Classes = Traits + States + Glue

Beyond mixins and multiple inheritance[1]

Nathanael Schaerli, Stéphane Ducasse, Oscar Nierstrasz

Software Composition Group, University of Berne[2]

1. Submission to the The Inheritance Workshop at ECOOP 2002.

2. *Author's address:* Institut für Informatik (IAM), Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland. *Tel:* +41 (31) 631.4618. *Fax:* +41 (31) 631.3965. *E-mail:* schaerli@iam.unibe.ch. *WWW:* www.iam.unibe.ch/~scg

ABSTRACT We present a simple, component-based model of mixins, called *traits*, and argue that this simple model sidesteps many of the practical problems with other approaches to mixins and multiple inheritance. With our model, classes are built from a set of *traits by* specifying *glue code* that connects them together and accesses the necessary *state*. We briefly discuss practical experience with an implementation of traits for Squeak, and we list a number of open questions for discussion.

1. INTRODUCTION

Over the years, many programmers have lamented the fact that single inheritance is not expressive enough to factor out common features (i.e., instance variables or methods) shared by classes in a complex hierarchy. This fact has led many language designers to propose various forms of multiple inheritance [11][12][16] for programming languages, as well as other mechanisms, such as mixins [13], that allow classes to be incrementally composed from sets of features.

Despite countless proposals having been implemented, both multiple inheritance and mixins continue to be considered controversial features that only find limited application in practice [18]. We believe the main reasons for this state of affairs are as follows:

- *Complicated solutions to conflict resolution:* any approach to multiple inheritance or mixins must provide some way to resolve conflict that arise when conflicting features are inherited along different paths. The most insidious case is when conflicting *state* is multiply inherited [10]. Even if the declarations are consistent, it is not clear whether the same state should be inherited just once or multiply [12]. Standard solutions are to provide either some built-in linearization algorithm in the language [11] or to provide the programmer with renaming mechanisms to resolve the conflicts [12]. In either case, programmers have a tough time getting the behaviour they want.
- *Hard to design reusable artifacts:*whatever scheme is chosen, it is hard to design reusable mixins or classes that can be composed flexibly without leading to conflicts.
- *Fragile hierarchies:* class hierarchies that are built using either multiple inheritance or mixins tend to be very fragile with respect to changes in the base classes or mixins. Changes tend to break the complicated conflict resolution algorithms or code, and lead to hard to debug anomalies [8].

The solution adopted by the designers of Java was to abandon multiple inheritance of implementation, and only support multiple inheritance of interfaces. (A class may *inherit* from at most one superclass, but may *implement* many interfaces.) The same interface inherited along multiple paths does not pose a problem, whereas conflicting interfaces are clearly an error in any case.

We propose a simple component-based approach that sidesteps most of the problems we have identified, yet offers more than Java's solution:

- We distinguish classes and *traits*. (See also traits in Mesa [7] and SELF [20])
- Traits *provide* a set of *services*, i.e., features that implement behaviour (methods), but not state (no instance variables).
- Traits also *require* a set of services, i.e., those used by the services provided.
- Traits never directly access state, but only indirectly, through required accessor services.
- A class can be constructed from a set of traits by providing the necessary state and the missing services. These services represent the *glue code* because they specify how the traits are connected together and how possible conflicts are resolved.
- Simple tools are provided to keep track of the dependencies between traits and classes.

We will illustrate the model by means of an example in section 2. We compare our traits model to other approaches in section 3. We briefly illustrate how traits are implemented in Squeak and how they benefit from tools support in section 4. We conclude with a list of open questions in section 5.

2. THE TRAITS MODEL

In the following, we present our model of traits. Although the model can be applied to different types of programming languages, this paper focuses on object-oriented languages with single inheritance. Whereas many other inheritance models focus on providing features such as method renaming or so-

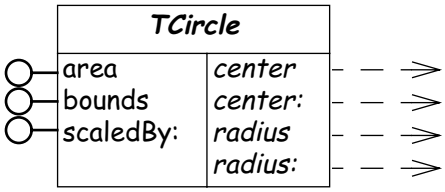

Figure 1 A Trait with provided and required services

phisticated rules for conflict resolution, we try to go into the other direction and focus on simplicity. The main goal is to support the programmer in writing code that is easy to understand and does not show unexpected or surprising behaviour.

2.1 Classes and traits

In our model, a trait is the most primitive unit of code reuse. A trait is a component that *provides* a set of services, and may also *require* some services. The required services are the *plugs* that must be connected before the trait can function, and the provided services are the *sockets* that can be plugged into other traits.

Traits differ from classes in that they do not contain any kind of state and they do not support inheritance. Figure 1 shows (in a kind of pseudo-UML) a trait TCircle encapsulating services that might be of use to various kinds of Circle classes. The left column represents the provided behaviour (the *sockets*). For conciseness, it only consists of the services area, bounds and scaledBy:. In the right column, there are the required services center, center:, radius, and radius. They represent the *plugs* that must be connected when the trait is used.

Whereas traits are used to implement a parameterized and reusable behaviour, classes can typically be instantiated and specify some state together with a relatively concrete functionality that may consist of multiple different aspects. In order to specify classes in a more high-level way, they can be built as the composition of zero or more traits. When a class uses a trait, the behaviour provided by the trait gets incorporated into the class. This means that the semantics is basically the same as if the services (methods) provided by the trait were implemented in the class itself. However, there are two exceptions:

- *Overriding.* Methods implemented directly in the class have higher precedence than equally named services provided by the incorporated traits. This means that methods defined in the class override equally labelled services that are provided by the used traits.
- *Conflict resolution.* All the traits that are used by a class have the same precedence. Therefore, conflicting services (services with names that are provided by more than one trait) have to be explicitly resolved. In our model, this can be done in two different ways: First, the class can implement its own variant of the conflicting service, which then overrides all the implementations provided by the traits and therefore resolves the conflict. Alternatively, a class can specify a set of conflict resolutions, which associate a service name to a certain mixin. This explicitly defines that the service defined by this mixin takes precedence.

In order to be *complete*, a class has to provide an implementation for every service required by any of the used traits. That is, *every plug must be connected to some socket*. These services define how the traits are glued into the class, and therefore, they represent the *glue code*. The glue code can be implemented in the class itself, in a direct or indirect superclass, or in another trait that is used by the class. It is important to note that our model allows traits to be composed from other traits in the same way that classes are. But unlike classes, traits do not have to be complete, which means that they do not have to define all the services that are required by the incorporated traits. Unconnected plugs of the constituent traits simply become plugs of the composite trait.

Also note that trait composition does not subsume single inheritance. Inheritance is still needed for a class to reuse the features provided by another class. In particular, representation can only be shared by inheritance, and super only makes sense in the context of single inheritance.

Figure 2 shows how a class ColoredVisualCircle is built as the composition of the traits TCircle, TColor, and TVisual. In order to properly use these traits, the class has to ensure that there is a glue method for all the required services and that there is no unresolved method conflict. For the trait TCircle, the requirements are resolved by implementing accessors that associate center and radius to instance variables. Alternatively, we could create a more specialized Circle class that only uses one instance variable for center and automatically adjusts the radius so that the circle always goes through the origin of the coordinate system. In a similar way, the requirements rgb and rgb: of the TColor trait are associated to an instance variable rgb.

The requirements of the trait TVisual are a bit more interesting. Since TCircle already provides a service bounds returning the bounding box of the circle, one of the requirements is already fulfilled and we do not have to explicitly specify a method in the class. (However, we would be able to override it if necessary). For the second requirement drawOn: we implement a method that draws the circle on the canvas that is provided as the argument form. We thereby use the colour specified by the rgb method of the TColor trait. We also have one method conflict because both TCircle and TVisual provide a method scaledBy:. We resolve this conflict by implementing a method that scales the circle and then updates the visual representation.

This example illustrates how the usage of traits supports very flexible code reuse. In addition, it provides the class with a structure that makes it much easier to understand, because we only have to understand the glue methods and do not have to bother about all the other methods that are provided by the traits. (The provided methods only depend on the require-

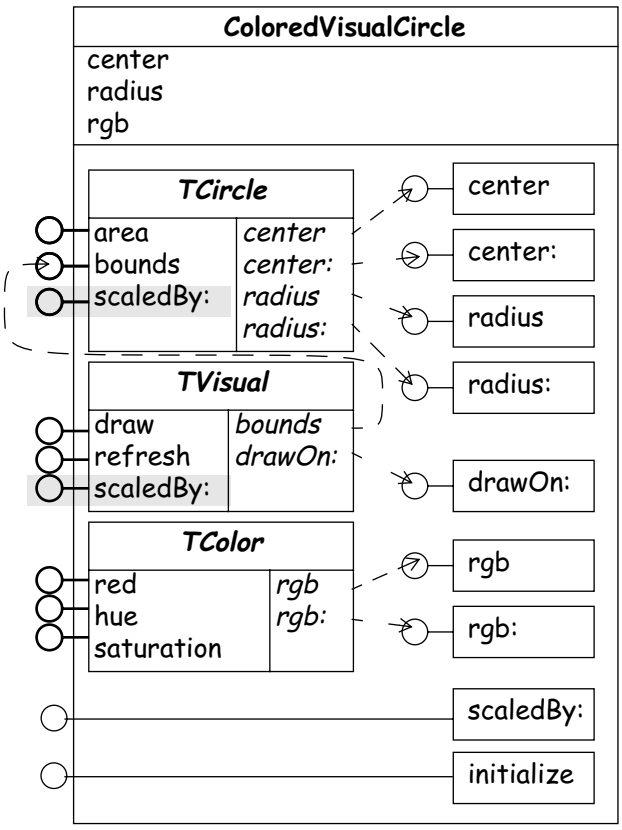

Figure 2 Composing Traits to create a Class

ments, and therefore, we can be sure that they are working correctly once we have provided the necessary glue).

This structure is especially helpful if we have appropriate programming tools (cf. section 4). Using those, we see that the class is implemented as the composition of three different traits, and there are views that immediately show the implementations of the glue code that is specified for each of these traits. As an example, there would be a view that shows the glue methods for the trait TCircle, which are center, center:, radius, radius: and scaledBy:. This allows us to immediately understand how this trait is glued into the class. Similarly, there would be a view that shows the glue code for the trait TVisual, which consists of the methods bounds, drawOn: and scaledBy:. This view shows us that the method drawOn: is provided by the class itself, whereas the glue method bounds is provided by another trait, namely TCircle. Furthermore, it shows that the glue method scaledBy: implemented in the class resolves the conflict that arises because the traits TVisual and TCircle provide services with identical names.

2.2 Trait composition vs. inheritance

We have pointed out that trait composition is not intended to subsume inheritance. The following example shows how the two may be fruitfully combined to build powerful abstractions. Let's assume that there is a framework providing several classes that use the methods read and write to access some data in an unsynchronized way. Using both inheritance and traits, we can synchronize these data accesses in a reusable way as follows:

First, we write a trait TSyncReadWrite that provides two methods read and write, which ensure synchronization before they call the original implementation of read respectively write. In Smalltalk, the implementation of the method read could look as follows:

```
TSyncReadWrite>>read
  self waitForSemaphore.
  ^ super read.
```

Then, we create a subclass of all the classes that should use this synchronized data access. In each of these subclasses, we incorporate the trait TSyncReadWrite, which results in classes that provide the synchronized data access behaviour. It is important to understand that the keyword super specified in the method read of the trait TSyncReadWrite is not bound in the trait, because traits do not support inheritance. Instead, super refers to the superclass of the class that incorporates the trait, and thus, the expression "super read" calls the unsynchronized read method.

3. COMPARISON TO MIXINS AND OTHER INHERITANCE MODELS

In the introduction, we have pointed out that there are various other strategies and models that have been used in object-oriented languages to achieve better code reuse and more flexibility. In particular, there is another model that uses entities called traits as an approach to multiple inheritance [7]. The main difference to our model is that those traits still carry state, that they do not support explicitly required services and that they allow multiple implementations for a single service. Also in SELF [20], there is a notion called traits that is used to share behavior amongst prototypes.

In the following, we compare the other popular reuse strategies to our approach and justify our design decisions. Please note that for all of these strategies, there are several different variants, and we focus on the most common ones. Thus, it maybe the case that some of our statements do not apply to more exotic variants.

3.1 Multiple inheritance

Multiple inheritance allows a class to inherit from more than one parent class. Many object-oriented programming languages such as C++ [16], Eiffel [12], CLOS [11], and Python [21] support this concept, and although the basic model is everywhere the same, the implementations differ when it comes to more advanced issues such as conflict resolution.

At a first glance, trait composition is very similar to multiple inheritance. Both approaches allow one to reuse functionality from more than one source at the same time, which has the benefit of better code reuse and allows more flexible class hierarchies. Nevertheless, there are essential differences:

Reuse of traits vs. reuse of classes. Multiple inheritance allows functionality to be reused from multiple classes which

generally implement an already specialized version of a certain behaviour and also include state. This means that the notion of a class is used in two very different ways: On one hand, classes are used to represent already specialized entities that can be instantiated, and at the same time, they represent the most primitive entities of code reuse. With trait composition, these two roles are completely separated and are provided by two different entities. Traits are the primitive entities of code reuse and the majority of the code is implemented in such traits. Their structure is simpler than the one of classes (e.g. no state), which enforces the programmer to write the behaviour in a very general and reusable way. In contrast to traits, classes represent more concrete and specialized functionalities, which typically consist of different aspects that are realized by using appropriate traits and specifying the necessary glue code.

Diamond problem. One of the most troublesome aspects of multiple inheritance is the "diamond problem", which occurs when a class inherits state from the same base class via multiple paths. With traits, we sidestep this problem because traits do not define any state.

Traits do not replace single inheritance. Both traits and multiple inheritance are an extension of traditional single inheritance. However, whereas multiple inheritance replaces single inheritance, traits are an orthogonal concept that coexists with single inheritance. This means that a language with traits still provides all the well-accepted features and benefits of single inheritance: A class can be derived from (at most) one parent class, it can inherit state from (at most) one parent class, and it can explicitly call inherited services in an unambiguous manner.

Simplicity. Many multiple inheritance implementations provide powerful and sophisticated features to resolve and avoid method conflicts. CLOS, for example, allows a programmer to freely define how the inheritance graph is linearized. Eiffel allows explicitly renaming or deleting of inherited methods in order avoid method conflicts, and C++ supports explicit calling of an arbitrary inherited method from within the code of another method. In contrast to that, our trait model has been developed with the main goal of simplicity and clarity. This makes our approach less flexible, but it also enforces cleaner designs and ensures that the resulting code is better understandable.

3.2 Mixins

The notion of parametric heir classes or mixins avoids many of the complications caused by multiple inheritance, but it still allows more flexible class hierarchies and better code reuse than traditional single inheritance. As the first name suggests, a mixin is an abstract subclass; i.e., a subclass definition that maybe applied to different parent classes to create a related family of modified classes [5]. As the first name suggests, a mixin is a uniform extension of many different parent classes with the same set of fields and methods. As such, the concept of mixins allows the programmer to achieve better code reuse without sacrificing the simplicity of linear inheritance chains. Mixins are used in languages such as Ruby [19] and Smallscript [15] and there exist several extensions of Java [3] or Smalltalk with mixins.

The main differences between mixins and traits are twofold. First, mixins are just a more general form of classes, which means that they usually include state in the same way classes do and that there is often no explicit notion of required and provided services (Jam is an exception [3]). Second, the extension of a class with a mixin always results in a new class, and in the same way, composition of two mixins yields another one. This avoids the problems caused by multiple inheritance paths, but it also leads to the following problems:

- The programmer has to specify a well-defined order in which different mixins are applied. This gets rather unnatural when a class is built from many different mixins that are mostly orthogonal, because there may not be a natural order in which to compose them.
- Multiple mixins cannot be glued together in a single entity, because multiple mixins can only be applied one after the other. Certain work on module mixins [6][22] proposes some operation to manipulate the visibility but does not allow the definition of glue.
- There can be an explosion of classes or mixins. In most object-oriented languages, there are no anonymous classes. Thus, extending a class with a mixin results in a new class with an explicit name.

As an example, assume that we would like to use a traditional mixin implementation (such as Jam [3]) to write a class that provides the behaviour corresponding to the mixins Circle, Color, Visual, and Serializable. In order to do that, we have to define a particular order such as Circle → Color → Visual → Serializable, create explicit entities ColoredCircle, VisualColoredCircle, SerializableVisualColoredCircle, and spread the necessary glue code among all of them.

3.3 Interfaces

Since the appearance of Java, interfaces have become a popular and well-accepted concept. Whereas a class can only inherit from a single parent class, it can be made a subtype of several interfaces by implementing the specified methods. This concept has some similarities to traits, because it also introduces the notion of a more primitive composition that coexists with single inheritance. However, the major difference is that traits are designed to specify reusable behavior whereas interfaces do not support any form of behavior reuse. Indeed, traits define behavior that can be reused by a set of classes not related by inheritance. Interfaces only specify a set of method signatures that classes implementing the interface must implement. As such, the benefit of interfaces is limited to the type system and documentation purposes.

4. THE SQUEAK IMPLEMENTATION AND ITS TOOL SUPPORT

The traits model described in this paper has been developed and implemented in the Squeak programming language, which is a popular open source implementation of Smalltalk-80 [17]. On the level of the language kernel, we have implemented a new first class entity for representing traits and have extended the definition of `Class` and `Metaclass` so that they can incorporate traits. All these entities support reflection in the sense that they can be queried about required services, provided services, overridden services, and so on.

In addition to extending the language kernel, we have worked on tools that support programmers in specifying classes as the composition of traits. Although the traits model has been designed to support a more structured and high-level mode of programming, we have found that suitable tools can make it much easier to manage traits and their composition. The main features of the implemented tools are:

- *Extended browser:* We have extended the classical Smalltalk browser to expose the relationship between classes and traits. A programmer can select a number of different views of a class. The *flattened* view displays the composed class as if it had been programmed without traits. The *glue* view shows the methods that the class adds to glue in the traits. The *traits* view shows individual traits, and exposes how they are glued in. In particular, for a given trait, one can request to see the *provided* services (i.e., methods provided by the trait), the *required services* (i.e., methods implemented by other traits, by a superclass, or by glue methods), the *overridden* services (i.e., methods overridden by glue methods of the class). The browser provides visual feedback indicating whether a class is complete or not (i.e., whether or not all required services are implemented, and all conflicting services are resolved).
- *Type inferencer:* We use a simple type inferencer that is smoothly integrated with the incremental compilation concept of Smalltalk. Whenever a method is added, changed or removed, it gets analysed by the type inferencer. Changes may incur both *local* and *global* consequences. Locally, changes may affect either the list of *provided* or *required* services of a trait. Both are automatically updated. These local changes may in turn have global consequences, introducing new conflicts or causing complete classes to become incomplete. The type inferencer automatically detects the impact and generates a “to do” list of broken classes to be fixed.
- *Automation:* There are several automation tools that support the programmer in composing traits and generating the necessary glue code. Required accessors, for example, can automatically be generated when instance variables are introduced in a composed class. Conflict resolution is also be semi-automated by presenting the programmer with a list of alternative implementations to choose from. The necessary glue code is then automatically generated.

5. FUTURE WORK AND OPEN QUESTIONS

This work was initiated out of frustration with the need to re-implement and duplicate boilerplate code throughout the Squeak class hierarchy. Since we were aware of the difficulties with overly ambitious solutions to multiple inheritance, we sought for a simple, minimal solution that would eliminate the need to duplicate code to be shared across multiple classes.

Initial experience with our prototype implementation is very promising. We now intend to carry out various experiments, and attempt to answer some more fundamental questions:

- *What real impact do traits have on the class hierarchy?* We have started to refactor the Squeak implementation hierarchy to evaluate how useful traits can be in practice.
- *What synergy do traits have with refactoring?* We would like to adapt the refactoring browser [14] so that shared code can be semi-automatically factored out as traits.
- *What design guidelines should drive the development of traits?* To what extent can potential traits be detected automatically by analysis of duplicated code [9] or by means of concept analysis [4].
- *How easily can our model of traits be adapted to other languages?* In which languages can traits be trivial implemented? For example, in C++, traits can likely be implemented as template mixins.
- *What is a suitable operational semantics for traits?* Trait composition is very similar to form composition in the Piccola composition language [1]. We expect that we can use a subset of the semantic foundation used to explain Piccola to formalize our model of traits [2].
- *What kind of type system is needed to reason about traits?* We need a type system that explicitly distinguishes between required and provided services.
- *Will programmers accept working with traits?* Will there be a clear division between programmers who develop reusable traits and those who reuse them, or can we achieve “traits for the common programmer”?

Acknowledgements

We would like to thank all the people involved in this work. In particular, we would like to thank Andrew P. Black for his valuable ideas and suggestions that helped us developing this model.

REFERENCES

[1] Franz Achermann and Oscar Nierstrasz, "Applications = Components + Scripts — A Tour of Piccola," *Software Architectures and Component Technology*, Mehmet Aksit (Ed.), pp. 261-292, Kluwer, 2001.

[2] Franz Achermann, "Forms, Agents and Channels - Defining Composition Abstraction with Style," Ph.D. thesis, University of Berne, January 2002.

[3] Davide Ancona, Giovanni Lagorio and Elena Zucca, "Jam - A Smooth Extension of Java with Mixins," *ECOOP 2000, Lecture Notes in Computer Science 1850*, pp. 145-178, 2000.

[4] Gabriela Arévalo and Tom Mens, "Analysing Object Oriented Framework Reuse using Concept Analysis," Technical Report, 2002, Draft, submitted paper.

[5] Gilad Bracha and William Cook, "Mixin-based Inheritance," *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, October 1990, pp. 303-311.

[6] Gilad Bracha and Gary Lindstrom, "Modularity Meets Inheritance," *Proceedings of the IEEE International Conference on Computer Languages, pages = 282-290*, April 1992.

[7] Gael Curry, Larry Baer, Daniel Lipkie and Bruce Lee, "TRAITS: an Approach to Multiple Inheritance Subclassing," *Proceedings ACM SIGOA, SIGOA Newsletter*, Philadelphia, June 1982, Published as Proceedings ACM SIGOA, SIGOA Newsletter, volume 3, number 12.

[8] R. Ducournau, M. Habib, M. Huchard and M.L. Mugnier, "Monotonic Conflict Resolution Mechanisms for Inheritance," *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, October 1992, pp. 16-24.

[9] Stéphane Ducasse, Matthias Rieger and Georges Golomingi, "Tool Support for Refactoring Duplicated OO Code," *Proceedings of the ECOOP'99 Workshop on Experiences in Object-Oriented Re-Engineering*, Stéphane Ducasse and Oliver Ciupke (Eds.), Forschungszentrum Informatik, Karlsruhe, June 1999, FZI-Report 2-6-6/99.

[10] Dominic Duggan and Ching-Ching Techaubol, "Modular Mixin-Based Inheritance for Application Frameworks," *Proceedings OOPSLA 2001, ACM SIGPLAN Notices*, October 2001, pp. 223-240.

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

[12] Bertrand Meyer, *Object-oriented Software Construction*, Prentice-Hall, 1988.

[13] David A. Moon, "Object-Oriented Programming with Flavors," *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, November 1986, pp. 1-8, Published as Proceedings OOPSLA '86, ACM SIGPLAN Notices, volume 21, number 11.

[14] Don Roberts, John Brant and Ralph E. Johnson, "A Refactoring Tool for Smalltalk," *Theory and Practice of Object Systems (TAPOS)*, vol. 3, no. 4, 1997, pp. 253-263.

[15] Dave Simmons, "SmallScript Language System", www.smallscript.com.

[16] Bjarne Stroustrup, *The C++ Programming Language*, Addison Wesley, Reading, Mass., 1986.

[17] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace and Alan Kay, "Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself," *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, vol. 21, no. 11, November 1997.

[18] Antero Taivalsaari, "On the Notion of Inheritance," *ACM Computing Surveys*, vol. 28, no.3, pp. 438-479, Sept., 1996.

[19] David Thomas and Andrew Hunt, *Programming Ruby*, Addison Wesley, 2001.

[20] David Ungar and Randall B. Smith, "Self: The Power of Simplicity," *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, vol. 22, no. 12, pp. 227-242, Dec., 1987.

[21] Guido van Rossum, "Python Reference Manual," Stichting Mathematisch Centrum, Amsterdam, 1996.

[22] Marc van Limberghen and Tom Mens, "Encapsulation and Composition as Orthogonal Operators on Mixins: A Solution to Multiple Inheritance Problems," *Object Oriented Systems*, vol. 3, no. 1, 1996, pp. 1—30.