Talents: Dynamically Composable Units of Reuse

{Pre-proceedings version}

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

Software Composition Group, University of Bern, Switzerland

http://scg.unibe.ch/

Abstract

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 underline 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 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.

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. Due to this several forms of multiple inheritance have been proposed [3, 19, 28, 36, 38]. However, multiple inheritance introduces problems that are difficult to resolve [10, 39]. 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 [29] and mixins [5], 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 [12, 37] 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.

Ruby [25] 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 which model features that 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 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 a Smalltalk prototype of our approach.

- We describe two flattening techniques which merge the behavior adaptations into the original class hierarchy or into a new class.

Outline. In Section 2 we motivate the problem. Section 3 explains the talent approach, its composition operations and a solution to the motivating problem. the flattening techniques. 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 discussed about talents features like state sharing, scoping and flattening. In Section 7 we present examples to illustrate the various used of talents. Section 8 summarizes the paper and discusses future work.

2. Motivating Example

Moose is a platform for software and data analysis that provides facilities to model, query, visualize and interact with data [17, 30]. Moose represents the source code in a model described by FAMIX, a language-independent meta-model [40]. 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.

Moose can model applications written in different programming languages, like 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 `namespace` to the `FAMIXClass`, while the Smalltalk extension adds the method `isExtended`. Smalltalk however does not support namespaces, and Java does not support extended classes. 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 the developer productivity and it is error prone.

A possible solution is to create subclasses for each supported language. However, there are some situation in which the model requires a combination of extensions: Moose JEE [32, 33] — 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, the diamond problem makes it difficult to handle the situation where two languages 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 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.

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 presented in other approaches.

The prototype of talents[1] and the examples presented in this paper are implemented in Pharo Smalltalk[2], an open-source Smalltalk [18] implementation.

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. There is one more particularity with this example.

[1] `http://scg.unibe.ch/research/talents/`

[2] `http://www.pharo-project.org/`

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

```
aTalent := Talent new.
aTalent excludeMethod: #clientClasses.
aClass := FAMIXClass new.
aClass acquire: aTalent.
```

In this case the existing method `clientClasses` 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. 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.

```
javaClassTalent := Talent new.
javaClassTalent
   defineMethod: #namespace
   do: '^ self owningScope'.
jeeClassTalent := Talent new.
jeeClassTalent
   defineMethod: #isTestClass
   do: '^ self methods anySatisfy: [ :each | each
     isTestMethod ]'.
aClass := FAMIXClass new.
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.

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.

```
javaClassTalent := Talent new.
javaClassTalent
   defineMethod: #isTestClass
   do: '^ self methods anySatisfy: [ :m | m
     isAnnotatedWith: #Test ]'.
jeeClassTalent := Talent new.
jeeClassTalent
   defineMethod: #isTestClass
   do: '^ self inheritsFrom: #TestCase'.
aClass := FAMIXClass new.
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:

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

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.

```
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.

4. Implementation

In this section we describe how talents are implemented.

4.1 Bifröst

Talents are built on top of the Bifröst reflection framework [35]. 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.

In recent years researchers have worked on the idea of applying traditional object-oriented techniques to the meta-level while attempting to solve various practical problems

motivated by applications [26]. 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.

4.2 Talents

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 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. Talents do not affect the behavior of development tools in any way. 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.

5. Related Work

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

Mixins

Flavors [29] 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 [5], Mens and van Limberghen [27], Flatt, Krishnamurthi and Felleisen [14], and Ancona, Lagorio and Zucca [1].

Mixins present drawbacks when dealing with composition. Mixins use single inheritance for composing features and extending classes. Mixins have to be composed linearly thus limiting the ability to define the glue code necessary to avoid conflicts. 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 [4], 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 [25] 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 [8] is an object-oriented extension of Lisp. Multiple inheritance in CLOS [22, 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 [13]. 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, render-

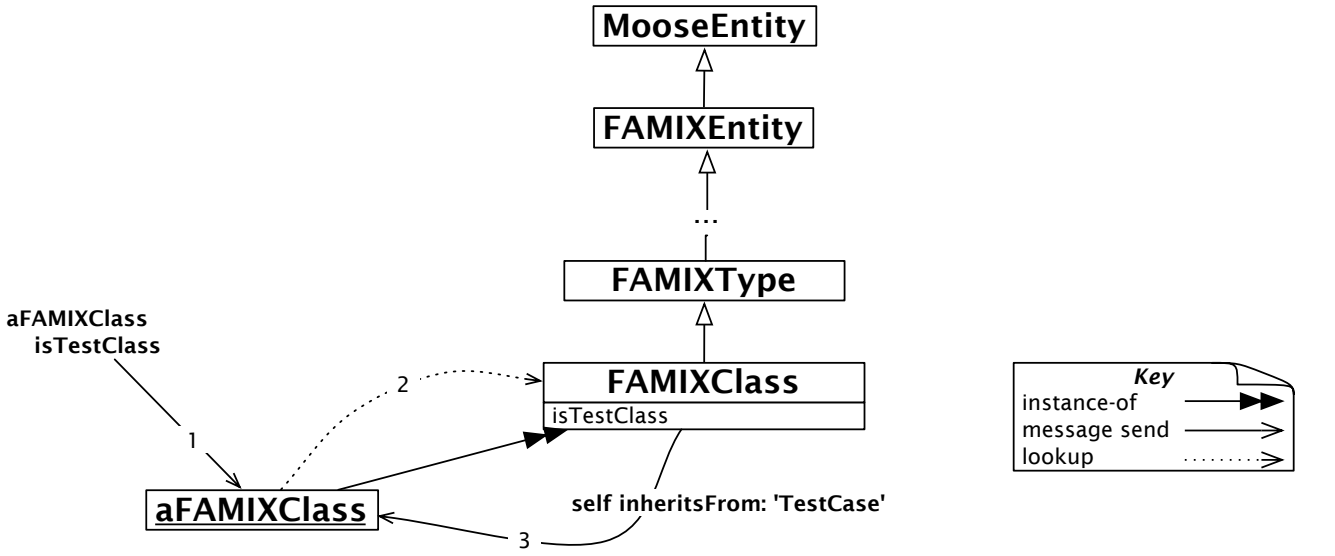

Figure 1. Default message send and method look up resolution.

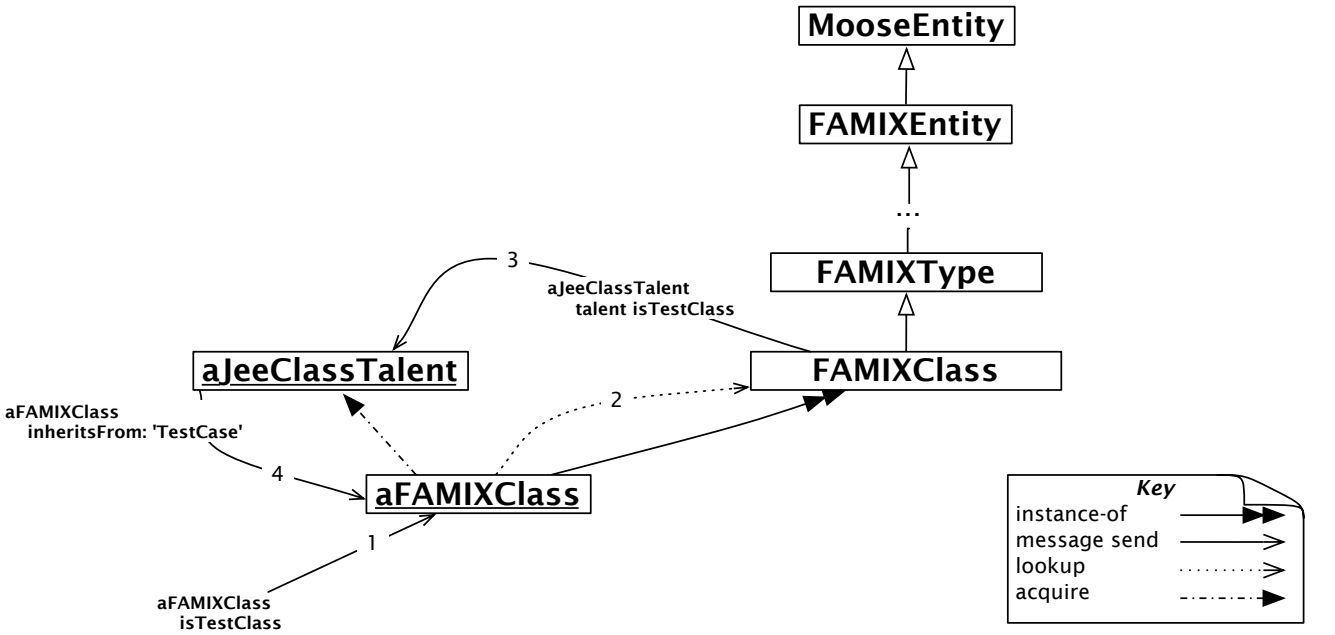

Figure 2. Talent modeling the Moose FAMIX class behavior for the method `isTestClass`.

ing this technique dependent on custom code provided by the user.

Traits

Traits [12, 37] 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.

Object Extensions

Self [41] is a prototype-based language which follows the concepts introduced by Lieberman [23]. 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.

Object extension [9, 16] 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.* [16] suggested a calculus in which conflicting changes cannot occur, by letting the same object assume different roles in different contexts.

Drossopoulou proposed Fickle [11], 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 [7]. This approach proposes three variants of evolution, relying on inheritance, mixins and shakeins [34]. The authors introduce the notion of evolvers, a mechanism for maintaining class invariants in the course of reclassification [11]. 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.* [6] 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 [24]. 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.

Aspect Oriented Programming

Aspect Oriented Programming (AOP) [20] provides a general model for modularizing cross cutting concerns. Join points define points in the execution of a program that trigger the execution of additional cross-cutting code called advice. Join points can be defined on the run-time model (i.e., dependent on control flow). Both aspects and talents can add new methods to existing classes. Most implementations of aspect-oriented programming such as AspectJ [21] 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 Smalltalk.

6.1 State

Bifröst structural meta-objects provide features for adding and removing state from a single object. Theoretically, talents can provide something that trait cannot, state. Moreover, talents will provide operators for composing state adaptations. This composition is not present in object-specific techniques like mixins and Newspeak modules.

However, stateful traits [2] have shown that state composition is not simple to achieve.

6.2 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 extend 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 two examples to understand the motivation for talents scoping.

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 . 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.

6.3 Flattening

Flattening is the technique that folds into a class all the behavior that has been added to and 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.

7. Examples

In this section we present a number of example applications of talents.

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:`, we are not interested in the complexities of `assetsOf:`

```
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.

7.2 Compiler examples

Cohen and Gil provide an example in the context of object evolution [7]. In many compiler designs, the parser generates an Abstract Syntax Tree (AST) from the source code; the back-end then processes this tree. Often, the parser does not have the knowledge required for classifying a given AST node at its most refined representation level. For example, in the Smalltalk compiler's parser a variable access is models as an `ASTVariableNode`, there is no distinction between instance variables, class variables, global or temporals. As the compiler advances through its phases these AST nodes are going to be classified in an abstraction called the lexical scope tree. However, when analyzing the AST structure we require information about types of variables and scopes. Using talents we can add behavior to AST variable nodes to specify them as instance variable, class variables and temporals. We can also remove methods from AST nodes that do not make sense for the new specification of the node. The talent composition mechanism will be particularly useful in merging these different talents on AST nodes.

7.3 State Pattern

The state pattern [15] 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 [15] 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.

8. Conclusion

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 development tools are unaware of them. We plan on developing a user interface which takes talents into account both helping in their definition and composition.

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.

Acknowledgments

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).

References

[1] D. Ancona, G. Lagorio, and E. Zucca. Jam — a smooth extension of Java with mixins. In *ECOOP 2000*, number 1850 in Lecture Notes in Computer Science, pages 145–178, 2000.

[2] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. *Journal of Computer Languages, Systems and Structures*, 34(2-3):83–108, 2008. ISSN 1477-8424. doi: 10.1016/j.cl.2007.05.003.

[3] A. H. Borning and D. H. Ingalls. Multiple inheritance in Smalltalk-80. In *Proceedings at the National Conference on AI*, pages 234–237, Pittsburgh, PA, 1982.

[4] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Dept. of Computer Science, University of Utah, Mar. 1992.

[5] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, Oct. 1990.

[6] G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as objects in newspeak. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 405–428, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14106-4, 978-3-642-14106-5.

[7] T. Cohen and J. Y. Gil. Three approaches to object evolution. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 57–66, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-598-7. doi: 10.1145/1596655.1596665.

[8] L. G. DeMichiel and R. P. Gabriel. The Common Lisp object system: An overview. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP '87*, volume 276 of *LNCS*, pages 151–170, Paris, France, June 1987. Springer-Verlag.

[9] P. Di Gianantonio, F. Honsell, and L. Liquori. A lambda calculus of objects with self-inflicted extension. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 166–178, New York, NY, USA, 1998. ACM. ISBN 1-58113-005-8. doi: 10.1145/286936.286955.

[10] R. Dixon, T. McKee, M. Vaughan, and P. Schweizer. A fast method dispatcher for compiled languages with multiple inheritance. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 211–214, Oct. 1989.

[11] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 130–149, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42206-4.

[12] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, Mar. 2006. ISSN 0164-0925. doi: 10.1145/1119479.1119483.

[13] R. Ducournau, M. Habib, M. Huchard, and M. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, volume 27, pages 16–24, Oct. 1992.

[14] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, 1998. ISBN 0-89791-979-3. doi: 10.1145/268946.268961.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, Reading, Mass., 1995. ISBN 978-0201633610.

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

[17] T. Gîrba. *The Moose Book*. Self Published, 2010.

[18] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983. ISBN 0-201-13688-0.

[19] S. E. Keene. *Object-Oriented Programming in Common-Lisp*. Addison Wesley, 1989.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings ECOOP 2001*, number 2072 in LNCS, pages 327–353. Springer Verlag, 2001.

[22] J. A. Lawless and M. M. Milner. *Understanding Clos the Common Lisp Object System*. Digital Press, 1989.

[23] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 214–223, Nov. 1986. doi: 10.1145/960112.28718.

[24] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison Wesley, Reading, Mass., 1993. ISBN 0-201-62430-3.

[25] Y. Matsumoto. *Ruby in a Nutshell*. O'Reilly, 2001. ISBN 0596002149.

[26] J. McAffer. Engineering the meta level. In G. Kiczales, editor, *Proceedings of the 1st International Conference on Metalevel*

Architectures and Reflection (Reflection 96), San Francisco, USA, Apr. 1996.

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

[28] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[29] D. A. Moon. Object-oriented programming with Flavors. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 1–8, Nov. 1986.

[30] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. ISBN 1-59593-014-0. doi: 10.1145/1095430.1081707. Invited paper.

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

[32] F. Perin. Moosejee: A moose extension to enable the assessment of jeas. In *Proceedings of the 26th International Conference on Software Maintenance (ICSM 2010) (Tool Demonstration)*, Sept. 2010.

[33] F. Perin, T. Gîrba, and O. Nierstrasz. Recovery and analysis of transaction scope from scattered information in Java enterprise applications. In *Proceedings of International Conference on Software Maintenance 2010*, Sept. 2010.

[34] A. Rashid and M. Aksit, editors. *Transactions on Aspect-Oriented Software Development II*, volume 4242 of *Lecture Notes in Computer Science*, 2006. Springer. ISBN 3-540-48890-1.

[35] J. Ressia, L. Renggli, T. Gîrba, and O. Nierstrasz. Runtime evolution through explicit meta-objects. In *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)*, pages 37–48, Oct. 2010.

[36] C. Schaffert, T. Cooper, B. Bullis, M. Killian, and C. Wilpolt. An Introduction to Trellis/Owl. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 9–16, Nov. 1986.

[37] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003. ISBN 978-3-540-40531-3. doi: 10.1007/b11832.

[38] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, Reading, Mass., 1986. ISBN 0-201-53992-6.

[39] P. F. Sweeney and J. Y. Gil. Space and time-efficient memory layout for multiple inheritance. In *Proceedings OOPSLA '99*, pages 256–275. ACM Press, 1999. ISBN 1-58113-238-7. doi: 10.1145/320384.320408.

[40] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00)*, pages 157–167. IEEE Computer Society Press, 2000. doi: 10.1109/ISPSE.2000.913233.

[41] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, Dec. 1987. doi: 10.1145/38765.38828.