components and Patterns (TOOLS'11)*, *LNCS*, vol. 6705, Springer-Verlag, Jun. 2011; 115–130, doi:10.1007/978-3-642-21952-8_10. URL http://scg.unibe.ch/archive/papers/Lang11aSubjectivity.pdf.

[52] Bergel A, Nierstrasz O, Renggli L, Ressia J. Domain-specific profiling. *Proceedings of the 49th International Conference on Objects, Models, Components and Patterns (TOOLS'11)*, *LNCS*, vol. 6705, Springer-Verlag: Berlin, Heidelberg, 2011; 68–82, doi:10.1007/978-3-642-21952-8_7. URL http://scg.unibe.ch/archive/papers/Berg11b-Profiling.pdf.

[53] Ressia J, Bergel A, Nierstrasz O. Object-centric debugging. *Proceeding of the 34rd international conference on Software engineering*, ICSE '12, Jun. 2012, doi:10.1109/ICSE.2012.6227167. URL http://scg.unibe.ch/archive/papers/Ress12a-ObjectCentricDebugging.pdf.

[54] Bracha G. Generics in the java programming language Jul 2004. Java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[55] Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional: Reading, Mass., 1995.

[56] Aryani A, Perin F, Lungu M, Mahmood AN, Nierstrasz O. Can we predict dependencies using domain information? *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE 2011)*, Oct. 2011, doi:10.1109/WCRE.2011.17. URL http://scg.unibe.ch/archive/papers/Aria11aWCRE11.pdf.

[57] Bunge P. Scripting browsers with Glamour. Master's thesis, University of Bern Apr 2009. URL http://scg.unibe.ch/archive/masters/Bung09a.pdf.