Supporting Unanticipated Changes with Traits and Classboxes

Alexandre Bergel Software Composition Group
University of Berne, Switzerland
{bergel}@iam.unibe.ch

Stéphane Ducasse
Language and Software Evolution Group Université de Savoie, France
{ducasse}@iam.unibe.ch

Abstract: On the one hand, traits are a powerful way of structuring classes. Traits support the reuse of method collections over several classes. However, traits cannot be used when specifying unanticipated changes to an application. On the other hand, classboxes are a new module system that supports the local redefinition of classes: a collection of classes can be locally extended with variables and/or methods and the existing clients do not get impacted by changes. However, an extension applied to a class by a classbox cannot be reused for other classes. This paper describes how combining Traits and Classboxes supports the safe introduction of crosscutting collaborations: safe because the existing clients of the classes do not get impacted, crosscutting because collaborations between several classes can be put in place in a unanticipated manner. In the resulting system, a collaboration is represented by a classbox and a role by a trait.

Keywords: open classes, traits, classboxes, modules, mixins, reuse, extensibility, composition, introduction, collaboration, role

1 Introduction

Within the spirit of mixins, traits [4] offer a simple compositional model for structuring object-oriented programs. A trait is essentially a group of methods that serves as a building block for classes and is a primitive unit of code reuse. In this model, classes are composed from a set of traits by specifying glue code that connects the traits together and accesses the necessary state. However, traits do not help in evolving an application.

Within the spirit of modules, classboxes [2] offer a module system for object-oriented languages that provide class extension (*i.e.*, method addition and replacement). Moreover, changes made by a classbox are only visible to that classbox and classboxes that import the modified classes. Such class extensions allow one to define unanticipated modifications on a system. However, extensions for a particular class cannot be reused for another class.

Obviously, these two enhancements of the traditional class model independently go toward a better reuse and adaptability of applications. And they could work in symbiosis. This paper describes the combination of traits and classboxes resulting in a system where classes can be extended with crosscutting aspects in a safe manner: several traits can be applied to collaborating classes, and existing clients of the classes do not get impacted by the changes that can be brought dynamically. Concretely, we first present an enhancement of classboxes that supports the local use of a trait by a class. Then we present an application of the model to express collaborations useful to define a layered software architecture.

In this paper, Section 2 describes traits and illustrates them with an example based on modeling geometrical objects. Section 3 presents classboxes and their assets. Section 4 presents the symbiosis of these two models. Then Section 5 summarizes the idea behind collaborations and illustrates the symbiosis of traits and classboxes by structuring an application of graph traversal. Finally, an evaluation is presented in Section 6, and Section 7 summarizes the paper.

2 Traits

Traits are essentially sets of methods that serve as the behavioral building block of classes and the primitive units of code reuse [4]. Classes (and composite traits) are composed from a set of traits by specifying glue code that connects the traits together and accesses the necessary state. With this approach, classes retain their primary role as generators of instances, while traits are purely units of reuse. As with mixins, classes are organized in a single inheritance hierarchy, thus avoiding the key problems of multiple inheritance, but the incremental extensions that classes introduce to their superclasses are specified using one or more traits.

Several traits can be applied to a class in a single operation: trait composition is unordered. Traits contain method definitions and method requirements. While composing traits, method conflicts may arise. A class is specified by composing a superclass with a set of traits and some *glue methods*. Glue methods are defined in the class and they connect the traits together; *i.e.*, they implement required trait methods (possibly by accessing state), they adapt provided trait methods, and they resolve method conflicts.

Trait composition respects the following three rules:

- Methods defined in the class take precedence over trait methods. This allows the glue methods defined in a class to override methods with the same name provided by the used traits.
- Flattening property. A non-overridden method in a trait has the same semantics as if it were implemented directly in the class using the trait.
- Composition order is irrelevant. All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

A *conflict* arises if we combine two or more traits that provide identically named methods

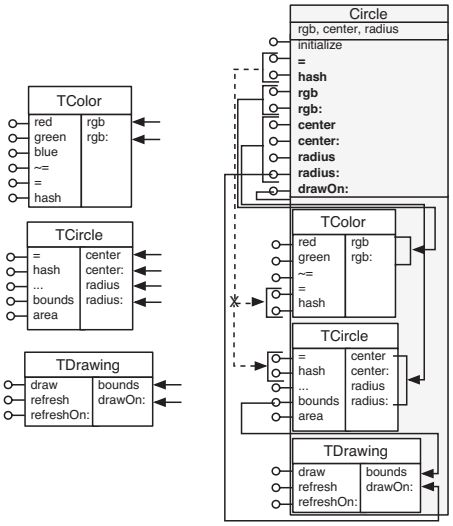

Figure 1: The class Circle is composed of three traits TColor, TCircle and TDrawing. A trait consists of a name, a list of defined methods (left-hand pane) and a list of required methods (right-hand pane). A conflict is resolved by redefining the methods hash and =.

that do not originate from the same trait. Conflicts are resolved by implementing a glue method at the level of the class that overrides the conflicting methods, or by excluding a method from all but one trait. In addition traits allow method aliasing; this makes it possible for the programmer to introduce an additional name for a method provided by a trait. The new name is used to obtain access to a method that would otherwise be unreachable because it has been overridden [4].

Example: geometrical objects. A graphical object (circle, rectangle, ...) can be decomposed into three reusable pieces of behavior: managing its color, its shape, and its rendering. Figure 1 illustrates this principle for a circle[1].

The shape of a circle is expressed by the TCircle trait, color management by the TColor trait and the rendering by TDrawing. Graphically, a trait is depicted by a box having three distinct parts. The upper part contains the name of the trait, the left part contains the methods defined, and the right part declares the list of required methods. TCircle requires four methods: center, center:, radius, and radius: and provides methods such as draw, refresh, and refreshOn:. The TColor trait requires rgb and rgb: and provides red, green, blue and some comparison methods. Finally TDrawing requires bounds and drawOn: to offer a rendering.

[1]The implementations of traits and classboxes are in Squeak therefore the source code presented in this paper uses a Smalltalk syntax.

The Circle class is a subclass of Object. It defines three instance variables (center, radius, rgb) and their accessors. Circle is composed of the three traits previously described. Because both TCircle with TColor define the hash and = methods, the composition of these two traits needs to resolve the conflicts that occur with hash and =. This is done by removing the entry of hash and = in both traits and creating new entries (colorHash, circleHash, ...) corresponding to the deleted ones.

Traits are composed explicitly at class creation time with the uses: keyword. For instance, the class Circle is defined as:

```
Object subclass: #Circle
 instanceVariableNames: 'center radius rgb'
 uses: {   TDrawing +
           TCircle @ {#circleHash → #hash . #circleEqual: → #=} +
           TColor @ {#colorHash → #hash . #colorEqual: → #=} }
```

The + operator yields a union of two traits, - removes one entry from a trait (not used in the previous example), and → copies an entry under a new name (for instance, m1 → m2 defines m1 as a new name for the m2 method).

Conflict is explicitly resolved by (re)defining at the level of the class methods hash and = using the new alias obtained from the traits composition (colorHash, circleHash, ...). Note that ↑ is the Smalltalk keyword for returning a value from a method.

```
Circle≫hash
 ↑self circleHash
   bitXor: self colorHash
```

```
Circle≫= anObject
 ↑(self circleEqual: anObject)
   and: [self colorEqual: anObject]
```

Unanticipated changes cannot be expressed with traits. A trait composition is specified when a class is created and belongs to the definition of this class. Even if Smalltalk allows classes to be recompiled at any-time, only one version of a class is present. Trait composition is invasive: all instances of the class to which the composition is applied are affected. As a consequence, traits do not help applying unanticipated changes.

3 Classboxes

Classboxes are a module system where classes can be extended with addition and redefinition of class members in a well-delimited scope. By controlling the visibility of definitions, class extensions brought by an application are confined to a delimited scope, avoiding conflicts that may occur with other applications. Several versions of a same class can coexist at the same time in the same system. Each class version corresponds to a particular view of this class. Class identity is preserved when importing a class [2].

A classbox can define classes, import classes from other classboxes, and define methods and variables on any classes visible within this classbox (*i.e.*, defined or imported classes).

A *class extension* is a method that is added/redefined or an instance variable defined on an imported class. Within a classbox, extensions defined on classes have a scope limited to (i) the classbox that defines them and to (ii) classboxes that import the new version of the class. Outside the classbox that extends a class and other classboxes that depend on it, extensions are not visible.

The notion of class extension differs from that of object-oriented specialization because the identity of the extended class is kept and no subclass is created, therefore all the clients of the extended class can benefit from these extensions.

Defining unanticipated changes. An important property of classboxes is that *local method definitions have precedence over imported ones*. This means that when a method is redefined on an imported class, any method called within the defining classbox will use the new implementation of the method [2].

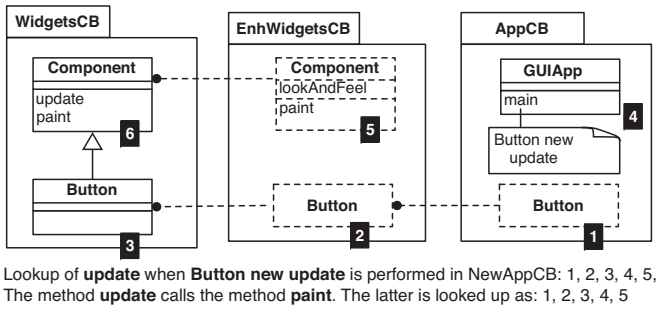

Figure 2: Locality of changes entail a new method lookup semantics. The numbers within the boxes indicate the steps taking in looking up a message sent to a button.

Figure 2 shows a classbox WidgetsCB composed of a class Component and a subclass Button. Component has two methods paint and update of which the latter invokes the former. The classbox EnhWidgetsCB imports these two classes, redefines the paint method and adds a variable lookAndFeel to Component. Button is imported without being extended. The AppCB classbox imports Button from EnhWidgetsCB and defines the class GUIApp that instantiates the imported Button and triggers the update on it. The classbox WidgetsCB, defining a base system, is not impacted by the extension of EnhWidgetsCB. Moreover, clients of WidgetsCB (not shown on the figure) are not impacted by these extensions.

A base system can be modified by extending the classes it is composed of. These modifications are confined to the classboxes that defines them and to other classboxes that use these modifications. This means that the original system and its clients are not impacted by these modifications because they are not visible outside the defining classboxes.

New method lookup semantics. Figure 2 illustrates the lookup of messages update and

paint. When the message update is sent to an instance of Button in the classbox AppCB, the lookup algorithm first searches for the implementation of update in the classbox AppCB (1). This method is not defined in this classbox, therefore the lookup follows the chain of import (2). In EnhWidgetsCB, update is not defined, so the lookup continues in WidgetsCB (3). In this classbox, the class Button is not imported anymore but defined in it. Therefore, update is looked up in the superclass Component but starting from the source classbox (AppCB, in step 4). Because Component is not visible within AppCB and Button is imported from EnhWidgetsCB, the lookup continues to EnhWidgetsCB (5). The class Component is visible, but the method update is not implemented. Finally the method is found in WidgetsCB. The method update triggers the message paint. In a similar way, the method paint is looked up following steps 1 through 5.

Class extensions cannot be reused. Within a classbox, extensions defined on an imported class cannot be applied to other classes. Class extensions cannot be reused for classes other than the one they are applied to.

4 A Traits and Classboxes Symbiosis

In this section, we present a model resulting from a symbiosis between traits and classboxes. Classboxes are refined with the ability for a class to be extended by locally using a trait. Within a well-delimited scope, a set of traits can be used by a set of classes without being specified in the definitions of these classes. The goal of this symbiosis is to offer a better support for the introduction of unanticipated changes, in particular crosscutting collaborations involving multiple classes.

We enhanced the notion of class extension with two new constructs: import of traits (Section 4.1) and extending a class by making it use a trait (Section 4.2). Finally, Section 4.3 is dedicated to the scope of class extension visibility.

4.1 Import of Traits

A trait, like a class, belongs to one and only one classbox. A classbox defines a namespace where no more than one trait or a class can be bound to a name. Several traits having the same name cannot be simultaneously visible in a given classbox, but can live in different classboxes.

There can be an *import* relationship between two classboxes. Syntactically, an import is described as ColoredWidgetsCB import: #TColor from: ColorCB meaning that the classbox ColoredWidgetsCB imports the trait TColor from the classbox ColorCB. This operation makes TColor visible in ColoredWidgetsCB: TColor can be used by any classes defined or imported within ColoredWidgetsCB. Note that classes, as well as traits, can be imported. Classes that are imported or defined can be extended by using traits, which are themselves defined or imported.

4.2 Class Extension

We refine the notion of class extension by making local trait use possible. Figure 3 shows a situation where a class Circle imported from a classbox WidgetsCB by a classbox ColoredWidgetsCB is visible within ColoredWidgetsCB. Because Circle is visible within ColoredWidgetsCB, Circle can be extended in this classbox with variable additions, methods additions/redefinitions, and trait uses. An imported class can be extended by: (i) adding new methods, (ii) redefining some of its imported methods, (iii) adding new instance variables, and (iv) using one or more traits.

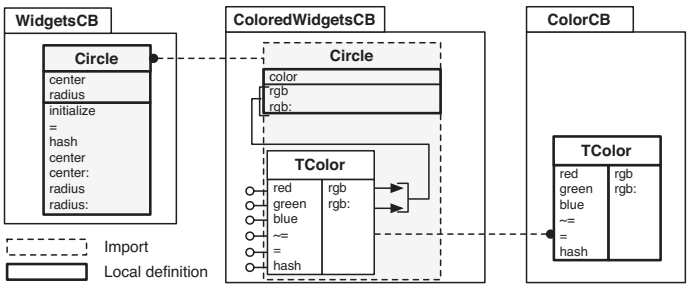

Figure 3: The class Circle is imported in ColoredWidgetsCB and extended by using the imported trait TColor.

The class Circle is defined within the classbox WidgetsCB and the trait TColor is defined within ColorCB. The classbox ColoredWidgetsCB imports Circle from WidgetsCB and TColor from ColorCB. Circle is extended with a new variable color and two methods rgb and rgb: that access color. Circle uses the imported trait. Instances of Circle understand the methods defined in TColor within ColoredWidgetsCB and other classboxes that import Circle from ColoredWidgetsCB (not shown on the figure).

The use of TColor by Circle is possible because Circle defines the two methods rgb and rgb: required by TColor. The classbox ColoredWidgetsCB is defined as follows:

```
Classbox named: #ColoredWidgetsCB.
ColoredWidgetsCB import: #Circle from: WidgetsCB.
ColoredWidgetsCB addVariableNamed: #color to: #Circle.

Circle>>rgb
      ↑color
Circle>>rgb: aColor
      color := aColor

"Import the trait TColor"
ColoredWidgetsCB import: #TColor from: ColorCB.
Circle use: {TColor}.
```

The classbox ColorCB defines the trait TColor as follows:

```
Classbox named: #ColorCB.
Trait named: #TColor.
TColor>>red
        ...
TColor>>rgb
        self requirement
TColor>>rgb: aNumber
        self requirement
```

Use of traits has a visibility limited (i) to the scope of the classbox ColoredWidgetsCB that establishes this *use* relationship between a class and a trait, and (ii) to classboxes that import the class Circle from ColoredWidgetsCB.

4.3 Visibility of Extensions

Classboxes allow the visibility of definitions to be bound to a well-delimited scope. Contrary to AspectJ [8] (with inter-type) and Multijava [3, 9] (with open-class) where visibility of extensions is global, with classboxes extensions are local to the classbox that defines them and to other classboxes that import the extended classes. By making the relationship between a class and trait an extension, the visibility of using a trait is bound to the classbox that defines this relationship.

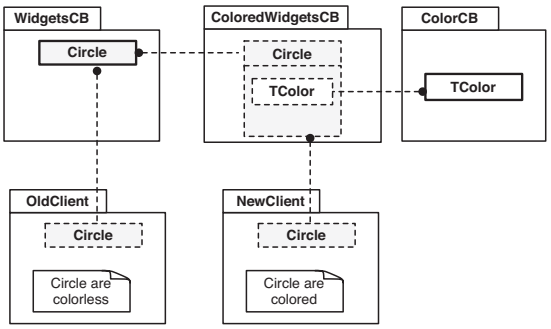

Figure 4: Two versions of Circle coexist at the same time. From the point of view of OldClient circles are colorless, and from the point of view of NewClient they are colored.

Figure 4 shows an example of two clients relying on two different versions of the class Circle. Within the classbox OldClient, circles are colorless, and within NewClient circles are colored because within this classbox the class Circle uses the trait TColor.

A base system can be modified by a set of classboxes. By combining traits with classboxes, class extensions can be applied several times (*i.e.*, to different classes) via traits. This combination allows unanticipated changes to be applied to several places in the system

without impacting clients that rely on the original version of the system. This combination allows us to model an architecture based on collaboration [6] without the limitations of current implementations.

5 Cross-cutting Collaborations and Unanticipated Modifications

Collaborations have been introduced to describe functionalities that cross-cut several classes [6, 10]. In a collaboration, a *role* is a set of features intended to be applied to a class and a *collaboration* is a set of roles. A collaboration is expressed with a combination of traits and classboxes by regarding a role as a trait and a collaboration as a classbox.

5.1 Example of Architecture in Collaboration

To stress the expressive power of the traits and classboxes combination, we implemented a graph traversal application. This application was initially presented by Holland [6], and then in several others like in VanHilst and Notkin [12] and Smaragdakis [10]). This graph traversal application is the canonical example to illustrate a collaboration-based achitecture.

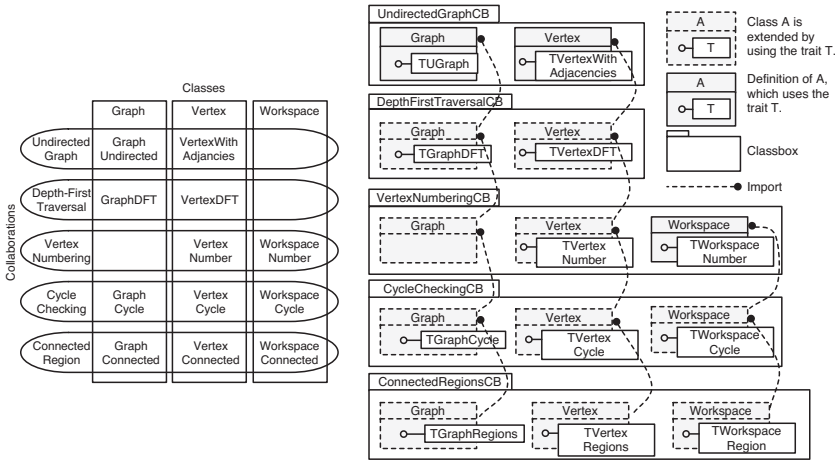

Figure 5: Decomposing into collaborations. Left-hand side: ovals represent collaborations, rectangles refer to classes, and at their intersection the squares refer to the roles. Right-hand side: classboxes represent the collaborations and traits represent the roles

Holland's example defines three operators (*i.e.*, algorithms) applied to an undirected graph, based on a depth-first traversal. These operators are: (i) *vertex numbering* numbers all nodes in the graph in depth-first order, (ii) *cycle checking* examines whether the graph is

cyclic, and (iii) *Connected Regions* classifies graph nodes into connected graph regions. The application itself consists of three classes: Graph defines a graph as a container of nodes, Vertex defines some properties of a node, and Workspace contains some global variables specific to each graph operation. For instance, the *workspace* object for a *vertex numbering* operation holds the value of the last number assigned to a vertex.

This application is decomposed into five distinct collaborations: (i) *undirected graph* encapsulates properties of an undirected graph, (ii) *depth-first traversal* encapsulates the features of depth first traversals and provides an interface for extending traversals, (iii) *vertex numbering* numbers the set of nodes, (iv) *cycle checking* checks whether cycles in the graph are present, and (v) *connected region* classifies nodes into distinct connected graph regions.

5.2 Collaboration with Traits and Classboxes

A collaboration is represented by a classbox, and a role by a trait. Figure 5 depicts the graph traversal application expressed with collaborations based on traits and classboxes. This application consists of five classboxes representing collaborations, each of these defining traits representing roles. These traits are used by imported or defined classes. For the sake of keeping the figure clear, Figure 5 shows class and trait definitions and class imports only.

The first classbox UndirectedGraphCB defines classes Graph and Vertex. It defines properties of undirected graphs. These properties are implemented in the traits TUGraph and TVertexWithAdjacencies. The former is used by the class Graph and the latter by the class Vertex. The second classbox DepthFirstTraversalCB defines a deep-first traversal algorithm with the two traits TGraphDFT and TVertexDFT. The two classes Graph and Vertex are imported from UndirectedGraphCB and extended by using these two traits. The classbox VertexNumberingCB imports the two classes extended by the previous classboxes and defines a class Workspace, and the traits TVertexNumber and TWorkspaceNumber used by Vertex and Workspace, respectively. The collaboration *VertexNumbering* does not define any role for the class Graph, therefore it is imported without being extended. The two other classboxes CycleCheckingCB and ConnectedRegionsCB are built in a similar way.

Figure 6 shows refinements defined by the collaboration *VertexNumbering* on *DepthFirstTraversal*. The role *VertexNumber*, modeled by the trait TVertexNumber, is used by the imported class Vertex. This class is extended with two instance variables number and workspace and fours methods representing the variable's accessors and mutators. The trait TVertexNumber defines a method compute: and accessing these variables through some accessors is needed.

The version of Vertex from DepthFirstTraversalCB contains a method compute:. Within VertexNumberingCB this method has to be redefined by the trait TVertexNumber. As shown below, the previous implementation provided by DepthFirstTraversalCB has to be accessible to VertexNumberingCB. Therefore the method compute: provided by DepthFirstTraversalCB is renamed as dftCompute: then the entry compute: is removed before

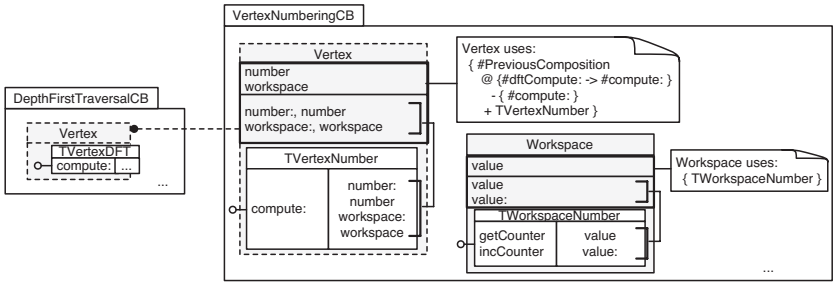

Figure 6: Definition of the collaboration *VertexNumbering*.

performing an union with the trait TVertexNumber.

Trait applications are incrementally defined: each classbox can make an imported class use a new trait. To make this possible, a trait composition of a class has to be obtained to be incrementally modified. This is achieved using the variable PreviousComposition. PreviousComposition in a trait composition clause refers to the composition of the imported class. The alias mechanism of traits (@ { #dftCompute: -> #compute:}) makes the compute: method defined by TVertexDFT accessible through a new name dftCompute:. Then - { #compute: } removes the entry from the composition in oder to not raise conflicts with + TVertexNumber.

The method compute: defined by TVertexNumber refers to method dftCompute: which corresponds to the renamed compute: of TVertexDFT:

```
TVertexNumber≫compute: aBlock
    number := workspace value.          "A new number to the Vertex"
    workspace incCounter.               "The counter is incremented"
    self dftCompute: aBlock             "Call of the previous implementation of compute:"
```

The classbox VertexNumberingCB defines the class Workspace which contains an instance variable value and two accessors value and value:. This class uses the traits TWorkspaceNumber defined within this classbox, specifying a trivial composition. TWorkspaceNumber requires the mutator and accessor of value.

```
TWorkspaceNumber≫getCounter
    self value ifNil: [ self value: 0 ].
    ↑self value
```

```
TWorkspaceNumber≫incCounter
    self value: (self value + 1)
```

5.3 Gain with Traits and Classboxes

Identity of participant preserved. Defining a role with a *mixin* uses subclassing [10]. Using subclassing, an unanticipated change cannot be applied without modifying former clients [5] because the clients have to reference subclasses to benefit from extensions. Using traits and classboxes allows one to define extensions while preserving the identity of the extended classes. As a result, applying a role to a class is done without subclassing, therefore the identity is preserved.

A collaboration as an unanticipated change. *Mixins* fit well for an architecture well-defined in advance. However, they do not help in refining an application with changes that were not initially planned. With our model, a new collaboration layer can be added to an existing application by preserving former clients from being impacted. As a result, a collaboration defines a unanticipated changes. Moreover such a collaboration (defined as a classbox) can apply a role (defined as a trait) to several classes.

Bounded visibility of extension. Because some of the clients rely on the original version of a system, not all of the clients have to be impacted when a change is applied to a system. This is why it is necessary to control the propagation of a change when applied to a system. With classboxes extensions are bound to a scope limited to the classbox that defines these extensions and to other classboxes that import the extended classes.

Coherence is achieved by trait method requirements. Having a coherent architecture of various collaborations is a crucial problem already tackled by Batory [1] and Steyaert [11]. Batory *et al*. suggested to define properties that are propagated along the collaboration architecture. The coherence of a particular collaboration is achieved with the presence of certain properties. Steyaert *et al*. propose to use a constraint system to control the construction of inheritance hierarchies. By combining traits and classboxes, no extra mechanism needs to be added in order to achieve coherence. Coherence is achieved with the required methods that need to be fulfilled when traits are used by a class.

6 Discussion and Evaluation

Link class-trait separated from the class definition. In the original trait model, the *use* relationship between a class and the traits it uses is specified within the definition of the class [4] at class creation time. With the combination of traits and classboxes, the responsibility of using a trait is not only offered to the class creator but also to programmer needing to extend the class. Hence it is possible to change dynamically the classes and the traits they use.

Supporting incremental changes. A class can be incrementally extended by a chain of classboxes. This class is imported and refined in each classbox. In a classbox, extending

a class by making use of a trait has to be the result of a composition with this trait and the previous composition obtained from the provider classbox (*i.e.*, the classbox where the class is imported from).

In order for a class to be incrementally refined by using new traits in classboxes, the trait composition visible in the provider classboxes has to be obtained while extending a class. We added therefore a new variable PreviousComposition that refers to the trait composition stated in the provider classbox. The use of this variable is illustrated in Figure 6.

Use of traits as a class extensions. Originally, the purpose of a trait was to factor out a set of method definitions that would be reused by several classes that do not belong to the same hierarchy. Our approach extends the range of application for traits. With our model, traits can also be used to define extensions (*i.e.*, group of methods) that can be applied to different classes.

Coherent collaborative architecture. By importing and extending several classes, a classbox defines a cross-cutting change. To apply a trait to a class, the class has to provide the methods required by the trait. The required method mechanism ensures that collaborations are sorted according to the dependencies among them. For instance, in Figure 5, the collaboration VertexNumberingCB cannot be misplaced: it has to be based on DepthFirstTraversalCB because this last one defines the method compute: needed by the trait TVertexNumber (compute: is specified in the trait composition rule as shown in Figure 6). This method has to be provided by the class before applying TVertexNumber. Declaring required methods forces the collaborative architecture to be coherent. However, our model does not offer any guarantee that traits intended to collaborate with each other will behave as expected once applied. For instance, let's assume a classbox ObserverPattern defines two traits TObserver and TObservable. Applying such a collaboration to an architecture does not guarantee that an "observer" object will collaborate only with "observable" objects.

7 Conclusion

We describe an approach supporting unanticipated crosscutting changes based on a symbiosis between traits and classboxes. The implementation in Squeak [7] (an open-source Smalltalk) of the described approach and application is available at www.squeaksource.com/Collaborations. The notion of class extension offered by classboxes has been enhanced by allowing a class to locally use a trait. This is the result of separating class definition from the trait composition. Several classes can be extended with the same group of methods (*i.e.*, trait). Extensions modeled as traits are applicable multiple times (*i.e.*, to different classes), as in the original version of classboxes. Since traits are stateless, they cannot be reused to add the same instance variables to several classes.

This paper shows how this combination between traits and classboxes can be applied to express a collaborative architecture where a classbox defines a collaboration. Each collab-

oration contains a set of roles that are modeled with traits.

In conclusion, traits and classboxes are two extensions of the class module that are *simple* and *expressive*. Their combination offers an elegant model supporting unanticipated changes where class extensions can be reused for different classes within a well-defined scope.

Acknowledgment. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02) and Recast: Evolution of Object-Oriented Applications (SNF 2000-061655.00/1).

References

[1] Don Batory and Bart J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering (special issue on Software Reuse)*, pages 62–87, February 1997.

[2] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems and Structures*, 31(3-4):107–126, May 2005.

[3] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, 2000.

[4] Stéphane Ducasse, Nathanael Schärli, Oscar Nierstrasz, Roel Wuyts, and Andrew Black. Traits: A mechanism for fine-grained reuse. *Transactions on Programming Languages and Systems*, 2005. under revision.

[5] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 94–104. ACM Press, 1998.

[6] Ian M. Holland. Specifying reusable components using contracts. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, volume 615 of *LNCS*, pages 287–308, Utrecht, the Netherlands, June 1992. Springer-Verlag.

[7] 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. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326. ACM Press, November 1997.

[8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceeding ECOOP 2001*, number 2072 in LNCS, pages 327–353. Springer Verlag, 2001.

[9] Todd Millstein, Mark Reay, and Craig Chambers. Relaxed multijava: balancing extensibility and modular typechecking. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 224–240. ACM Press, 2003.

[10] Yannis Smaragdakis and Don Batory. Implementing layered design with mixin layers. In Eric Jul, editor, *Proceedings ECOOP '98*, volume 1445 of *LNCS*, pages 550–570, Brussels, Belgium, July 1998.

[11] Patrick Steyaert, Wim Codenie, Theo D'Hondt, Koen De Hondt, Carine Lucas, and Marc Van Limberghen. Nested mixin-methods in agora. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 197–219, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[12] Michael VanHilst and David Notkin. Using C++ Templates to Implement Role-Based Designs. In *JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37. Springer Verlag, 1996.