# Classboxes

## Controlling Visibility of Class Extensions

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

**Alexandre Bergel**

von Frankreich

Leiter der Arbeit: Prof. Dr. S. Ducasse, Prof. Dr. O. Nierstrasz
Institut für Informatik und angewandte Mathematik

# Classboxes

## Controlling Visibility of Class Extensions

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

**Alexandre Bergel**

von Frankreich

Leiter der Arbeit: Prof. Dr. S. Ducasse, Prof. Dr. O. Nierstrasz
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 21.11.2005

Der Dekan:

Prof. Paul Messerli

# Acknowledgments

## Personal Acknowledgments

I would like to dedicate all the work I did over the last four years to my father Robert, my mother Isabela, my brothers Jérôme and Geoffrey, and to the rest of the Bergel family. Despite distances between us, you have always been beside me. Without your support and encouragements, I probably would not be able to finish what I wanted to achieve.

Recently I discovered a large branch of my family, for which I feel really close to. I thanks Maria C. Dusia Black, Dusia White and all the people I still have to meet.

I also want to thanks my dearest friends: Aska, Bachir G., Christelle V., Eric (Christophe) L., Franziska O., Gilles F., Jérôme S., Joris P., Kelly S., Lucilene G., Marcela F., Muriel M., Nicolas C., Nicolas T., Robert C., Roland B., Sonia P.. Real adventures cannot be lived without great fellows. Memorable and great actions already occur in Belgium, Canada, Czech Republic, France, Hungary, Italy, Morocco, Poland, Romania, Singapore, Spain, Sweden, Switzerland, Tunisia, UK. Is this World big enough? Answering this question requires solid, consistent, large experiments and investigations. What next? Where do we go now?

## Professional Acknowledgments

First of all, I want to thank all the members of my Ph.D. committe, Stéphane Ducasse, Oscar Nierstrasz and Roel Wuyts. Thanks Stéphane for having trusted me from the very first time I met you. Oscar, thanks from your excellent professional support. Roel, thanks for always been extremely rigorous and accurate. Stéph, Oscar and Roel, it was a real pleasure to work with you. I learnt so many things...

Then I would like to thank all former and current members of the Software Composition Group. Gabriela Arévalo, Juan Carlos Cruz, Marcus Denker, Markus Gälli, Tudor Girba, Orla Greevy, Adrian Lienhard, Laura Ponisio, Matthias Rieger, Nathanael Schärli, and Therese Schmid.

Many thanks also to all of the colleagues who have helped me and collaborated with me during my work on classboxes. I especially thanks Gilad Bracha, William Cook, Erik Ernst, Günther Kniesel, Eric Tanter, Dave Thomas, Klaus D. Witzel.

*Alexandre Bergel*
*November 2005*

# Abstract

Unanticipated changes to complex software systems can introduce anomalies such as duplicated code, suboptimal inheritance relationships and a proliferation of run-time downcasts. Refactoring to eliminate these anomalies may not be an option, at least in certain stages of software evolution.

A *class extension* is a method that is defined in a module, but whose class is defined elsewhere. Class extensions offer a convenient way to incrementally modify existing classes when subclassing is inappropriate. Unfortunately existing approaches suffer from various limitations. Either class extensions have a global impact, with possibly negative effects for unexpected clients, or they have a purely local impact, with negative results for collaborating clients. Furthermore, conflicting class extensions are either disallowed, or resolved by linearization, with subsequent negative effects.

To solve these problems we present *classboxes*, a module system for object-oriented languages that provides for behavior refinement (*i.e.* method addition and replacement). Moreover, the changes made by a classbox are only visible to that classbox (or classboxes that import it), a feature we call *local rebinding*.

We present an experimental validation in which we apply the classbox model to both dynamically and statically typed programming languages. We used classboxes to refactor part of the Java Swing library, and we show two extensions built on top of classboxes which are (i) runtime adaptation with dynamically classboxes and (ii) expressing crosscutting changes.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

It is well-established that object-oriented programming languages gain a great deal of their power and expressiveness from their support for the *open/closed principle* [63]: classes are *closed* in the sense that they can be instantiated, but they are also *open* to incremental modification by inheritance.

Nevertheless, classes and inheritance alone are not adequate for expressing many useful forms of incremental change. For example, modern object-oriented languages introduce *modules* or *packages* as a complementary mechanism to structure classes and control visibility of names. Reflection is another example of an increasingly mainstream technique used to modify and adapt behaviour at run-time. Aspect-oriented programming, on the other hand, is a technique to adapt sets of related classes by introducing code that addresses cross-cutting concerns.

Systems written in a modular way are closed in the sense that they can be executed, but they are open for unanticipated extensions that add, refine, or replace modules or whole subsystems. Module and class systems have evolved to meet the demand of reusable software components and defining incremental changes. However the range of supported extensions is often limited because subclassing is not expressive enough to refine code. In the literature this limitation is referenced under the *extensibility problem* [40]. System extensions implemented with class inheritance introduce duplicated code and type-checking issues.

## 1.1 Hypothesis: Class Extension to Support Unanticipated Changes

A module is regarded as a closed entity containing class definitions. Incremental changes are enabled by adding or replacing modules. However to regard a module as closed hampers extensibility in many ways. For instance, cross-cutting changes cannot be modeled using classical mechanisms like Java packages.

In this thesis we focus on a particular technique, known as *class extensions*, which addresses the need to extend existing classes with new behaviour. Smalltalk [43], CLOS [74], Objective-C [75], and more recently MultiJava [28] and AspectJ [49] are examples of languages that support class extensions.

A *class extension* is a set of class members (*e.g.* variable and methods) defined in a modular unit that is distinct from the one that defines the class to which these class members are related. A classical

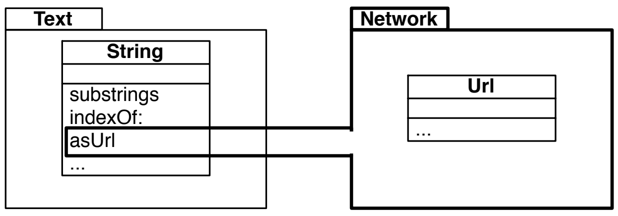


Figure 1.1: The package Network extends the class String defined in the package Text with a new method asUrl

example of a class extension present in most of Smalltalk libraries is a method that converts a string into a url. One natural and concise way of defining such a method is on the class String. This is illustrated in Figure 1.1. The package Text defines the class String which contains methods like substrings and indexOf:. The package Network defines network-related classes and extends the class String with a method asUrl. As a result, a url can be obtained from a string by directly sending a message asUrl to it, as in the Smalltalk expression 'http://www.iam.unibe.ch/∼bergel' asUrl.

Class extensions offer a good solution to the dilemma that arises when a developer would like to modify or extend the behaviour of an existing class. In such a case, subclassing is often inappropriate because that *specific* class is referred to by existing clients (and the source code of the class in question cannot be modified). But a class extension can then be applied to that specific class.

## 1.2 Understanding the Problem

Despite the demonstrated utility of class extensions, a number of open problems have limited their widespread acceptance. Briefly, these problems are:

1. *Globality.* In existing approaches, the effects of a class extension are either global (*i.e.*, visible to all clients), or purely local (*i.e.*, only to specific clients named in the application of the class extension). In the first case, clients that do not require the class extension may be affected. In the second case, *collaborating clients* that are not explicitly named will not see the class extension.

2. *Conflicts.* If two or more class extensions attempt to extend the same class, this may lead to a conflict. In existing approaches, conflicts are either forbidden, or extensions are linearized. This may lead to unexpected behaviour. In either case, the class extension utility is severely impacted.

## 1.3 Our proposal: Classboxes

We propose a modular approach to class extensions that solves the two problems outlined above by defining an implicit context in which class extensions are visible. We introduce the notion of a *classbox* which acts as a kind of module with three main characteristics:

- It is a *unit of scoping* in which classes, global variables and methods are defined. Each entity belongs to precisely one classbox, namely the one in which it is first *defined*, but a class can be made visible to other classboxes by means of an *import* mechanism. Importing from a classbox a class from another classbox makes this class visible. Methods can be defined for any class visible within a classbox, independently of whether that class is defined or imported. Methods defined (or redefined) for imported classes are called *class extensions*.

- A class extension is *locally visible* from the perspective of the classbox in which it is defined. This means that the extension is only visible in (i) the extending classbox, and (ii) other classboxes that directly or indirectly import the extended class.

- A class extension supports *local rebinding*. This means that, although extensions are locally visible, their effects extend to all collaborating classes. A classbox thereby determines a namespace *within* which local class extensions behave *as though they were global*.

**Thesis statement.** A scoping mechanism is necessary to efficiently control visibility of changes by means of class extensions.

## 1.4 Contributions

The main contributions of this thesis are summarized as follows and have been published as shown by the references:

- *First-class environment module calculus* – Understanding the multitude of module systems requires a common foundation in which differences between various semantics are expressed. We define a module calculus for this purpose. Because the notion of namespace is implicitly associated to module, this calculus makes the notion of environment a first-class entity [12].

- *Analysis of a large library* – An analysis of a large and widely used Java library (Swing) is used to define criteria for a better mechanism to deal with changes. This analysis points out code duplications, broken inheritance and explicit type checking in this Java library, which justifies the need of having at the language level constructs to express changes and how they can be applied [13].

- *Scoped class extensions* – Scoping facilities to deal with changes by means of class extensions are provided by means of a new module system, classboxes [11, 15, 67].

- *Strategies to efficiently implement scoped class extensions* – A description of two implementations of classboxes in the Smalltalk dialect Squeak is proposed. The first implementation implies a modification of the virtual machine, and the second is based on bytecode manipulation. Benchmarks are provided for each of these implementations [16, 14].

- *Dynamic classboxes* – Uniform and expressive mechanism to support crosscutting changes made of class extensions [10].

## 1.5 Thesis Outline

This dissertation is structured as follows: In Chapter 2, we present a detailed state of the art based on an analysis of different module systems. Several properties are formalized according to features of various module systems. These properties are then used to establish a taxonomy. In Chapter 3, we identify and illustrate issues with classical object models when dealing with incremental changes. An analysis of the Java Swing library is presented and we point out some anomalies and inconsistencies. This defines the focus of Chapter 4. We introduce classboxes by means of a formal model. We evaluate the classbox model against the problems identified in Chapter 3: we refactor Swing using classboxes, and anomalies in the original version are removed. In Chapter 5, we describe two implementations of classboxes in the Smalltalk dialect Squeak: the first one based on a modification of the virtual machine, and the second one based on bytecode manipulation. In Chapters 6, 7, and 8 we present three extensions of classboxes: expressing runtime adaptation with dynamic classboxes, symbiosis with traits and classboxes, and classboxes in a statically typed environment. In Chapter 9, we conclude by summarizing how the main results of our work support the statement of the thesis, and we outline areas of future work related to classboxes.

# Chapter 2

# Language and Module Constructs to Support Changes

"Software maintenance" is an activity consisting of modifying a software after it has been delivered. Sommerville [87] and Davis [30] estimate that the cost of software maintenance accounts for 50% to 75% of the overall cost of a software system. Supporting unanticipated changes is probably one of the most challenging task in software engineering. This issue can be tackled using *forward engineering* and *reverse engineering* techniques. This thesis focuses on the former by offering new language constructs to ease software maintenance and evolution. This chapter analyses the existing approaches in three different fields related to supporting unanticipated changes: module system supporting class refinement, packaging and deploying static changes, and dynamic application of crosscutting changes.

*Module systems supporting class refinement.* Each object-oriented programming language proposes various grouping mechanisms to bundle interacting classes (*i.e.* packages, modules, selector namespaces, etc). To understand such diversity and to compare the different approaches, a common foundation is needed. As far as we are aware of, no major attempt in that direction has been realized to date.

Section 2.1 presents a simple module calculus together with a set of operators for modeling the composition semantics of different grouping mechanisms. Using this module calculus we are able to express the semantics of Java packages [5], C# namespaces [24], Ruby modules [92], selector namespaces [97], gbeta classes [37], MZScheme units [41], and MixJuice modules [45]. This calculus supports the identification of system families sharing similar characteristics. In addition it provides a uniform way to represent and analyze fine-grained module semantics.

*Packaging and deploying static changes.* Section 2.2 presents various approaches to package and deploy system changes. These changes are qualified as static because they are applied at compile time. The section studies static changes based on (i) class extension, (ii) module systems, (iii) mechanism enhancing or completing class inheritance, and (iv) aspect oriented programming.

*Dynamic application of crosscutting changes.* Section 2.3 summarizes the existing approaches in dynamic application of crosscutting changes. Mainly two approaches are studied: dynamic code adaption and dynamic extension and replacement of classes.

## 2.1 Module systems supporting class refinement

Object-oriented languages support the construction of applications based on sets of interacting classes. Classes, methods and global definitions are then grouped together as packages or modules for deployment reasons or for delimiting abstraction layers. Unfortunately, though the intent behind packages and modules is clear, their semantics often is not. The simple fact that the terms *module* and *package* are overloaded with different semantics reveals a larger problem: the diversity of grouping mechanisms hampers their comparison and understanding.

Modeling modules by means of a calculus has always been an active research topic (*e.g.* [3, 17, 21, 55, 62]). These calculi are aimed at tackling various issues with modules like mutual recursion and higher order features [3] or to model class and mixins application in a typed setting [17]. In this section we focus on expressing various module operators semantics obtained from programming languages like Java, C#, MZScheme, Modular Smalltalk, and gbeta under a single module calculus.

**Summary.** Classical module systems, like those of Modula-2 [98], Modula-3 [27], Oberon-2 [69], Ada [90], Java, C++, and Eiffel [64] do not support the notion of class extensions (*i.e.* the fact that a method can be added or overidden from a package different from the one that defines the class). However, class extensions are widely used in the languages that support it, such as Smalltalk [97], CLOS [47] and gbeta [37]. OpenClasses [28], Keris [100] and MixJuice [45] offer packaging systems that introduce class extensions, virtual classes and other new features to packages.

Other languages such as Ruby [92] and Unit [41] support the definition and application of mixins to modules at different levels. Languages such as VisualWorks Smalltalk [96] totally decouple the issues of namespaces from those of code packaging, hence a package in VisualWorks does not provide any support for scoping of names.

**Contributions.** In this section we introduce a simple calculus of modules, together with a set of operators designed to express various encapsulation policies, composition rules, and extensibility mechanisms. The contributions of this section are:

- A *formalism* in which the semantics of different module systems can be expressed,
- The identification of a set of *properties* useful to characterize different languages, and
- A *taxonomy* of different module systems.

The approach presented in this section enables a language designer to compare module systems for various OOP languages.

**Structure of the section.** In Section 2.1.1 we define the calculus and its operators. In Sections Section 2.1.2 through Section 2.1.8 we use the calculus to model Java packages, C# namespaces, Ruby modules, selector namespaces, gbeta virtual classes, MZScheme units and MixJuice modules. We chose Java and C# as they are mainstream languages, Ruby as it defines the notion of module mixin, Modular Smalltalk [97] and Smallscript [85] as they define changes that crosscut classes with selector namespaces, Beta [53] as it introduces the notion of virtual classes, MZScheme unit [40] as it separates the unit definition from the dependancy statements, and MixJuice [45] as it constrains only one class version to be present in the system.

In Section 2.1.9 we develop a taxonomy to characterize the studied module systems according to the set of properties we modeled. In Section 2.1.10 we describe related work. In Section 2.1.11 we summarize the results obtained and outlining future work regarding the module calculus.

### 2.1.1 A Simple Module Calculus

The module calculus we propose makes use of the primitive notion of an *environment*. We first define environments and the basic operators for manipulating them. Then we define modules as abstractions over environments, and we propose further operators for composing and manipulating modules.

Note that we have implemented all the formulas described below in DrScheme [33]. The source code, accompanied with unit tests, is in the Annex A.

**Environments.** The following definitions introduce environments and some basic operators.

**Definition 1 environment**: an *environment* $\epsilon : D \rightarrow R^\star$ , is a mapping from some domain $D$ to an extended range $R^\star = R \cup \{\bot\}$, such that the inverse image $\epsilon^{-1}(R)$ is finite. The set of environments is $\mathcal{E}$ and we assume it to be a subset of $R$.

We will represent environments as finite sets of mappings, for example:

$$\epsilon_1 = \{a \mapsto x, b \mapsto y\}$$

is an environment that maps $a$ to $x$ and $b$ to $y$. All other values in the domain of this environment (let's say, $c$) are mapped to $\bot$.

We will normally leave out unessential parentheses. Since an environment is a function, we simply invoke it to look up a binding. In this case, $\epsilon_1\ a = x$, $\epsilon_1\ b = y$ and $\epsilon_1\ c = \bot$.

**Definition 2 keys**: The set of keys of an environment $\epsilon : D \rightarrow R^\star$ is $\kappa\ \epsilon$:

$$\kappa\ \epsilon \overset{\text{def}}{=} \epsilon^{-1}(R)$$

For instance, $\kappa\ \epsilon_1 = \{a, b\}$.

**Definition 3 override**: An environment $\epsilon : D \rightarrow R^\star$ may *override* another environment $\epsilon'$. We define $\epsilon \rhd \epsilon' : D \rightarrow R^\star$ as follows:

$$(\epsilon \rhd \epsilon')\ x \overset{\text{def}}{=} \begin{cases} \epsilon'\ x & \text{if } \epsilon\ x = \bot \\ \epsilon\ x & \text{otherwise} \end{cases}$$

For example, if $\epsilon_2 = \{b \mapsto z, c \mapsto w\}$, then $(\epsilon_1 \rhd \epsilon_2)\ a = x$, $(\epsilon_1 \rhd \epsilon_2)\ b = y$, and $(\epsilon_1 \rhd \epsilon_2)\ c = w$.

**Definition 4 extend**: An environment $\epsilon : D \rightarrow R^\star$ such that $\epsilon\ a = \bot$ may be *extended* to produce a new environment $\epsilon \| \{a \mapsto x\}$ containing all the mappings of $\epsilon$ plus $a \mapsto x$.

$$\epsilon \| \{a \mapsto x\} \overset{\text{def}}{=} \begin{cases} \epsilon \rhd \{a \mapsto x\} & \text{if } \epsilon\ a = \bot \\ \bot & \text{otherwise} \end{cases}$$

Note that the $\|$ operation does not allow an entry to be added if the key is already present.

**Definition 5 exclusion**: Given an environment $\epsilon : D \rightarrow R^\star$ and a key $a$, the *exclusion* $\epsilon \backslash a$ is:

$$(\epsilon \backslash a)\ x \overset{\text{def}}{=} \begin{cases} \epsilon\ x & \text{if } x \neq a \\ \bot & \text{otherwise} \end{cases}$$

Exclusion simply removes any binding present for the given key.

**Modules.** We define modules as abstraction built over environments. Modules are composed and manipulated using operators

**Definition 6 Module**: A *module* $m : \mathcal{E} \rightarrow \mathcal{E}^\star$ , is a mapping from an environment to an environment. We denote $\mathcal{M}$ the set of modules.

**Example.** We represent modules as functions taking an environment and returning an environment. An example of two modules $m_1$ and $m_2$:

$$\begin{aligned} m_1 &= \lambda\text{self}.\ \{a \mapsto 1, b \mapsto 2\} \\ m_2 &= \lambda\text{self}.\ \{a \mapsto 3, b \mapsto \text{self}\ a\} \end{aligned}$$

As we see in $m_2$, the $\epsilon$ parameter makes it possible for entries in a module to look up other bindings in the parameter environment (*i.e.* self). Shortly we will see how a module can be *instantiated* to an environment by taking its fixpoint with the *fix* operator. Therefore, an instantiated module can look up bindings in itself.

We overload the operators previously introduced, and from the context it is always clear whether we mean the environment or module operator.

**Definition 7 extend**: A module can be extended with a new mapping using the $\|$ operation:

$$m \| \{a \mapsto x\} \overset{\text{def}}{=} \begin{cases} \lambda\text{self}.\ (m\ \text{self})\ \|\ \{a \mapsto x\} & \text{if } (m\ \text{self})\ a = \bot \\ \bot & \text{otherwise} \end{cases}$$

**Definition 8 override**: Two modules $m$ and $m'$ can be merged to produce a third module $m \ \rhd\ m'$.

$$m\ \rhd\ m' \ \overset{\text{def}}{=}\ \lambda\text{self}.\ (m\ \text{self})\ \rhd\ (m'\ \text{self})$$

Note that this operator is associative but not symmetric.

**Definition 9 exclusion**: Exclusion on modules is expressed using $\backslash$:

$$m \backslash x \ \overset{\text{def}}{=}\ \lambda\text{self}.\ (m\ \text{self}) \backslash x$$

Restricting a key x on a module removes x from the environment that defines this module.

**Module Encapsulation Operators.** Module encapsulation is articulated by operators manipulating the set of keys visible from outside the module. We call this set the module's interface. Three operators are involved: *fix* is used to instantiate a module, $\kappa$ to get all the mapping names (keys) of a module, and *hide* to remove one mapping from a module interface.

**Definition 10 fix**: A module $m : \mathcal{E} \rightarrow \mathcal{E}^\star$ can be *instantiated* to an environment as follows:

$$\mathit{fix}\; m \overset{\text{def}}{=} \mu\epsilon.m\epsilon$$

(We assume the usual definition of $\mu$, where $\mu x.e$ reduces to $e[\mu x.e/x]$, so $\mathit{fix}\; m = m(\mathit{fix}\; m)$.)

Since $m$ is a function from environment to environment, $\mathit{fix}\; m$ represents a fixpoint in which all the mappings provided by the module are made available to each other. The use of fixpoints to model self-references in object-oriented languages was introduced by Cardelli [26].

**Example.** For instance, for the two modules $m_1 = \lambda\mathsf{self}.\ \{a \mapsto 1, b \mapsto \mathsf{self}\}$ and $m_2 = \lambda\mathsf{self}.\ \{a \mapsto 2\}$ we have:

$$\begin{array}{lcl} (\mathit{fix}\; (m_2 \triangleright \quad m_1 \quad))\; b\; a & = & 2 \\ (\mathit{fix}\; (m_2 \triangleright \; \lambda\mathsf{self}.\ \{b \mapsto (\mathit{fix}\; m_1)\; b\}\; ))\; b\; a & = & 1 \end{array}$$

In the first example, let $\Phi = \mathit{fix}(m_2 \triangleright m_1)$. By the fixpoint operator, this gives $\Phi = \{a \mapsto 2\} \triangleright \{a \mapsto 1, b \mapsto \Phi\}$, so $\Phi\; b\; a = \Phi\; a = 2$. Since $m_1$ and $m_2$ are effectively merged, $m_2$'s binding of $a$ becomes visible within $m_1$.

In the second example, let $\Phi_1 = (\mathit{fix}\; m_1) = \{a \mapsto 1, b \mapsto \Phi_1\}$ and $\Phi_2 = \mathit{fix}(m_2 \triangleright \lambda\mathsf{self}.\ \{b \mapsto \Phi_1\; b\}) = \{a \mapsto 2\} \triangleright \{b \mapsto \Phi_1\}$. So $\Phi_2\; b\; a = \Phi_1\; a = 1$. Here $m_1$ is effectively closed, so merging it with $m_2$ has no effect on the binding of $a$.

**Definition 11 keys**: The set of keys of a module $m$ is defined as:

$$\kappa\; m \overset{\text{def}}{=} \kappa\; (\mathit{fix}\; m)$$

**Definition 12 hide**: Given a module $m : \mathcal{E} \rightarrow \mathcal{E}^\star$ , a binding to the key *a* can be removed from its interface using the *hide* operation:

$$\begin{array}{rcl} \mathit{hide}\; a & \overset{\text{def}}{=} & \lambda m.\ \lambda\mathsf{self}.\ m\backslash a\; (\{a \mapsto (\mathit{fix}\; m)\; a\} \triangleright \mathsf{self}) \\ \mathit{hide}\; \{x_1, x_2, \ldots, x_n\} & \overset{\text{def}}{=} & (\mathit{hide}\; x_1)(\mathit{hide}\; \{x_2, \ldots, x_n\}) \end{array}$$

**Example.** Hiding a binding of a module removes the entry from the module's interface, but the binding's value is still accessible through other bindings. For example, for the two modules $m_1 = \lambda\mathsf{self}.\ \{a \mapsto 1, b \mapsto \mathsf{self}\}$ and $m_2 = \lambda\mathsf{self}.\ \{a \mapsto 2\}$, we have:

$$\begin{array}{lcl} (\mathit{fix}\; (\mathit{hide}\; a)(m_1))\; a & = & \perp \\ (\mathit{fix}\; (\mathit{hide}\; a)(m_1))\; b\; a & = & 1 \end{array}$$

**Class Definition.** Within a module, a value associated to a key represents a *definition*. If the calculus is intended to express the semantics of a module system which is part of a procedural (or functional) language, values of bindings describe functions [2, 55]. In the rest of this section we will focus only on expressing semantics of module systems for class-based object-oriented programming languages. Definitions contained within a module describe classes. A class is represented as an environment whose mappings define state and methods.

We define the following domains:

- The set of modules is represented by $\mathcal{M}$.
- The set of class names by $\mathcal{C}$. It represents keys of the mappings defining a module.
- The set of definitions defining the state and behavior of a class is denoted by $\mathcal{D}$.

**Example.** For instance a module containing $Point$ and $PointFactory$ classes can be defined as:

$$
\begin{array}{l}
\textit{GraphicsModule} = \lambda\epsilon.\ \{ \\
\quad \textit{Point} \mapsto \{ \\
\qquad\qquad x \mapsto 0, \\
\qquad\qquad y \mapsto 0, \\
\qquad \textit{moveBy} \mapsto \lambda dx.\ \lambda dy.\ \lambda \textit{self}.\ \{ \\
\qquad\qquad\qquad x \mapsto \textit{self}\, x + dx, \\
\qquad\qquad\qquad y \mapsto \textit{self}\, y + dy\ \} \rhd \textit{self}\}, \\
\quad \textit{PointFactory} \mapsto \{\ \textit{newPoint} \mapsto \lambda \textit{self}.\ \epsilon\ \textit{Point}\ \}\ \}
\end{array}
$$

Within our calculus, inheritance over classes is expressed by the *extendClass* operator, defined hereafter.

**Definition 13** In a module *m*, a class *c* extends a superclass *sup* with a set of definitions *d* using the *extendClass* operation defined as:

$$
\begin{array}{lcl}
\textit{extendClass} & : & \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{C} \rightarrow \mathcal{D} \rightarrow \mathcal{M} \\
\textit{extendClass} & = & \lambda m.\ \lambda sup.\ \lambda c.\ \lambda d.\ \lambda\epsilon.\ m\ \epsilon \| \{c \mapsto d \rhd (m\ \epsilon\ sup)\}
\end{array}
$$

Class members are denoted by $d$. When creating a new class, these simply override definitions provided by the superclass ($m\ \epsilon\ sup$) by performing $d \rhd (m\ \epsilon\ sup)$.

**Example.** The previously described *GraphicsModule* is refined into a *ColoredGraphicsModule* as:

$$
\begin{array}{l}
\textit{ColoredGraphicsModule} = \textit{extendClass GraphicsModule Point ColoredPoint ext} \\
\text{where } \textit{ext} = \{\ \textit{color} \mapsto \textit{nil}, \\
\qquad\qquad \textit{setColor} \mapsto \lambda\ \textit{newCol}.\ \lambda \textit{self}.\ \{\textit{color} \mapsto \textit{newCol}\} \rhd \textit{self}\,\}
\end{array}
$$

**The Calculus in Action.** The following sections illustrate the calculus by expressing various module system semantics. For each of these module systems a set of operators is provided to describe elementary relationships used to compose units of modularization together for a particular programming language. Depending on the language under consideration, a module is a Java package, a C# namespace, a selector namespace (Modular Smalltalk [97]), a Ruby module [92], a virtual pattern in gbeta [37], a MZScheme unit [40], or a MixJuice module [45].

A typical operator is applied to a module $m_t$ with some arguments $m_s$ and some classnames $c$. We make use of the following conventions: (i) each operator is expressed as a module generator (combining two modules together yielding a new module), (ii) a $t$ subscript (*e.g.* $m_t$) refers to a template module: the result of the operator is a modified copy of the template module, (iii) an $s$ subscript (*e.g.* $m_s$) refers to the module provided as argument.

### 2.1.2 Java

Java [5] classes are grouped within *packages*. Packages can be "composed" with each other by means of the *import* relationship. (In this section, we do not consider the use of fully qualified names in Java or other languages, *i.e.* a class name preceded by the name of the package.) A package that imports a class from another package simply references this class by its name. There are two levels of granularity of the *import* relationship: (i) a package may import a single class from another package, and (ii) a package may import all visible classes (*i.e.* public at the package level).

The semantics of Java packages is expressed using two different import operators (*importClass* and *importPackage*) corresponding to the two granularity levels. Class privacy is expressed using the *private* operation, which is described later.

**Individual class import.** A package $m_t$ that imports a class $c$ defined in a package $m_s$ yields a copy of $m_t$ augmented with a new mapping for the imported class.

$$\begin{aligned} \textit{importClass} &: \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{M} \\ \textit{importClass} &= \lambda m_t.\ \lambda m_s.\ \lambda c.\ (\textit{hide}\ c)(m_t \parallel \{c \mapsto (\textit{fix}\ m_s)\ c\}) \end{aligned}$$

A package that imports a class cannot re-export it. This is expressed by $(hide\ c)$. According to Definition Section 12, $(\textit{hide}\ c)$ is a function that takes a module as argument and returns a copy of it where $c$ is removed from its interface only. This class $c$ is accessible to definitions in the importing package, however it cannot be accessed from another package.

A conflict occurs when a defined and an imported class have the same name. The restriction on key uniqueness is expressed by the $\parallel$ operation, which does not allow a key to be added if already present. A name that already refers to a defined class cannot be used to refer to an imported one and vice-versa.

References between classes are static, this means that importing a class does not rebind the references. Let's suppose a package *graphics* contains a class *PointFactory* that refers to a class *Point*. Importing the class *PointFactory* (in another package) does not impact the original references between *PointFactory* and *Point*, even if in the importing package a class *Point* is present. This restriction is expressed by $(\textit{fix}\ m_s)$. An example of a graphics package is:

```
package graphics;
public class Point {
    int x = 0, y = 0;
    void moveBy (int dx, int dy) {
        x = x + dx; y = y + dy;
    }
}
public class PointFactory {
    static void newPoint () {
        return new Point();
    }
}
```

$$\begin{array}{l}
\textit{graphics} = \lambda\, \epsilon.\ \{ \\
\quad \textit{Point} \mapsto \\
\qquad \{x \mapsto 0, \\
\qquad y \mapsto 0, \\
\qquad \textit{moveBy} \mapsto \lambda\, \textit{self}.\ \lambda\, dx.\ \lambda\, dy. \\
\qquad\quad \{\, x \mapsto \textit{self}\, x + dx, \\
\qquad\quad y \mapsto \textit{self}\, y + dy\} \rhd \textit{self}\}, \\
\quad \textit{PointFactory} = \\
\qquad \{\, \textit{newPoint} \mapsto \epsilon\ \textit{Point}\, \}\}
\end{array}$$

Importing the class PointFactory from graphics in a new package graphics2 where a class Point already exists does not make PointFactory use Point of graphics2.

$\textit{graphics2} = \textit{importClass}\ \lambda \textsf{self}.\ \{Point \mapsto \{\}\}\ \textit{graphics PointFactory}$

Evaluating *graphics2 PointFactory newPoint* returns an environment containing the keys $x$, $y$, and $moveBy$. However, evaluating *graphics2 Point* yields an environment with no mapping in it (according to the definition of $Point$ in $graphics2$.

**Individual package import.** Importing a whole package is equivalent to importing individually each class defined in the imported module. In a Java program, this is expressed as `package mt; import ms.*;`.

$$\begin{array}{rcl}
\textit{importPackage} & : & \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{M} \\
\textit{importPackage} & = & \lambda m_t.\ \lambda m_s.\ (\textit{hide}\ (\kappa\ m_s))(\lambda\textsf{self}.\ (m_t\ \epsilon) \rhd\ (\textit{fix}\ m_s))
\end{array}$$

When importing a whole package, locally-defined classes mask the classes that are imported. For instance, the following code is correct:

```
package p1;
public class A{}
public class B{}
```

```
package p2;
import p1.*;
public class A extends B{}
```

In the package p2, a class named A is locally defined, which masks the class A implicitly imported from p1. The name A within p2 refers to the p2 implementation of A, whereas in p1 the name A refers to the p1 implementation of A.

This is expressed by the $\rhd$ operator (which allows one mapping to be replaced by a new one) used in *importPackage*.

**Class privacy.** Classes declared as private in a package cannot be imported in other packages.

$$\begin{array}{rcl}
\textit{private} & : & \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{M} \\
\textit{private} & = & \lambda m.\ \lambda c.\ (\textit{hide}\ c)\ m
\end{array}$$

Syntactically this is written `package m; class C {...}`. The class `C` is visible only within `m` and not accessible from the outside.

### 2.1.3 C#

In C# a unit of modularization is called a *namespace*. Classes defined in a namespace can be imported under a different name (alias) in the importing namespace. C# provides a directive `using` to import a class (aliased or not) to a namespace and to import a whole namespace.

We express the semantics of the `using` directive with the three operators *usingClassAs*, *usingClass* and *usingNamespace*.

**Using alias and class directives.** An imported class can be aliased. This means that this class is accessed within the importing namespace under a different name. This is expressed in C# as `namespace MT { using A = MS.C; }`.

$$\begin{array}{rcl} \textit{usingClassAs} & : & \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{C} \rightarrow \mathcal{M} \\ \textit{usingClassAs} & = & \lambda m_t.\ \lambda m_s.\ \lambda a.\ \lambda c.\ (\textit{hide}\ a)(m_t \| \{a \mapsto (\textit{fix}\ m_s)c\}) \end{array}$$

The value *a* refers to the new name given to the class *c*. The need for $(\textit{hide}\ a)$ and $(\textit{fix}\ m_s)$ is similar to that in Java's case: $(hide\ a)$ restrains the imported class to be imported from another module, and $(fix\ m_s)$ ties all classes contained in $m_s$ to their dependencies. Also, when importing $c$ in $m_t$, dependencies of $c$ are preserved.

Importing a class without renaming it is expressed as an import aliased to the class name. This is expressed in C# as `namespace MT { using MS.C; }`.

$$\begin{array}{rcl} \textit{usingClass} & : & \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{M} \\ \textit{usingClass} & = & \lambda m_t.\ \lambda m_s.\ \lambda c.\ \textit{usingClassAs}\ m_t\ m_s\ c\ c \end{array}$$

Note that *usingClass* is equivalent to *importClass* previously described.

**Using the namespace directive.** As in Java, all the classes defined in a namespace can be imported using a single directive. In a C# program, this would be expressed as `namespace MT { using MS; }`.

$$\textit{usingNamespace} = \textit{importPackage}$$

The operator *usingNamespace* is equivalent to *importPackage* previously described.

### 2.1.4 Ruby

In Ruby, modules encapsulate functions, methods, classes, and constants. A module is a namespace that can also be used as a mixin [19] (*i.e.* class parametrized with its superclass).

**Modules as namespaces.** A module $m_t$ uses the code provided by another module $m_s$ by means of an include directive. This directive takes as parameter the name of the module intended to be reused. In the following example a module named MPoint defines a class Point containing a constructor. Another module MColoredPoint imports the definitions of MPoint and defines a subclass ColoredPoint of Point.

```
#Defined in a file MPoint.rb
module MPoint
    class Point
        def initialize(x, y)
            @x = x
            @y = y
            end end end
```

```
#Defined in a file MColorPoint.rb
load "MPoint.rb"
module MColoredPoint
    include MPoint
    class ColoredPoint < Point
        # ...
        end end
```

We define an *includeModule* operation that expresses the semantics of this include relationship between two modules. The resulting module of this operation is a merge between the two where (i) local definitions hide those of the imported module, and (ii) references of classes contained in the provider module are preserved.

$$\begin{array}{lcl} \textit{includeModule} & : & \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{M} \\ \textit{includeModule} & = & \lambda m_t.\ \lambda m_s.\ \lambda \mathsf{self}.\ m_t\ \epsilon \rhd (\textit{fix}\ m_s) \end{array}$$

Definitions contained in the importing module $m_t$ have precedence over those of the imported module $m_s$ in case of duplicate definitions. Before including a module, this one needs to be fixed because references between classes in this module are preserved.

**Modules as mixins.** As defined by Bracha and Cook [20], a mixin is a subclass definition that may be applied to different superclasses to create a related family of modified classes. In Ruby, a *module mixin* is a set of methods intended to be used as part of class definitions. When a module mixin defines only a set of functions, a module can be used as a mixin using an include operation stated within a class. In that case all the functions defined in the module are methods applicable to any instance of the class.

The following example shows the definition of a module mixin named MColor and a class ColoredPoint that uses it:

```
#Defined in a file MColor.rb
module MColor
    def getColor()
        @color
    end
    def setColor(col)
        @color = col
    end
    def setToBlack()
        self.setColor("Black")
    end
end
```

```
load "MColor.rb"
class ColoredPoint
    include MColor
    def initialize (x, y)
        @x = x
        @y = y
        self.setToBlack()
    end
    def getX() @x end
    def getY() @y end
end
```

The two methods getColor() and setColor(col) can be invoked on instances created by the class ColoredPoint. For example, the following code yields 5 and Black.

```
    @p = ColoredPoint.new(2,3)
    puts @p.getX() + @p.getY(); puts @p.getColor()
```

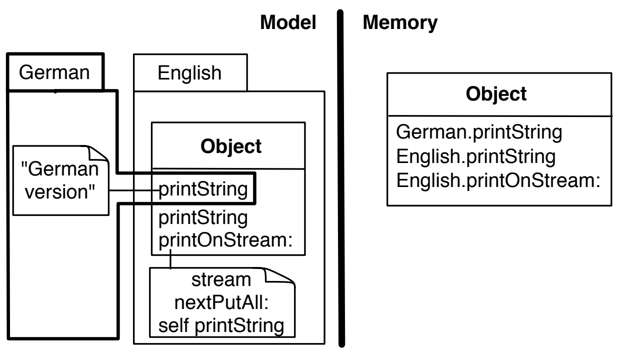


Figure 2.1: The class Object is composed of three methods: two versions of printString and a printOnStream: method

The semantics of the include operation which treats a module as a mixin can be expressed within the calculus using the following operator:

$$\begin{aligned} \mathit{newClassWithMixin} &: \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{D} \rightarrow \mathcal{M} \\ \mathit{newClassWithMixin} &= \lambda m_t.\ \lambda mixin.\ \lambda c.\ \lambda d. \\ & \quad m_t \| \{ c \mapsto \mathit{fix}(\lambda \sigma.\ d \rhd \mathit{mixin}\ \sigma) \} \end{aligned}$$

The module mixin intended to be included in the class $c$ is named *mixin*. The self-reference contained in the module *mixin* is rebound to the class being created by the *fix* operator.

Use of mixin is specified when a class is created, therefore we could not create an operator *include* because the class creation and use of mixin are combined. We defined the *newClassWithMixin* operator expressing the semantics of creating a new class composed of one mixin. In order to keep the model concise, we do not handle situations (i) where a subclass includes some mixins and (ii) when a class can be composed of several mixins. These can easily be expressed by an operator that would accept a superclass and a set of mixins.

### 2.1.5 Selector Namespaces

It is a tradition for Smalltalk and Lisp-based programming languages to offer a mechanism for introducing *class extensions* [14]. A class extension is a method addition or a redefinition applied to a class already present in a system (see Chapitre 1, Section 1.1). The result is an evolution of the behavior defined by this class without creating any subclass. The intent of this mechanism is to enable better distribution of responsibility among the involved classes.

This subsection focuses on *selector namespace*, a particular class extension mechanism offered by Modular Smalltalk [97] and Smallscript [85]. A selector namespace is a namespace containing class definitions and method extensions.

For instance, Figure 2.1 shows a class Object defined in a namespace English. This class contains a method printOnStream: and two methods printString. A first implementation of printString is provided by the selector namespace English, and the second one by German. In memory, the class Object contains three entries in its dictionary of methods. Each method has its name preceded by the name

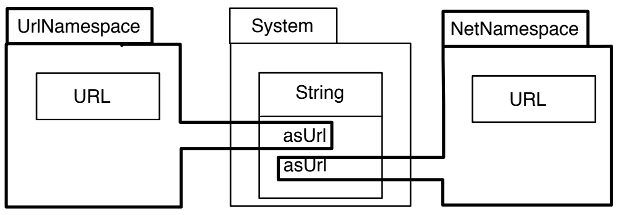


Figure 2.2: Two class extensions occurs on the class String: two methods asUrl are added by two different namespaces UrlNamespace and NetNamespace

of the selector namespace that implements it. At runtime, the lookup of methods is done according to which the namespace messages are sent from.

Sending the message printOnStream: within the selector namespace German is performed according to the following steps: (i) look up for German.printOnStream:, (ii) German.printOnStream: does not exist, therefore, (iii) because German imports Object from English, the algorithm continues by looking up System.printOnStream:. (iv) This method is found and invoked.

**Selector namespaces are non reentrant.** If within a selector namespace a particular method is not found, then the lookup is pursued in the selector namespace from which the class is imported. However, a method implementation is always looked up according to the namespace where the message is *actually sent*. For instance, invoking printOnStream: within the namespace German triggers the method English.printOnStream:. This method triggers printString, also the implementation used is English.printString because the call for it occurs in English. Even if called from within German, the method English.printOnStream: *cannot* invokes German.printString. We qualify this lookup as *non reentrant*.

**Avoiding conflict between class extensions.** The concept of selector namespaces was introduced first with Modular Smalltalk [97], and more recently with Smallscript [85], a Smalltalk implementation for .Net. A selector namespace defines a namespace for methods and is used to manage conflicting class extensions. Within such a namespace one can extend any class in the system without producing conflict: another namespace can contain a class extension having the same name. This is illustrated in Figure 2.2 where the class String is extended by two namespaces UrlNamespace and NetPackage, each of them adding a method asUrl.

**Importing and extending.** Two operators can be performed on a selector namespace: (i) import and extend a class (*extend*), or simply (ii) import (*import*) a class. Importing from a namespace $m_s$ and extending a class $c$ with a set of methods $d$ is expressed with the *extend* operator:

$$\begin{aligned} \textit{extend} &: \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{M} \rightarrow \mathcal{M} \\ \textit{extend} &= \lambda m_t.\ \lambda m_s.\ \lambda c.\ \lambda d.\ m_t \| \{c \mapsto (\textit{fix}\ d) \rhd ((\textit{fix}\ m_s)\ c)\} \end{aligned}$$

When a selector namespace extend a class, the extending methods need to keep a reference to the scope that contains them. In order to keep this reference, a set of methods is a module (note that for the above formula $d \in \mathcal{M}$) and it is fixed when used to extend a class *(fix d)*.

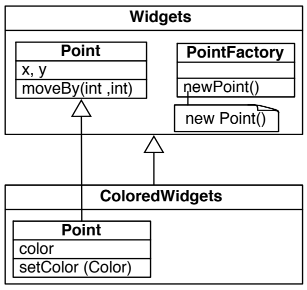


Figure 2.3: The outer class ColoredWidgets refines the class Point. Because classes are looked up, points produced from a factory obtained from ColoredWidgets are colored

For instance, the namespace English that contains the class Object (Figure 2.1) is defined as:

$$\begin{aligned}&\textit{English} = \textit{extend}\ \lambda\textsf{self}.\ \{\}\ \textit{Object}\\ &\qquad\lambda s.\ \{\textit{printString} \mapsto \lambda \textit{self. englishVersion}\},\\ &\qquad\quad \textit{printOnStream} \mapsto \lambda\textit{self. } (s \rhd \textit{self})\ \textit{printString}\}\end{aligned}$$

This class Object is extended with a German implementation of the printString method. The German namespace is defined as:

$$\begin{aligned}&\textit{German} = \textit{extend}\ \lambda\textsf{self}.\ \{\}\ \textit{English Object}\\ &\qquad\lambda s.\ \{\textit{printString} \mapsto \lambda \textit{self. germanVersion}\}\end{aligned}$$

The second operator associated to selector namespace is the *import*. A namespace imports a class without extending with:

$$\begin{aligned}\textit{import} &: \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{M}\\ \textit{import} &= \lambda m_t.\ \lambda m_s.\ \lambda c.\ \textit{extend}\ m_t\ m_s\ c\ \ \lambda\textsf{self}.\ \{\}\end{aligned}$$

### 2.1.6 Virtual Classes

The notion of *virtual classes* offered by gbeta [38], Caesar [65] or Keris [99] allows class names to be dynamically looked up (rather than statically, at compilation time). Virtual classes unify the method and class lookup under a common lookup algorithm: as well as methods, class definitions are looked up along the inheritance of encapsulating classes.

In gbeta, virtual classes are implemented as inner classes, and outer classes define the unit of modularization. Class names are looked up in the same way that methods are looked up: inner classes can be refined within subclasses of the outer class.

Figure 2.3 shows the case where a set of inner classes contained in an outer class Widgets is refined in ColoredWidgets. The class Point defined in Widgets is subclassed into a new class Point in ColoredWidgets. The class PointFactory is visible into this last because it is inherited. When the method newPoint() is triggered, the class Point is looked up according to the hierarchy of encapsulating classes. If the factory is obtained from an instance of the outer class ColoredWidgets, then the points produced are colored.

Two operators modeling inheritance are involved when handling virtual classes: (i) inheritance between outer classes and (ii) inheritance between inner classes. Within our calculus the semantics of

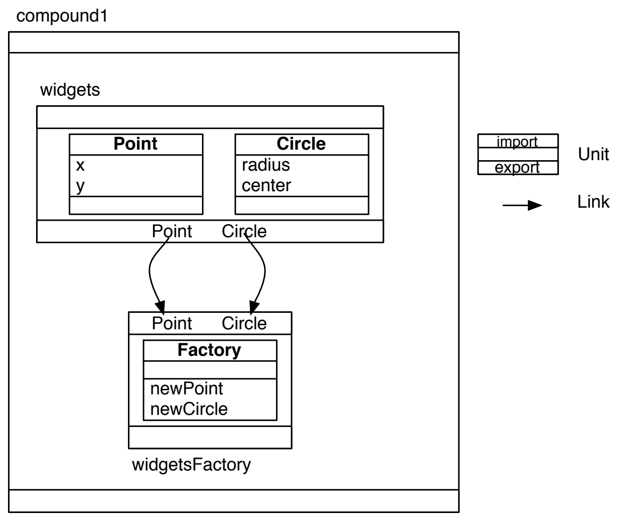


Figure 2.4: Composing two units, widgets and widgetsFactory, into one compound, compound1.

these two operators are expressed with the operators *extendEncapsulated* and *extendInner*.

Elements in the subclass hide ones defined in the superclass. Inheritance between outer classes is simply expressed using the operator $\triangleright$. The *extendEncapsulated* operator is defined as:

$$\begin{aligned} \textit{extendEncapsulated} &: \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{M} \\ \textit{extendEncapsulated} &= \lambda m_t.\ \lambda m_s.\ m_t\ \triangleright m_s \end{aligned}$$

Inheritance between inner classes is defined in a similar way than classical inheritance (Definition Section 13). The *extendInner* operator is:

$$\begin{aligned} \textit{extendInner} &: \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{D} \rightarrow \mathcal{M} \\ \textit{extendInner} &= \lambda m_t.\ \lambda c.\ \lambda d.\ \lambda \epsilon.\ (m_t \backslash c)\ \epsilon \| \{c \mapsto d \triangleright\ (m_t\ \epsilon\ c)\} \end{aligned}$$

For instance, assuming an outer class *Widgets*, *ColoredWidgets* is defined as:

*ColoredWidgets* = *extendInner (extendEncapsulated* $\lambda$self. $\{\}$ *Widgets)*
*Point colorExtensions*
where *colorExtensions* $=\{$*color* $\mapsto$ *black,*
*setColor* $\mapsto$ $\lambda$*newCol.* $\lambda$*self.*$\{$*color* $\mapsto$ *newCol*$\}$ $\triangleright$ *self*$\}$

### 2.1.7 Units

MZScheme [40] offers an advanced module system based on units. A program *unit* is an unevaluated fragment of code intended to be linked with other units in order to form executable programs. There is no global namespace of units.

A unit describes its import requirements without specifying a particular unit that supplies those imports. The actual linking of the unit is specified externally at a later stage. Unlike in ML, unit linking is specified for groups of units with a graph of connections, which allows mutual recursion across unit

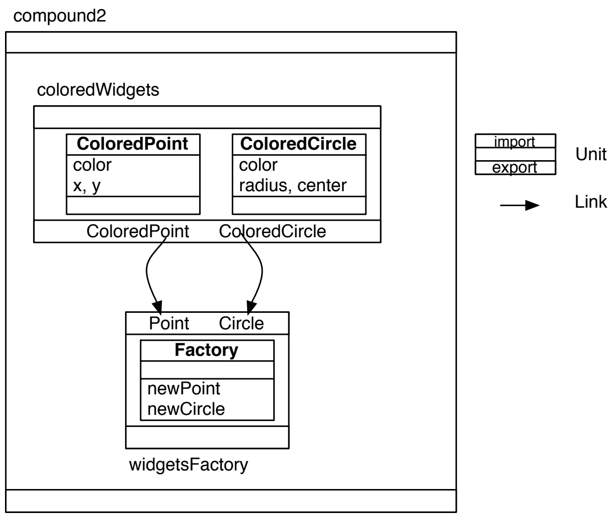


Figure 2.5: The unit widgetsFactory is composed with a new unit coloredWidgets.

boundaries. Furthermore, the result of linking a collection of units is a new (compound) unit that is available for further linking. One important point of this module system is that connections between modules are specified separately from their definitions.

The *link* operation is defined as:

$$\begin{array}{lcl} \textit{link} & : & \mathcal{M} \to \mathcal{M} \to \mathcal{C} \to \mathcal{C} \to \mathcal{M} \\ \textit{link} & = & \lambda m_t.\ \lambda m_s.\ \lambda a.\ \lambda c.\ \lambda \epsilon.\ m_t\ \epsilon \| \{a \mapsto m_s\ \epsilon\ c\} \end{array}$$

Applying a *link* combinator to units $m_t$ and $m_s$ makes the value associated to $c$ in $m_s$ available in $m_t$ under the alias $a$. For instance, Figure 2.4 shows a *widgets* unit defining two classes *Point* and *Circle* and a *widgetsFactory* unit defining a class *Factory*. The corresponding expression is:

$$\begin{array}{lcl} \textit{widgets} & = & \lambda\epsilon.\ \{\textit{Point} \mapsto \{\ x \mapsto 0,\ y \mapsto 0\}, \\ & & \quad \textit{Circle} \mapsto \{\textit{radius} \mapsto 0,\ \textit{center} \mapsto \epsilon\ \textit{Point}\}\} \\ \textit{widgetsFactory} & = & \lambda\epsilon.\ \{\textit{Factory} \mapsto \{\ \textit{newPoint} \mapsto \epsilon\ \textit{Point}\}, \\ & & \qquad\qquad \textit{newCircle} \mapsto \epsilon\ \textit{Circle}\}\} \end{array}$$

To make *widgetsFactory* use the widgets a new compound *compound1* is created using:

*compound1 = link (link widgetsFactory widgets Point Point)*
*widgets Circle Circle*

The unit *compound1* is the result of linking classes *Point* and *Circle* defined in *widgets* under their original names (*Point* and *Circle*) in the unit *widgetsFactory*. This compound is obtained by linking the class *Point* obtained from *widgets* to the name *Point* in the unit *widgetsFactory*. Then, the class *Circle* of *widgets* is linked to the name *Circle* in *widgetsFactory*.

As illustrated in Figure 2.5, the widget factory is used by colored widgets, without altering the original definition of widgetsFactory. This is expressed with:

$\textit{coloredWidgets} \ = \ \lambda\epsilon.\ \{\textit{ColoredPoint} \mapsto \{\ \textit{color} \mapsto \textit{blue},\ x \mapsto 0,\ y \mapsto 0\},$

$$\begin{aligned}
&\qquad\qquad\quad \textit{ColoredCircle} \mapsto \{\textit{color} \mapsto \textit{blue, radius} \mapsto \textit{0},\\
&\qquad\qquad\qquad\qquad \textit{center} \mapsto \epsilon\ \textit{Point}\}\}\\
&\textit{compound2} = \textit{link (link widgetsFactory coloredWidgets ColoredPoint Point)}\\
&\qquad\qquad \textit{coloredWidgets ColoredCircle Circle}
\end{aligned}$$

The unit *compound2* is the result of linking the class *ColoredPoint* and *ColoredCircle* obtained from *coloredWidgets* to the unit *widgetsFactory* under the names *Point* and *Circle*.

### 2.1.8 MixJuice

MixJuice [45] is a module system for Java in which a module encapsulates the differences between the original program and the extended program. The difference is a set of definitions of additions and modifications of classes, fields and methods. Modules may inherit other modules. As will be explained in Section 4.4.2, the difference with classboxes and virtual classes is that different versions of a class cannot coexist at the same time in the same system. For instance, a module defining a point is defined as:

```
    module point {
        define class Point {
            define int x = 0;
            define int y = 0;
            define void moveBy (int dx, int dy) { x += dx; y+= dy;}
            define String toString () { return "point "+x+","+y;}
        }
    }
```

The keyword define is used to define a new class member. Without this keyword, the class is refined. Here the class Point is refined in a colored point:

```
module coloredPoint extends point {
        class Point {
            define Color c; // Variable addition
            // Redefinition of the method toString()
            String toString () { return "colored point "+x+","+y;}
        }
    }
```

The example above uses a single inheritance link. However, multiple inheritance is permitted. In that case, all modules are linearized by topological sort (similar to the class linearization done in CLOS [31]). The key point of the MixJuice module system is that an original and a modified version of a set of classes *cannot* be present at the same time in the same system. This is the major difference with classboxes apart from the import relationship being specified as inheritance between modules (more explanation will be given further, in Section 4.4.2).

Within our calculus, semantics of MixJuice modules are expressed using two operators: *extends* to express inheritance between modules, and *refineClass* to refine some part of a class using redefinition of its class members.

Inheritance of modules is expressed as:

$$\begin{aligned}
\textit{extend} &: \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{M}\\
\textit{extend} &= \lambda m_t.\ \lambda m_s.\ m_t \rhd m_s
\end{aligned}$$

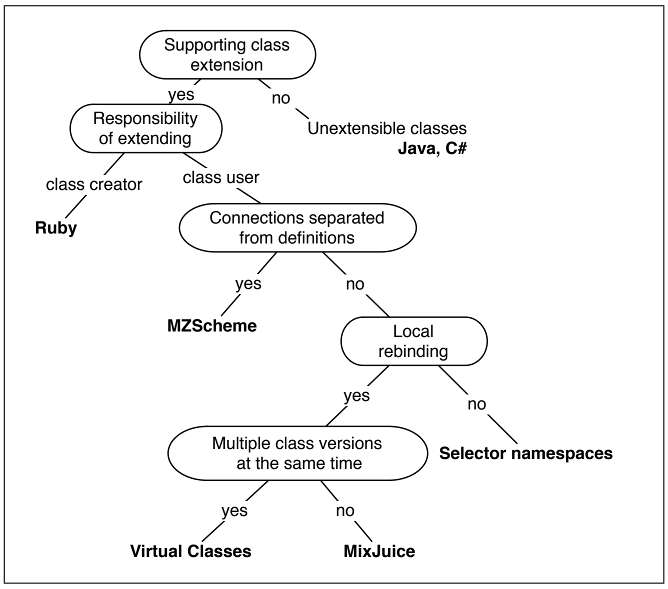


Figure 2.6: Taxonomy of different module systems

This operator is the same as that expressing inheritance between outer classes (*extendEncapsulated*) for gbeta.

Classes are refined using *refineClass*:

$$\begin{array}{lcl} \textit{refineClass} & : & \mathcal{M} \rightarrow \mathcal{C} \rightarrow \mathcal{D} \rightarrow \mathcal{M} \\ \textit{refineClass} & = & \lambda m_t.\ \lambda c.\ \lambda d.\ \lambda \epsilon.\ (m_t \backslash c)\epsilon \parallel \{c \mapsto d \rhd (m_t\ \epsilon\ c)\} \end{array}$$

For a given class, MixJuice does not allow multiple versions to be present at the same time in the same running system. Therefore, a program can be composed either of colorless points (*i.e.* using the point module) or of colored points (*i.e.* using the coloredPoint module). But a colorless point and a colored point cannot coexist in the same system. Note that this restriction is not expressed in the operators described above.

### 2.1.9 Module System Analysis

We present some of the key characteristics that the different module systems exhibit and that the calculus helps to clearly identify. Figure 2.6 presents a classification of the module systems we have discussed. The module systems are classified according to properties that enable extension.

**Unextensible Classes.** With Java or C#, classes can only be refined through subclassing. The definition of the imported class cannot be enlarged with a set of new definitions (method or field addition or method change) from a package other than the package defining it. We refer to such imported classes as *unextensible*. These extensions have to be defined on subclasses.

This restriction is expressed within the calculus by (i) *fixing* the module from which the imported class comes from and (ii) not extending this class with $\rhd$ (override) or $\parallel$ (extend). Basically, this is identified by (i) the pattern (*fix* $m_s$) $c$ present in the import statement and by (ii) the absence of $\rhd$ or $\parallel$.

For instance, importing a class in Java (Section 2.1.2) is defined as:

$$\textit{importClass} \;=\; \lambda m_t.\ \lambda m_s.\ \lambda c.\ (\textit{hide}\ c)(m_t \parallel \{c \mapsto (\textit{fix}\ m_s)\ c\})$$

The new class $c$ is defined as $((\textit{fix}\ m_s)\ c)$ which make it unextensible because it contains *fix* and no overriding or extension operations.

**Class Extensions.** Some module systems allow new method definitions specified in a module to be added to a class defined in another module. This mechanism complements subclassing. A class extension is the result of a separation between a definition of a class and definitions that compose this class (*i.e.* method definitions). AspectJ [6] with the notion of inter-type declaration, MultiJava [66] with open-classes, HyperJ [91] with hyper-slices, Smalltalk, CLOS, and Objective-C offer such a mechanism. Within such systems, there is a conceptual difference between a class definition and its method definitions: a method definition is not physically included in a class definition but can be defined externally to the class it belongs to.

The ability of a module system to offer class extensions is expressed in performing an $\triangleright$ or a $\parallel$ operation on the imported class. For instance, ModularSmalltalk (Section 2.1.5) allows a class to be extended by adding new definitions to it. This is illustrated in the *extend* operation for selector namespaces (Section 2.1.5):

$$\textit{extend} \;=\; \lambda m_t.\ \lambda m_s.\ \lambda c.\ \lambda d.\ m_t \| \{c \mapsto d \triangleright ((\textit{fix}\ m_s)\ c)\}$$

The imported class $c$ is the result of $((\textit{fix}\ m_s)\ c)$, *i.e.* the definition of the class $c$ looked up in the fixed (self-rebound) module. A set of new definitions is added to it by using $d \triangleright \_$. The class obtained from the provider module $m_s$ is extended with a set of definitions $d$.

With virtual classes, class extensions and mixins, a part of the definitions composing a class can be stated at a different location than the class declaration. Gbeta allows a class to be refined in another unit of modularization (*i.e.* outer classes). With selector namespaces, part of the behavior can be defined in a namespace different from the one where the class is declared.

**Extension Responsibility.** We defined a class extension as a method addition or redefinition for an already existing class (Section 2.1.5). The responsibility of extending a class belongs to its users. For instance, with selector namespaces, a class is imported, and then extended. This responsibility belongs to the user of this class, and not to its creator.

The decision to use a mixin or not when creating a class is made by the creator of the class. This is why, in Ruby, including a module mixin in the definition of a class is not a class extension in the sense we defined previously. The choice of using a mixin is taken when the class is created. This is expressed with the *newClassWithMixin* operator where the mixin to use is designated when the class is created.

$$\begin{aligned}\textit{newClassWithMixin} \;=\;& \lambda m_t.\ \lambda mixin.\ \lambda c.\ \lambda d.\\ & m_t \| \{c \mapsto \textit{fix}(\lambda \sigma.\ d \triangleright \textit{mixin}\ \sigma)\}\end{aligned}$$

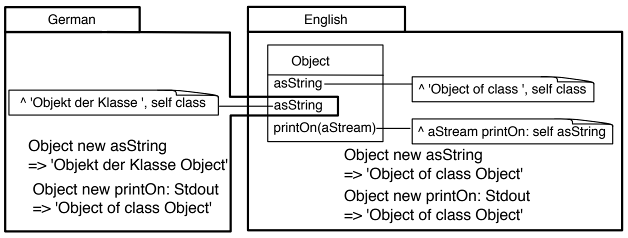

German
English
Object
asString
asString
printOn(aStream)
^ 'Objekt der Klasse ', self class
^ 'Object of class ', self class
^ aStream printOn: self asString
Object new asString
=> 'Objekt der Klasse Object'
Object new printOn: Stdout
=> 'Object of class Object'
Object new asString
=> 'Object of class Object'
Object new printOn: Stdout
=> 'Object of class Object'

Figure 2.7: In Modular Smalltalk, namespace selectors are not reentrant: invoking printOn() from the German namespace does not invoke the German version of asString

**Local Rebinding.** The *local rebinding* property is provided by an extension mechanism such that extensions are visible by and see former definitions of the code. A change defined by some class extensions can use the former definitions, and former definitions can use the new extensions.

A module system offers a *local rebinding* property if within an *import* statement a *fix* operation is **not** performed on the module from which a class is imported. The effect of this fixpoint is to make the imported class use definitions of the importer module. In the studied module systems, gbeta, and MixJuice have the local rebinding property but not selector namespaces.

For instance, in gbeta (Section 2.1.6), refinements over a set of inner classes are defined within a subclass of the encapsulating class using the *extendEncapsulated* operation:

$$\textit{extendEncapsulated} = \lambda m_t.\ \lambda m_s.\ m_t\ \rhd m_s$$

As no *fix* operation is involved, the class definition in the parent encapsulating class can introduce new refinements.

**Non reentrant.** Selector namespaces (Section 2.1.5) allow a class to be imported and then extended with new methods. These new methods can invoke the former methods. However, the other direction is not possible: former methods cannot invoke redefined definitions. We call this property *non-reentrance*. Selector namespaces do not support local rebinding because they are not reentrant. For instance, in Smallscript [85] a German translation could be defined as shown in Figure 2.7. A namespace German extends the class Object with a German translation of asString. However, the English version of this method belongs to the namespace English where printOn(aStream) is specified. Therefore, the English version of asString is always invoked by printOn(aStream) even if the execution occurs from within the German namespace.

The *extend* operator used to express the class extension semantics with selectors namespace is:

$$\textit{extend} = \lambda m_t.\ \lambda m_s.\ \lambda c.\ \lambda d.\ m_t \| \{c \mapsto d \rhd ((\textit{fix}\ m_s)\ c)\}$$

The module $m_s$ is fixed before taking the definition of the class $c$ intended to be extended. The definition obtained from $((\textit{fix}\ m_s)\ c)$ cannot call the extensions defined by $d$.

With virtual classes, on the other hand, a German translation would be printed whenever the printOn (aStream) is invoked from within the package German. These two systems exhibit the local rebinding property. Local rebinding is characterized by taking into account the calling context.

**Privacy in a Module.** Module privacy is expressed using the *hide* operator. For example to declare a class as private within a Java package, one may use:

$$\textit{private} = \lambda m.\ \lambda c.\ (\textit{hide}\ c)\ m$$

With Java and C#, an imported class may be referred to *only* within the importing package or namespace. An imported class does not belong to the module's interface, and so, cannot be imported from that module by yet another module. This is expressed by the *hide* operator applied to the result of the import operation. For instance, the C# *usingClassAs* operation (Section 2.1.3) expresses the class import under an alias and it is defined as:

$$\textit{usingClassAs} = \lambda m_t.\ \lambda m_s.\ \lambda a.\lambda c.\ (\textit{hide}\ a)(m_t \| \{a \mapsto (\textit{fix}\ m_s)\ c\})$$

The use of $(\textit{hide}\ a)$ prevents one from importing the class from another module.

**Mixin Behavior.** A Ruby module may define functions that are turned into methods whenever they are used by a class (Section 2.1.4). We call this a module mixin. The module mixin is applied to the environment representing the class being defined. This is expressed by the use of the fixpoint operator *fix* within *newClassWithMixin*.

$$\begin{array}{lcl} \textit{newClassWithMixin} & = & \lambda m_t.\ \lambda mixin.\ \lambda c.\ \lambda d. \\ & & m_t \| \{c \mapsto \textit{fix}(\lambda\sigma.\ d \rhd \textit{mixin}\ \sigma)\} \end{array}$$

The expression $mixin\ \sigma$ binds the class being defined $\sigma$ to the module argument $\epsilon$ of $mixin$. This mechanism is illustrated here with Ruby's module mixin but it is also applicable to other mixin mechanisms like those of MzScheme [41] or Jigsaw [19].

**Connections Separated from Module Definitions.** The advantage of units over conventional module and class languages is that connections between modules or classes are separately specified from their definitions. Separating the definition of classes from their use in different modules makes it easy to replace the original classes with new classes without modifying the client.

### 2.1.10 Other Formalisms

Considerable effort has been invested in studying theoretical foundations of module systems, but to the best of our knowledge there is no work defining a calculus to compare existing object-oriented module systems. We limit this section to summarizing work done in expressing module systems of various object-oriented languages.

In their work on mixins operators Van Limberghen and Mens [62] present the operator *encaps* appropriated to deal with multiple inheritance problems, which is an alternative to the *hide* operator proposed by Bracha and Lindstrom [22]. They mainly focused on multi-inheritance mechanisms.

Ancona and Zucca [3] define a module calculus suitable for encoding various existing mechanism for composing modules. They define a module as a set of imports, a set of exports, and a set of function definitions, *i.e.* components. Modules are composed using a set of operators: *sum*, *reduct*, *freeze*, *selection*. The operator *sum* glues two modules together, and is roughly equivalent to our *override* (in our calculus, import and export are not explicitly part of a module). The operator *reduct* is a form of renaming; import and export components are separately renamed via two renamings. The *freeze* operators binds input to output names. Finally, *selection* is used by clients of a module to access its components. Their approach enables a large variety of existing mechanisms for combining software components to be expressed (*e.g.* ML functions, mixin modules). However, no attempt has been made to express module systems of mainstream object-oriented languages.

Bono *et al.*, [17], define some basic object-oriented constructs in a lambda-calculus with records. While they focus on expressing mixin composition as the primary extension mechanism, they do not address the notion of modules and composition operators.

Leroy [55] presents an implementation of an SML-like module system. The SML module system consists of three notions: a structure which is a set of named components, a signature which is an interface for a structure, and a functor which is a function that maps a structure into a new structure. A module is defined by a structure which can be associated with more than one signature. A module can either be user-defined, or the result of applying a functor to another module. Leroy describes an attempt at transferring thus module system to other languages such as *core C* and *mini-ML* which are subsets of C and of ML, respectively.

Linking modules together by functor application prevents the definition of mutually recursive types or procedures across modules boundaries [40]. Objective Caml [71] provides an object-oriented layer as well as an SML-like module system. We did not include this in our comparison because it would be redundant with the study of MZScheme units.

### 2.1.11 Conclusion

We have defined a simple calculus in which modules and classes are combined using a set of basic operators like *hide* and *fix*. Then, for various object-oriented programing languages, we expressed their module systems (*i.e.* Java packages, C# namespaces, gbeta virtual classes, ...) by defining operators like *import* or *extend*. The focus of this section is to express various packaging mechanisms using a common foundation. Results of this analysis are summarized in the taxonomy presented in Section 2.1.9. Even if only a very few languages are treated in this section compared to the number of module systems proposed over the last decades, mainstream languages as well as representative languages are studied.

When defining the representation of classes, we expressed inheritance with the *extendClass* operator. However, we did not model the *super* reference. This would have brought some additional complexity to the calculus that would have shifted the focus of this section. Furthermore, not all the of languages we considered support *super* (*e.g.* gbeta).

Numerous formalisms have been developed in recent years to model new kinds of module systems and their features. However, to our knowledge, ours is the first attempt to develop a general calculus for modeling and comparing the diversity of module systems provided by various mainstream object-oriented programming languages.

## 2.2 Packaging and Deploying Static Changes

Over the last decade considerable research has focused on new ways to modularize or change a system. Previous sections presented a comparison of various module systems according to their module operators. This section summarizes various approaches to packaging and deploying changes at compile or development time (static changes).

The work presented in this section can be classified according to five families: (i) class extensibility (class extensions, Unit, Jiazzi, open classes), (ii) module (MixJuice, MJ), (iii) alternative to inheritance (mixins, virtual classes, nested and hierarchy inheritance), (iv) other approaches (AOP, namespaces).

### Class Extensibility

**Open classes.** MultiJava [66] is an extension of Java that supports open classes and multiple method dispatch. An open class is a class to which new methods can be added. Method redefinitions are not, however, allowed: an open class cannot have one of its existing methods refined.

**Jiazzi.** The *unit* system of MZScheme has been ported to Java. Jiazzi [60] is an enhancement of Java that adds support for encapsulated code modules as units. The main difference with MZScheme is that Jiazzi enables the creation of open classes that can be enhanced with new methods and fields without invasively modifying the original definitions or breaking their existing subclasses. This enables a modularization of cross-cutting concerns [61]. Refinements occur with links between units.

**Parcels.** VisualWorks Smalltalk has a sophisticated deployment mechanism named *parcels* [68]. Parcels are an atomic deployment mechanism for objects and source code that supports shape changing of classes, method addition, method replacement, and partial loading. Parcels permit object and source code transportation between and importation into systems. The system uses a binary format supports extremely fast loading and its provision of method replacements and partial loading frees the programmer from maintenance tasks required by less flexible technologies.

**Summary.** Change at the level of the class is done either by means of class extensions, open classes, or use of mixins. The characteristics of these approaches are twofold:

- A single and unique version of a class can be present in the same system at the same time. This is the consequence to forbid method redefinition (as with *open classes*) or to make a method redefinition global (as with *class extension*).
- Units use mixins to define refinement at the level of a class. New mixins can be defined to extend a class, and are applied to a class by creating a subclass.

### Modules

**Mixjuice.** Mixjuice [45] defines difference-based modules, in which a module can refine a class defined in another module by adding new class members. A refined class constitutes a new version.

Mixjuice provides constructs to define “delta” modules intended to be applied to a “base” module. This is achieved by creating multiple versions of a Java package. However, at runtime, only one particular version can be present in the system.

**MJ.** MJ [30] is a module system for Java that provides a high-level interface to abstract low-level Java technical issues related to class loading. The focus of MJ is to support the deployment of different versions of the same package. As such with MJ changes cannot be added to existing classes. In MJ, a module contains the following information: (i) class definition, (ii) dependencies with classes offered by other modules, (iii) access control for this module’s provided classes like class privacy and restriction for the clients in subclassing provided classes, and (iv) some initialization code.

By removing some technical limitations of the dynamic class loading mechanism related to the use of `CLASSPATH`, MJ allows multiple versions of a class to coexist at the same time within a system. These versions are referenced by different namespaces (*i.e.* classloaders), therefore, they are considered to be two different classes. New versions of a class cannot be propagated to formerly collaborating classes without modifying the original dependancies: modules are considered to be black boxes in which contained classes cannot be modified.

**Summary.** By providing for a given module $m$ a new version $m'$ of it:

- With Mixjuice, clients of $m$ can instead collaborate with $m'$ if the new version $m$ is not present in the system.
- With MJ, clients of $m$ cannot collaborate with $m'$ if both $m$ and $m'$ are present in the same system.

Current approaches to package changes by means of module operators have limitations regarding collaborations between different version of a modules and its clients.

### Alternatives to Inheritance

**Virtual classes.** Virtual classes were originally developed for the language BETA [53], primarily as a mechanism for generic programming rather than for extensibility [58]. Keris [99], Caesar [65], and gbeta [36] offer such a mechanism, where method and class lookup are unified under a common lookup algorithm. Virtual classes are not statically safe because they permit types of method parameter to change covariantly with subtyping. In a similar way that a method is looked up according to an instance, a class is looked up according to an instance (*i.e.* an encapsulating class). With such a unification of method and class lookup, the role of a class is overloaded with semantics of packages and objects constructor.

**Hierarchy Inheritance.** Cook [29] presents a use of inheritance as a derivation of modified hierarchies or other graph structures. Links between nodes in a graph are interpreted as self-references from within the graph to itself. By inheriting the graph and modifying individual nodes, any access to the original nodes is redirected to the modified versions. For example, a complete class hierarchy may be inherited, while new definitions are derived for some internal classes. The result of this inheritance is a modified class hierarchy with the same basic structure as the original, but in which the behavior of all classes modified that depend upon the classes explicitly changed is modified. Hierarchy inheritance is based on having a lookup of classes and on relationship between groups of classes.

**Nested inheritance.** The Jx programming language [70] is an extension of Java where members of an encapsulating class or package may be enhanced in a subclass or subpackage. Packages may have a declared inheritance relationship. Nested classes in Jx are similar to virtual classes. Unlike virtual classes, nested classes in Jx are attributes to their enclosing class, not attributes of instances of their enclosing class.

**Scala.** Scala [83] is a statically-typed object-oriented and functional programming language. Scala introduces a new concept to solve the extensibility problem (3.1.2): *views* allow one to augment a class with new members. Views follow some of the intuitions of Haskell's type classes, translating them into an object-oriented approach. The scope of a view can be controlled, and competing views can coexist in different parts of one program. A view is statically applied by the compiler to satisfy type constraints. For instance, if a variable `anA` is of type `A`, the compiler would translate an expression `var aB: B = a`, which declares a variable `aB` of type `B` and initializes it with a reference to `anA`, as `var aB: B = view(anA)`, where `view` is a method (or a function) provided by the programmer, taking an argument of type `A` and returning an object of type `B`. A conversion is done by using type information provided by the programmer.

**Mixin Layers.** A collaboration-based design [44, 95] aims at supporting large-scale refinements. A *collaboration* is a set of *roles* applied to a set of *participant objects*. Collaborations are layered linearly to form an application. In mixin layers [87], Smaragdakis and Batory represent a collaboration as a C++ template, a role as a mixin [21], and a participating object as a class. A layered application that uses mixin layers is open to changes by adding new collaborations. However, for an application that is not layered, mixin layers do not offer a satisfying solution to support unanticipated changes.

**Feature-oriented programming.** Feature-Oriented Programming is the study of feature modularity in product-lines [78]. AHEAD [9, 56] is an approach to Feature-Oriented Programming (FOP) where a base system is regarded as a constant and refinements intended to be added are functions adding features to this base system. A refinement is a function that takes a program as input and produces a refined program as output. FOP advocates program construction as a set of functions applied to a base system. New changes are modeled as new functions.

**Generic type.** Torgersen [93] uses generic type extensions of C# and Java to solve the extensibility problem in a secure and type safe manner. His solutions rely on the use of F-bounds [25] and wildcards in the declaration of type variable to make them type-safe when a system is extended with new data-types and operators. However, use of generic type has to be foreseen prior to apply an extension, as a consequence, this approach does not support unanticipated changes.

**Summary.** Triggering a class lookup for each class reference is an approach to define specialization and enhancement of modules, in a similar way of class inheritance. Modules are composed using inheritance links.

### Other Approaches

**Aspect-oriented programming.** Hyper/J [73] is based on the notion of *hyperspaces*, and promotes composition of independent *concerns* at different times. Hyperslices are building blocks containing fragments of class definitions. They are intended to be composed to form larger building blocks (or complete systems) called *hypermodules*. A hyperslice defines methods for classes that are not necessarily defined in that hyperslice: class members are spread over several hyperslices. With its notion of inter-type, AspectJ [6] allows class members to be separated from the class definition by being defined in an aspect.

**Sister namespaces.** In Java, a class type is uniquely identified at runtime by the combination of a class loader and a fully qualified class name. The same class loaded into two different class loaders (*i.e.* namespaces) has two distinct types [82]. Let's assume that two classloaders N1 and N2 load the same class C. One instance of the class C in the classloader N1 cannot be regarded as an instance of C in a second classloader N2 because they have different types. This is identified as the *problem of the version barrier*. *Sister namespaces* [82] relax the version barrier between application components by defining the notion of binary compatibility and extending the type checker. Sister namespaces make the exchange of instance of different class versions possible across classloaders by relaxing the type checker.

## 2.3 Dynamic Application of Crosscutting Changes

This chapter first presented a comparison of various module systems, then Section Section 2.2 was about packaging and deploying static changes. This section presents the state of the Art in installing and removing dynamic crosscutting changes.

Dynamic and crosscutting adaptation has been the source of various researches. We classify them according to two different approaches: dynamic code adaptation and dynamic extension and replacement of classes.

**Dynamic Code Adaptation.** With PROSE [77] aspects can be woven and unwoven at run-time. PROSE allows advices to be joined on a smaller number of join points such as accessing/modifying methods and entering/returning methods. However, it does not offer any feature related to class extension.

Guaraná [72] and MetaclassTalk [18] define a reflective architecture based on metaobjects supporting dynamic method switching. Even if they both support dynamic application changes, the definition of the class itself is static: dynamic evolution is done at the metalevel. By providing true delegation, Lava [50] supports dynamic unanticipated changes using class wrappers.

The SELF prototype-based language [1] allows slots to be dynamically added: a prototype can be freely extended at run-time with new variables or new methods. However, these slot additions are global so they may override already existing slots which might affect any clients.

**Dynamically Extending and Replacing Classes.** In MultiJava [28] an *open class* is a class that can have its behavior extended with new methods. However, redefining methods and adding new instance variables are not permitted.

IguanaJ [79] supports dynamic adaptation of the behavior of arbitrary classes and objects in unanticipated ways. Internal behavior of classes can be changed but it does not support modifications to their external interfaces. No preprocessor and no source code or the base application are required. However, it is not clear if a compiler is required or not.

Dynamic Classes [59] and HotSwap [32] allow Java programs to change class definitions during their execution. Instances are updated according to the new class definition following the tradition of Smalltalk and Lisp which have been using such a technique for decades to support interactive and incremental programming. However, Dynamic Classes and HotSwap do not provide higher level mechanisms for applying cross-cutting changes to several classes.

## 2.4 Conclusion

This chapter analyses the current mechanisms to support unanticipated changes according to three approaches: (i) module constructs to support extensibility, (ii) packaging and deployment of static changes, and (iii) dynamic installation and removal of crosscutting changes.

By means of a simple calculus we expressed module systems of various object-oriented programing languages in order to understand and compare them. The module calculus and the analysis of module systems were presented in Section 2.1.

Section 2.2 describes various strategies to package and deploy static changes (i) by means of class extensions, (ii) using dedicated module operators, (iii) defining a lookup for class references.

Section 2.3 presents various strategies to dynamically install and remove crosscutting changes using two different approaches: (i) dynamic code adaptation using aspects and meta objects and (ii) dynamic extension and replacement of classes by means replacing class definitions.

Chapter 3

# Problems with Classical Module Mechanisms

The *extensibility problem* [40] is concerned with the issue of modular extensibility of structure typically found in application programs. Depending on the programming language and the organization of the code, it is usually straightforward to add either new data types or new operations without changing the original program, but with the price that it is very hard to add the other kind [93].

This chapter gives an illustration of the extensibility problem in the AWT and Swing, two very large libraries written in Java (Section 3.1). The result of the analyses shows how important this extensibility problem is and how deep it can be anchored into mainstream libraries. Then we identify four properties that a packaging system for an object-oriented programming language should support and illustrate these properties through a small link checker application (Section 3.2).

The current state of the art presented in the previous chapter focuses on three approaches to support changes: (i) language constructs to support extensibility, (ii) packaging and deployment of static changes, and (iii) dynamic application of crosscutting changes. Section 3.3 shows three main problems with current approaches on this three domain. These will set up the program research of this thesis.

## 3.1 Swing/AWT Anomalies: Illustration of the Extensibility Problem

Using subclassing to incorporate crosscutting changes often introduces serious drawbacks to support maintainability and evolution such as duplicated code and mismatches between the original and the extended class hierarchy. We illustrate these problems by analyzing Swing [89], the Java standard framework for building GUIs. We first describe the Abstract Window Toolkit (AWT [7]) and its relationships with the Swing framework. Then we show how inheritance is used to share properties between classes. Finally we identify some important drawbacks of the Swing design.

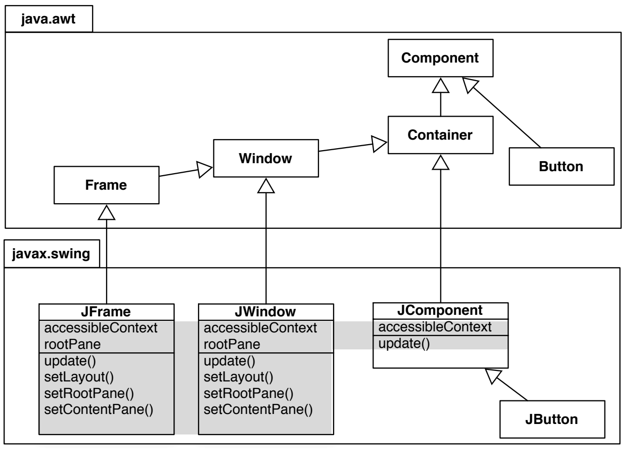


Figure 3.1: Swing is a GUI framework built on top of AWT. Fields and methods shown in JFrame, JWindow and JComponent are duplicated code (gray portion). More than 43% of JWindow is duplicated in JFrame

### 3.1.1 AWT and Swing History

In its first release launched in 1995, Java included AWT 1.0, a framework for building graphical user interfaces. AWT evolved rapidly into version 1.1 to provide a better event handling mechanism. AWT is close to the underlying operating system, therefore only a small number of widgets are supported to make code easier to port. In its latest version AWT consists of 345 classes and contains more than 140,000 lines of code.

Release 1.2 of the Java Development Kit included a completely new GUI framework named Swing. Swing contains 539 classes and more than 260,000 lines of code. This GUI framework is built on top of AWT. It provides a "pluggable look and feel", double buffering and more widgets. A small subset of the core of AWT (Component, Container, Frame Window), and Swing is depicted in Figure 3.1.

In AWT, the root of the graphical widget hierarchy is Component. It provides the essential functionalities of the GUI framework. JComponent is the base class for most of the Swing widgets. The core of Swing is defined by subclassing the core classes of AWT. Each Swing widget can be a container for other widgets, so JComponent inherits from Container. All the widgets except top-level containers (like windows and frames) inherit from JComponent. The classes JFrame and JWindow inherit from Frame and Window, respectively.

The AWT and Swing class hierarchies guarantee certain properties and behavior. In the AWT framework (i) a widget is a component – every widget inherits from Component, (ii) a frame is a window – Frame is a subclass of Window. On the other hand, the Swing framework has the following properties: (i) a Swing widget is not necessary a Swing component because not all of the Swing classes inherit from JComponent, (ii) a Swing frame is an AWT frame and an AWT window: JFrame inherits from Frame which has Window as its superclass, (iii) a Swing window is an AWT window: JWindow inherits from Window.

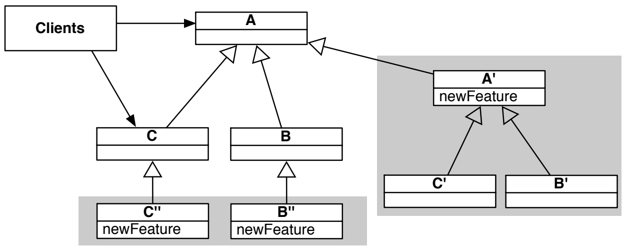


Figure 3.2: Two strategies (gray portions) to introduce changes without impacting existing clients

### 3.1.2 Problem Analysis

Subclassing and refinement relationships are fundamentally different: the former results in a new class containing the incremental changes to its parent class, whereas the latter results in the creation of scope within which the original class is changed. As pointed out by Findler *et al.* [39] and Torgersen [93] under *the extensibility problem*, subclassing does not solve the problem of adding new operations to a class without having to modify or recompile the original program component and its *existing clients*.

In Java, if we wish to extend the class Component by subclassing, without impacting existing clients, we can use either of two strategies (see Figure 3.2): either we build a completely new hierarchy derived from the root of the old hierarchy, duplicating old features in the new hierarchy, or we derive new classes from the leaves of the original hierarchy, duplicating the new features.

Swing illustrates an example of this problem. Swing is built on top of AWT and uses subclassing to extend AWT core classes with Swing functionalities. Since Java supports neither multiple inheritance nor class extension, this design leads, however, to the following severe consequences:

**Duplicated Code.** Due to the absence of an inheritance link between JFrame and JWindow, features defined in JWindow have to be duplicated in JFrame. In Swing, each widget can (i) describe itself (the accessibleContext variable refers to a description of the component) and (ii) support double buffering to provide smooth flicker-free animation (methods update(), setLayout(), . . . ). The source code of JWindow is 551 lines, and JFrame is 829 lines. As a result, 241 lines of code are duplicated between these two classes: 43% of JWindow reappears as 29% of JFrame.

**Breaking Subtyping Inheritance.** Whereas all AWT widgets are AWT components (because they inherit from Component), widgets defined in Swing can either be AWT or Swing components. Furthermore, the Swing design breaks the AWT inheritance relation: while a Window is a Component in AWT, a JWindow is not a JComponent in Swing. While a Button is a Component and JButton is a JComponent, a JButton is not a Button [54].

**Explicit Type Checks and Casts.** A Swing component is a container for other components. This is a feature obtained from Container by inheritance (JComponent is subclass of Container). Therefore types of subcomponents are Component, and not JComponent (the type of the collection of components is Component[]). The following code typifies what happens in Swing components:

```
public class Container extends Component {
  int ncomponents;
  Component components[] = new Component[0];
  public Component add (Component comp) {
     addImpl(comp, null, -1);
     return comp;
  }
  protected void addImpl (Component comp,
                          Object constraints, int index) {
     ...
     component[ncomponents++] = comp;
     ...
  }
  public Component getComponents(int index) {
    return component[index];
  }
}

public class JComponent extends Container {
  public void paintChildren (Graphics g) {
    ...
    for (; i > = 0 ; i--) {
      Component comp = getComponent (i);
      isJComponent = (comp instanceof JComponent);
      ...
      ((JComponent)comp).getBounds();
      ...
    }
  }
}
```

In the Swing framework numerous explicit type checks need to be performed to determine if a subcomponent is issued from Swing or from AWT. For instance, a JComponent needs to know if its subcomponents use double buffering or not. 16 type checks (... instanceof JComponent) and 25 casts to JComponent are performed in JComponent. In the whole Swing library, these numbers rise to 82 and 151, respectively.

Java packages do not provide any mechanism to support unanticipated changes. Once defined, a class cannot be modified or extended without triggering a recompilation, and therefore a global impact on all clients. The rest of this chapter is dedicated to a mechanism named *class extension* allowing new class members to be packaged separately from the declaration of the class.

## 3.2 Key Requirements for Class Extensions

Let us first consider a typical scenario, which will enable us to establish some key requirements for class extensions, while highlighting the main problems to be overcome.

A Link-Checker is an application whose purpose is to report a list of the dead links on a web-page at a given URL. One natural way to implement a Link-Checker, depicted in Figure 3.3, is to download the HTML page from the remote website, and parse it to get an abstract syntax tree of the page composed

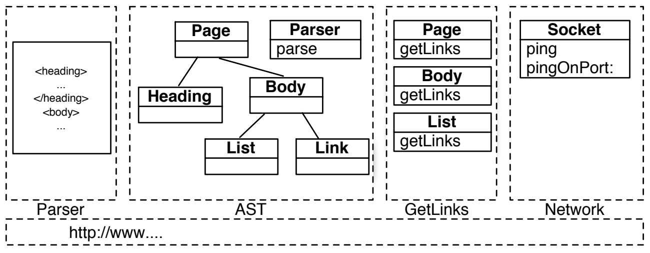


Figure 3.3: The conceptual decomposition of the deadlink checker: an HTML parser, an abstract syntax tree for HTML documents, facility to get links from a page, and a network library

of various elements representing the HTML tags. Then using a recursive call over the hierarchy, get the list of the links referenced in the page. The liveness of each these link elements is checked by pinging the associated host and trying to obtain the status of the linked page. When a timeout is issued or if the HTTP reply corresponds to an error the link is declared dead.

Based on this example we can identify four properties that a packaging system for an object-oriented programming language should support: *class extensions* allowing *redefinition*, *locality* of changes, *propagation* of changes to collaborating clients, and resolution of *conflicts*.

**Class extensions with redefinition.** First, the different elements composing the solution should be packaged so that they can be used in further applications. We can identify the following modules: an HTML scanner and parser, an abstract syntax tree for the HTML elements, a recursive call over these elements to get links contained in a page and some network facilities. One key point is that we have to be able to group together the definitions of the getLinks methods in a module that is different from that of the AST. This means that the GetLinks module has to be able to extend the class definitions of the tree node elements.

Although languages such as CLOS, Smalltalk, MultiJava, and AspectJ offer some solutions, most other languages (including Java), do not allow a class to be extended by a different module or package than the one defining the class. Note that subclassing the tree node elements is not a general solution, since clients that explicitly name the original class will not see the subclass extension.

In our development environment, the default Squeak distribution, the ping method used by the environment does not raise an exception but opens a dialog box when a target host cannot be reached. We therefore not only need the ability to add methods (for packaging the GetLinks module), but also to *redefine* them (to patch existing methods). *We therefore require a module system that supports class extensions with redefinition.*

**Locality of changes.** The second key aspect concerns the visibility of changes, *i.e.*, which modules see the extensions made by other modules. In most approaches that support them, class extensions have a global visibility. All clients have a common view of any given class, and any extensions are also seen by all clients. This may lead to unexpected behaviour for some clients.

In the case of the ping method, we only want our redefined version to be visible within the scope of our application. Other applications may actually rely on the *ad-hoc* behavior provided by ping. Therefore the extensions and changes to the system made by one module should not impact the system as a whole, but only the module introducing the changes and its client modules. *Class extensions should be confined to the module that introduces them.*

**Local rebindings.** Even though class extensions should be visible only to the module that introduces them, the actual effect *from the perspective of that module* should be as if the extension were global.

The pingOnPort: method first adjusts the port (value kept in a variable) and then calls the ping method. We want that any call to ping made by pingOnPort: triggers the definition brought by our LinkChecker application, even if pingOnPort: is defined in a scope that also contains a previous definition of ping. *Class extensions visible within a module should propagate to collaborating clients.*

**Conflicts.** Class extensions are useful when, for instance, a library needs to add a particular method to a class provided by the system. Conflicts arise when an application relies on two modules that extend the same method of the same class in different ways.

The ping method provided by Squeak is useful for pinging a remote host. Its default behavior is to display the result in a popup window. The Link-Checker application redefines this method to make it yield a value and to raise an exception if the host is not reachable. Conflicts can arise with other modules that make changes to this method. As a concrete example, Squeak has a SocketICMP module that implements the ICMP network protocol. Amongst other things, this implementation redefines the ping method with an ICMP-based implementation. Using both the Link-Checker and the SocketICMP module therefore leads to a conflict because both redefine the method ping.

There are several ways to handle this conflict: (1) the definition in Link-Checker overrides the definition in SocketICMP's, (2) SocketICMP's definition overrides Link-Checker's, (3) a conflict is detected at composition time and needs to be resolved, or (4) each extension is defined in a different namespace from that of the class.

With Smalltalk, CLOS and Objective-C the result depends on which module is loaded/initialized last which effectively impacts the system. On the other hand, Multijava and Hyper/J detect conflicting situations at compile time. *Selector namespaces*, Smallscript [85] and ModularSmalltalk [97] define the extension in a particular namespace: conflicts are avoided and both extensions are applied to the system within different scopes. *Resolution of conflicting class extensions should take the context of affected clients into account.*

## 3.3 Class Extension to Support Unanticipated Change

During its life time, an application has to be patched with unforeseen modifications. We characterize this kind of modification as an *unanticipated change*. Despite of their various nature, we focus in this thesis on few kind of changes.

**Refining a class.** Class extensions provide a mechanism to support *unanticipated changes* in a static setting where a class is refined with class member additions and redefinitions. We defined four properties that a package system should have in order to avoid shortcomings found in the Java library analysis. These properties are:

- *Class extensions with redefinition.* A class extension is a definition of class members that are separated from the declaration of the related class. A class extension mechanism has to support redefinition of class members.
- *Locality of changes.* Defining a system change as a set of class extensions does not have to impact original clients of this system that rely on the original definition of it.
- *Local rebindings.* Former code and class extensions have to interact as if extensions were part of the original code. Extensions have to be visible only to the module that introduces them, from the perspective of that module (and other module that rely on the extended classes) have to be as if the extensions were global.
- *Avoiding conflicts.* Applying different extensions on a system that live in different scopes does not have to raise conflict because of the locality of changes.

**Packaging crosscutting changes.** Correcting a bug or adding a feature to a software system often requires refinements to be applied to several classes. By means of a dedicated language, Aspect Oriented Programming [48] (AOP) allow for crosscutting changes. However, AOP technologies are not suitable to cope with software architecture. AOP comes with its own domain application, which is crosscutting changes, however aspects are not suitable to define architectural components mainly because they lack composition operations.

As shown in Chapter 2, Java's and C#'s module system allow for static definition of group of classes, therefore no extension mechanism (as a module operation) is offered. The other module systems presented offer ways to define class extensions, however they do not cope with large refinements over more than one classes.

**Dynamic change load and unload.** Current technologies to apply dynamic changes on a system while it is running come with limitations regarding the kind of the granularity of changes.

Dynamic AOP enables an aspect to be applied on and removed from a base system while it is running. However by supporting only a limited number of join points, aspects are limited in the kind of changes that they represent. For instance, PROSE [77] does not support class extension.

HotSwap and Dynamic class replacement [59, 32] allow for class definition change during their execution. A change application uses a grain level class. Large scale refinements are not supported.

## 3.4 Conclusions

In this chapter, we presented an analysis of the Swing Java library. Our analysis reveals that library contains duplicated code, broken subtype inheritance, and is strewed with explicit type checks and casts. This is mainly a consequence of a gap in adequacy of the Java language to package properly changes and extension that can be defined on a system. This illustrates the need of a class extension mechanism while controlling visibility of changes.

We defined an unanticipated change as an unforeseen modification of a software, usually represented by a "patch". In this thesis, we limit the domain of unanticipated changes to 3 focuses:

1. *Refinement of a class in a static setting.* Supporting class modifications as unanticipated changes in a static setting necessitates for a class extension mechanism to support : (i) class extensions with redefinition, (ii) locality of changes, (iii) local rebinding, and (iv) avoiding conflicts.
2. *Crosscutting refinement.* On the one hand, aspect oriented programming is an approach to define crosscutting concerns, however it is not a means for defining architectural components. On the other hand, module systems are made to support architectural components. While module systems offer ways to define refinements of classes, they do not support change at the granularity of the application, by means of crosscutting refinement.
3. *Dynamic application and removal of change.* Most of the current approaches to support application and removal of software modifications highly limit the domain of the changes.

We restricted the domain of unanticipated changes to three focus. This defines the research program of this thesis. Each point of this program is addressed in the coming chapters.

# Chapter 4

# Classboxes

In this chapter we present a mechanism to support *unanticipated changes* as a refinement of a class in a static setting. Refining a class by adding or redefining class members is the key to address the problems we have identified in the previous chapter. We propose as a solution the classbox model [14]. Classboxes are a module mechanism supporting local class extension.

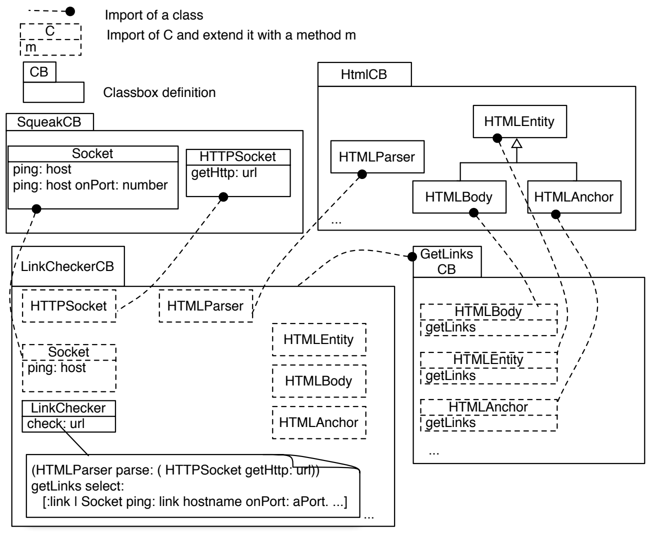
In Section 4.1, we introduce classboxes by means of graphical diagrams and running examples. Then in Section 4.2, we evaluate the classbox model against the problems identified in the previous chapter by proposing a refactoring of the Swing Java library based on the serious anomalies and inconsistencies we revealed during our analysis. Section 4.3 presents a formalization of classbox semantics. And finally, in Section 4.4 we revise the taxonomy presented earlier.

## 4.1 Classboxes in a Nutshell

A *classbox* is a module containing *scoped definitions* and *import statements*. Classboxes define classes, methods and variables. Imported declarations may be *extended*, possibly redefining imported methods. When a classbox is instantiated, it yields a *namespace* in which the directly defined, imported and extended entities co-exist with the implicitly imported entities.

**Scoped Definitions.** A classbox defines *classes*, *methods*, or *variables*. Each class, method or variable *belongs* to precisely one classbox, namely the one in which it is originally defined. Classes and variables defined in a classbox are globally accessible by all methods in the scope of that classbox.

**Imports.** A classbox may import classes and variables from other classboxes. Imported entities thus become available within the scope of the importing classbox. An imported class may be *extended* with new methods, or methods that redefine existing methods. The extended class is then visible within the scope of the extending classbox, but not in the defining classbox of the extended class. Any class visible within a classbox (*i.e.* imported or defined), can be imported from another classbox. The import relationship is therefore transitive.



```
(HTMLParser parse: ( HTTPSocket getHttp: url))
getLinks select:
  [:link | Socket ping: link hostname onPort: aPort. ...]
```

Figure 4.1: The dead-link checker modularized with classboxes

### 4.1.1 Scope of Methods

A method defined on a class in a classbox $CB$ is visible within that classbox, and within other classboxes that import this class from $CB$. In a given classbox all the methods defined along the chain of import are visible within this classbox.

If several classboxes extend a class with a method with the same name but with different implementations, the implementation chosen during an invocation is the one that is reachable according to the import chain.

A classbox $CB$ that defines a method that already exists in the import chain *hides* its former definition from this classbox $CB$ and other classboxes that may import the extended class from $CB$.

### 4.1.2 The Link-Checker with Classboxes

This section shows how to use classboxes to modularize the Link-Checker example. Because classboxes have been fully implemented in the Squeak [46] environment, code fragments are presented in Smalltalk.

The architecture of the Link-Checker application is depicted in Figure 4.1. The classbox SqueakCB contains the network facility for checking the existence of a remote host (class Socket with class method ping: host) and for fetching the content associated to a given URL (class HTTPSocket with class method getHttp: url).

The classbox HtmlCB defines the HTML framework facilities. The class HTMLParser is used to parse a text, yielding an abstract syntax tree (AST) composed of nodes such as HTMLEntity (the root of the structure), HTMLBody, HTMLAnchor (representing a link), . . .

The classbox GetLinksCB implements the recursive algorithm intended to produce a collection of all the links contained over the AST elements. It imports the relevant nodes from the classbox Html-

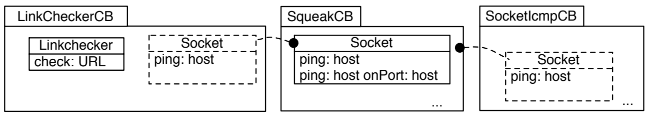


Figure 4.2: The method ping is extended by two different classboxes. Conflict is avoided because extensions are confined to their respective classboxes

CB and *extends* each of the classes representing HTML tag elements by defining the corresponding getLinks methods.

The classbox LinkCheckerCB contains the actual link checker application. It defines the class LinkChecker, containing one method (check: url) which is the entry point of the application. This method first gets the raw content of a page designated by url using the class HTTPSocket). It then parses the page using the class HTMLParser, obtaining an AST of the page. Then it invokes the method getLinks on the root of that AST, obtaining a collection of all the links on the page. Finally it checks the liveness of these links by pinging the hosts mentioned in each link. LinkCheckerCB imports the complete classbox GetLinksCB, so all the extended classes (HTML nodes) are visible within it. As a consequence, within the classbox LinkCheckerCB the AST generated by HTMLParser (class imported from HtmlCB) understands the extensions brought by GetLinksCB. To solve the problem that the method ping: host in the classbox SqueakCB displays its results in a dialog box, the classbox LinkCheckerCB redefines it to raise an exception instead.

### 4.1.3 Discussion

**Locality of Changes.** Although the method ping of class Socket is redefined, its visibility is confined to the LinkCheckerCB classbox. Unrelated code in the system relying on the original definition of this method is not affected. This illustrates both *class extensions with redefinition* and *locality of changes*.

**Local Rebinding.** The classbox SqueakCB defines the class Socket with two methods: ping: host onPort: number and ping: host. The first one calls the second one, and the latter posts a popup menu to display the result of pinging a host. This implementation is not suitable for our application. The classbox LinkCheckerCB imports the class Socket from SqueakCB and extends it by redefining the method ping: host with an implementation that throws an exception when a host is not reachable. Calling ping: host onPort: number within LinkCheckerCB triggers the new implementation of ping: host. This illustrates the *local rebinding* property.

**Conflict.** The classbox LinkCheckerCB extends the class Socket by redefining the method ping. This extension is local to the classbox. Figure 4.2 shows another classbox SocketIcmpCB that also imports the class Socket and redefines the same method ping. This class extension is local to SocketIcmpCB. Conflict is avoided because each extension is confined to the classbox that defines it.

## 4.2 Swing as a Classbox

Because the mechanism provided by Java to specialize code is inheritance, Swing is built on top of AWT using subclassing. As already shown in 3.1 this extension of AWT is developed at a high cost:

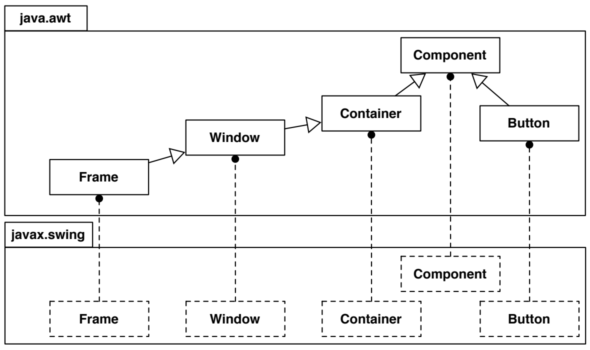


Figure 4.3: An ideal refactoring based on classboxes

(i) properties defined in AWT according to the inheritance property are not valid in Swing anymore (*i.e.* in AWT a Frame is a Window, but in Swing a JFrame is not a JWindow. Not all Swing widgets are JComponent), (ii) a serious amount of code is duplicated to emulate missing inheritance links in Swing (*i.e.* 43% of JWindow is duplicated in 29% of JFrame), and (iii) Swing code is littered with explicit type checks.

Figure 4.3 shows an ideal situation where Swing would be extending AWT using classboxes. Obtaining such a situation would be possible if Swing would have been implemented by following the inheritance tree of AWT (*i.e.* introducing a JContainer class) or if we could afford to perform a complete overhaul of Swing. Since Swing, however, is a large framework with complex logic we cannot rewrite it totally to obtain the situation depicted. In order to illustrate how classboxes offer a working solution, we refactored Swing as a classbox that refines AWT classes. In this section we first describe the new architecture of Swing made out of classboxes, then we present the results obtained, and finally we describe some issues that we encountered while refactoring.

### 4.2.1 Swing Refined from AWT Class

We focus on the refactoring of the core class JComponent, and then we describe how the classbox SwingCB is defined.

**Component refactored in two steps.** The goal of refactoring JComponent is to make the Swing version JComponent a refinement of the AWT version Component. As depicted in 3.1, the class JComponent is a subclass of the AWT classes Container and Component. As Container is an intermediate class between JComponent and Component, the refactoring of the class JComponent is done in two steps, as illustrated in Figure 4.4:

1. *Incorporating the class Container in JComponent.* A Swing component has the ability to contain other components. Features defined by Container have first to be included in JComponent. Container defines 108 methods and 21 fields, however only a few of them have to be duplicated (32 methods related to container management (*e.g.* add, remove) and events management, and 3 variables). We define this "enlarged" JComponent in the classbox SwingCB. This new class is a subclass of Component, which is imported in the new classbox SwingCB. JComponent overrides 22 methods in Container and most of the overriding methods do not perform any super

call. For the methods in JComponent that perform a super call, the two implementations are simply merged.

2. *Making this new JComponent a refinement of Component* The inheritance link between JComponent and the imported Component is replaced by a refinement link.

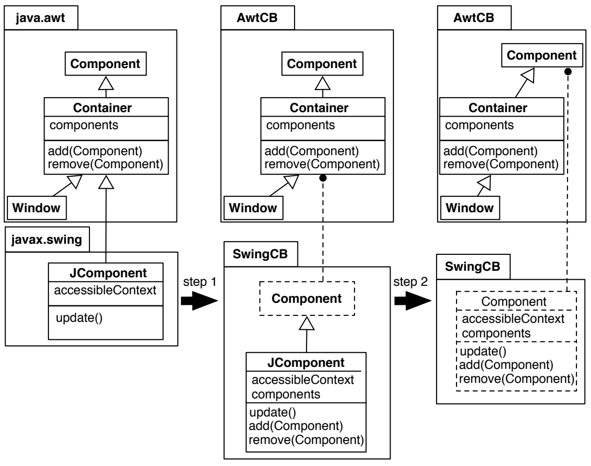


Figure 4.4: The refactoring of the AWT class Component is performed in two steps: (i) the intermediate class is merged to JComponent, then (ii) this merge becomes a refinement of the AWT class Component

**Swing as AWT refined.** Figure 4.5 depicts the new architecture of Swing. Because the definition of a Java package is a valid definition of a classbox, the package java.awt is immediately turned into the AwtCB classbox: no modification is applied to AWT.

The classbox SwingCB imports the class Component, Window, Frame, and Button from AwtCB. These classes are refined with the Swing features.

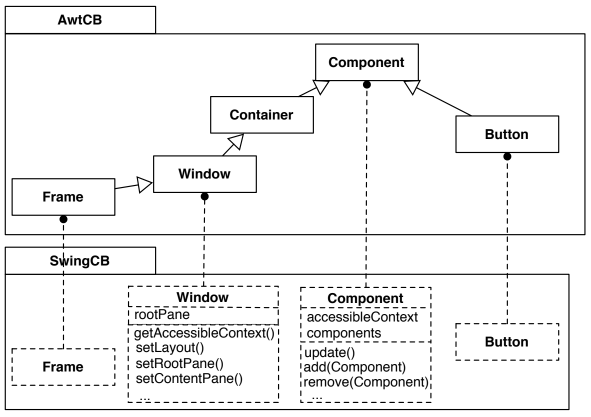


Figure 4.5: Swing refactored as a classbox

### 4.2.2 Advantages with Classboxes

The Swing classes JComponent, JButton, JWindow and JFrame have been refactored as refinement of their AWT counterpart classes. The amount of code refactored is about 6,500 lines of code spread over these 4 classes. Designing Swing with classboxes has several advantages over the original implementation.

**Inheritance coherence.** The inheritance link defined in the AwtCB is fully preserved in the SwingCB. Therefore every Swing widgets, including frames and windows, are swing components. The relation "a frame is a window" stated by AWT is true in SwingCB.

**Removed duplicated code.** JWindow and JFrame are refactored into refinements of Window and Frame. As a result, Frame remains a subclass of Window in Swing and all the duplicated methods and variables related to the layout, root pane and content pane in JFrame are removed. The size of refined Frame is 29% less than the original JFrame.

Because JFrame and JWindow do not inherit from JComponent, the update() method defined by the latter had to be duplicated in JFrame and JWindow. With Swing as a classbox, this duplication is eliminated.

**Explicit type checks avoided.** Within the SwingCB classbox, a Swing component is a Component. Therefore, all the explicit type-checks and casts used in the original Swing to check if a subcomponent is a Component or a JComponent are useless.

Since the checks (... instanceof JComponent) are always true, downcasts from Component to JComponent are simply removed. The 16 type-checks (... instanceof JComponent) and 25 casts to JComponent were removed while refactoring the class JComponent (no such expressions are present in the other refactored classes).

### 4.2.3 Issues and Limits

Now we discuss the results obtained and the impacts on the packages in terms of their visibility.

**Refactoring super calls.** Several methods related to the content management in JWindow like remove(Component) and setLayout( LayoutManager) override methods defined in Window. These methods perform a check on a property of the root pane, then call the original definition using a super call. For instance, the definition of setLayout( LayoutManager manager) in JWindow is:

```
public void setLayout(LayoutManager manager) {
    if (isRootPaneCheckingEnabled()) {
        throw createRootPaneException(”setLayout”);
    }
    else {
        super.setLayout(manager);
    }
}
```

The expression super.setLayout( manager) triggers the implementation defined in the AWT class Window. As this will be explained in Chapter 8, original is a key word part of the classbox system that allows for a previous method to be invoked when it is redefined. Refactoring this overriding method into a refinement of Window implies that the original keyword must be used to invoke the original AWT definition. This scenario convinced us of the need to introduce the original() construct to the classbox model.

**Need to enlarge visibility of some Swing classes.** Replacing the Swing class JComponent by a refinement of Component enlarges the visibility of some classes that were in Swing. For instance, JComponent references Swing classes like AncestorNotifier, which are private to the javax.swing package. Swing classes that were private to Swing need to be visible outside their defining package.

**Limitations of our refactoring.** Unfortunately, removing the class JComponent would entail a major overhaul of Swing. The reason is that each method of the class javax.swing.plaf.ComponentUI refers to the name JComponent. Given our limited resources for this experiment, we confined this overhaul to the classes JWindow, JFrame and JButton. As a consequence, our version of Swing does not contain the pluggable look and feel.

**Execution cost.** With our current implementation of classboxes, the new method lookup semantics is about 22 times slower than the normal one. This result is obtained from triggering 10000 times the update() methods redefined in Component. This loop takes 1008 ms, whereas it is 45 ms for the same method directly implemented in this class. As explained in the following section, our implementation is rather naive. In our work with classboxes in Smalltalk [14], we were able to optimize the implementation so that the cost of the redefined method lookup is only 1.1 times slower (compared to 22 times slower with the Java version).

## 4.3 The Classbox Model

In Chapter 2 we introduced a calculus for the purpose to model various module systems. For each module system, semantics of operators were expressed. The emphasis of this calculus was on providing basic environment operators intended to be used to express high level module operators. Modeling runtime execution was therefore not the focus. For instance, class hierarchies are flattened, class instantiation was not explicitly modeled and the pseudo variable *super* was absent.

This section presents a set-theoretic model that precisely defines the static and runtime semantics of classboxes. We abstract away from the operational details of statements and expressions of a given object-oriented language, and instead focus on the key features that interact with classboxes. We start by introducing a basic model of *classes*, *objects* and *namespaces*, where we capture *instantiation*, *message sending*, and *self*- and *super-calls*.

On top of this basic model, we then show how classboxes are defined as a mechanism for introducing class extensions, and for controlling the visibility of class extensions in different namespaces. We show how *locality of changes* and *local rebinding* arise as a consequence of the way that classboxes are composed.

### 4.3.1 Environments

We use the basic concept of an extensible *environment* as a mechanism for modeling classes, objects and classboxes. Note that Definition 14 and Definition 15 were already presented in Chapter 2.

**Definition 14** An *environment* $\epsilon : D \rightarrow R^\star$, is a mapping from some domain $D$ to an extended range $R^\star = R \cup \{\bot\}$, such that the inverse image $\epsilon^{-1}(R)$ is finite.

We represent environments as finite sets of bindings, for example: $\epsilon_1 = \{a \mapsto x, b \mapsto y\}$ is an environment that maps $a$ to $x$ and $b$ to $y$. All other values in the domain of this environment (for example, $c$) are mapped to $\bot$.

We normally leave out unessential parentheses. Since an environment is a function, we simply invoke it to look up a binding. In this case, $\epsilon_1 a = x$, $\epsilon_1 b = y$ and $\epsilon_1 c = \bot$.

**Definition 15** An environment $\epsilon : D \rightarrow R^\star$ may *override* another environment $\epsilon'$. We define $\epsilon \rhd \epsilon' : D \rightarrow R^\star$ as follows:

$$(\epsilon \rhd \epsilon')x \overset{\text{def}}{=} \begin{cases} \epsilon' x & \text{if } \epsilon x = \bot \\ \epsilon x & \text{otherwise} \end{cases}$$

For example, if $\epsilon_2 = \{b \mapsto z, c \mapsto w\}$, then $(\epsilon_1 \rhd \epsilon_2)a = x$, $(\epsilon_1 \rhd \epsilon_2)b = y$, and $(\epsilon_1 \rhd \epsilon_2)c = w$. We employ overriding both for method dictionaries and class namespaces.

### 4.3.2 Classes, Namespaces and Objects

The primitive elements of our model are the following disjoint sets: $\mathcal{C}$, a countable set of *class names*, $\mathcal{M}$, a countable set of *messages*, and $\mathcal{B}$, a countable set of *method bodies*.

**Definition 16** A *method dictionary*, $\delta \in \mathcal{D}$ is an environment, $\delta : \mathcal{M} \rightarrow \mathcal{B}^\star$ that maps a finite set of messages to bodies.

For example, $\delta = \{m_1 \mapsto b_1, m_2 \mapsto b_2\}$ defines a dictionary $d$ that maps message $m_1$ to body $b_1$ and $m_2$ to $b_2$, and all other messages to $\bot$.

Note that, for the purpose of this chapter, we are not concerned with the implementation details of the method bodies. We only consider which kinds of messages are sent in the bodies.

**Definition 17** A *class*, $\mathsf{c}\langle\delta, B, \epsilon\rangle$ consists of a method dictionary $\delta$, a superclass name $B \in \mathcal{C} \cup \{\mathsf{nil}\}$, and an environment $\epsilon$, called a *class namespace*, that binds class names to classes.

$\mathsf{nil}$ represents an empty class, from which the root of a class hierarchy inherits. By convention, every class namespace is assumed to contain the binding $\mathsf{nil} \mapsto \mathsf{c}\langle\emptyset, \mathsf{nil}, \emptyset\rangle$, which we therefore do not list explicitly.

**Definition 18** An *object* $\mathsf{o}\langle c, \phi\rangle$ consists of a class $c$ and an environment $\phi$, which is a class namespace (obtained from $c$) extended with a binding for $\mathsf{self}$.

Note that, for the present purposes, we do not model attributes (instance variables) of objects, aside from the pseudo-variables $\mathsf{self}$ and $\mathsf{super}$.

We can send messages to classes and to objects. We use the notation $x[m]$ to send the message $m$ to the class or object $x$.

**Definition 19** We can *instantiate* an object by sending the message $\mathsf{new}$ to a class $c = \mathsf{c}\langle\delta, B, \epsilon\rangle$:

$$c[\mathsf{new}] \stackrel{\text{def}}{=} \mu\sigma.\mathsf{o}\langle c, \{\mathsf{self} \mapsto \sigma\} \rhd \epsilon\rangle$$

At this point we recursively bind $\mathsf{self}$ to the value of the object itself.

As usual, $\mu x.E$ binds free occurrences of $x$ in $E$ to the value of the recursive expression itself, *i.e.*, $\mu x.E \stackrel{\text{def}}{=} E\{\mu x.E/x\}$, where $E\{y/x\}$ is the usual substitution operation, replacing free occurrences of $x$ in $E$ by $y$ while avoiding name clashes.

Although we do not model the internal details of method bodies here, we must take care to be precise about the environment within which methods are evaluated. As we shall see when we define classboxes, it is precisely the way in which these environments are composed that determines the scope within which class extensions are visible.

**Definition 20** A *method closure* $\mathsf{m}\langle b, \phi\rangle$ consists of a method body $b$ and a class namespace $\phi$ that additionally binds both $\mathsf{self}$ and $\mathsf{super}$.

Note that $\mathsf{super}$ is bound by methods, not objects, since $\mathsf{super}$-calls are relative to the class in which a method is defined, not the class from which the object is instantiated.

**Definition 21** We can *send a message* $m$ to an object $\mathsf{o}\langle c, \phi\rangle$, where $c = \mathsf{c}\langle\delta, B, \epsilon\rangle$ obtaining a method closure:

$$\mathsf{o}\langle c, \phi\rangle[m] \stackrel{\text{def}}{=} \begin{cases} \mathsf{m}\langle\delta m, \{\mathsf{super} \mapsto \mathsf{o}\langle\epsilon B, \phi\rangle\} \rhd \phi\rangle & \text{if } \delta m \neq \perp \\ \mathsf{o}\langle\epsilon B, \phi\rangle[m] & \text{else if } B \neq \mathsf{nil} \\ \perp & \text{otherwise} \end{cases}$$

This definition captures the basic method lookup algorithm of object-oriented programming languages. If the message sent does not correspond to a method defined in the class of the object, the lookup continues in the parent class, and so on. If the method is not found, the message is reported as not being understood ($\perp$). If a suitable method is found, it is evaluated in a context where $\mathsf{super}$ is bound to the current object, but from the perspective of the method's superclass. As we can clearly see, $\mathsf{super}$ is an object, not a class. Note that according to Definition 17 the superclass $B$ can be nil.

**Definition 22** A closure may be evaluated, in which case it may send various messages. Here we are interested in $\mathsf{self}$- and $\mathsf{super}$-sends, and static class references.

$$\begin{aligned} \mathsf{m}\langle b, \phi\rangle[\![\mathsf{self}\ m]\!] &\stackrel{\text{def}}{=} (\phi\ \mathsf{self})[m] \\ \mathsf{m}\langle b, \phi\rangle[\![\mathsf{super}\ m]\!] &\stackrel{\text{def}}{=} (\phi\ \mathsf{super})[m] \\ \mathsf{m}\langle b, \phi\rangle[\![C\ \mathsf{new}]\!] &\stackrel{\text{def}}{=} (\phi\ C_\phi)[\mathsf{new}] \end{aligned}$$

### 4.3.3 Classboxes

A classbox is an *open* entity that provides a number of classes, and which can be extended. When a classbox is *closed*, it yields an ordinary class namespace (Definition 17).

The key point in modeling classboxes is that multiple versions of the same class may be implicitly present within the same classbox. Suppose that we import the class LinkChecker from the classbox LinkCheckerCB, and we locally define a class Socket. Even though LinkChecker collaborates with Socket, ours is a *different* socket class that has nothing to do with the Socket class known to LinkChecker. To capture this aspect we must refine the notion of class names to express the *originating classbox* to which a class belongs:

- $\mathcal{C}$ is the countable set of *raw class names*,
- $\mathcal{X}$ is the set of classbox names,
- $\mathcal{C}^+ = \{C^n | C \in \mathcal{C}, n \in \mathcal{X}\}$ is the set of *decorated class names*.

The *decorated class name* simply encodes the classbox to which the class belongs, *i.e.*, where it was first defined. We call the superscript $n$ of a decorated class name $C^n$ its *origin*.

**Definition 23** A raw class name $C$ *matches* a decorated class name $B^n$ if $C = B$:

$$C \sim B^n \textit{ iff } C = B$$

For example, when we use the raw class name Socket, it may not be clear *which* Socket class we are referring to. However the decorated class name $\text{Socket}^{\text{SqueakCB}}$ unambiguously identifies the Socket class first introduced in the SqueakCB classbox.

Note that it is this *same* class that is extended in LinkCheckerCB, since there is no Socket class defined there. There is no $\text{Socket}^{\text{LinkCheckerCB}}$.

**Definition 24** A *classbox* $\mathsf{b}\langle n, \alpha\rangle$ consists of an identifier $n \in \mathcal{X}$ (*i.e.* classbox names) and a function $\alpha$ from class namespaces to class namespaces.

The intuition here is that a classbox is *open* because it can always be extended with new class definitions, imports and extensions. As a consequence, we do not yet know the class namespace of the classes it provides. However we can *close* a classbox, thereby fixing the class namespace of all the provided classes.

**Definition 25** A *classbox* $\mathsf{b}\langle n, \alpha\rangle$ can be *closed* by sending it the close message, generating a fixpoint: $\mathsf{b}\langle n, \alpha\rangle[\mathsf{close}] \stackrel{\text{def}}{=} \mu\epsilon.\alpha\epsilon$

The resulting class namespace must be closed, *i.e.*, all used class names must be defined. Since $\alpha$ is a function from class namespaces to class namespaces, $\mu\epsilon.\alpha\epsilon$ represents a fixpoint in which all the classes provided by the classbox are made visible to each other.

**Definition 26** We may *lookup* the decorated class name $C^n$ corresponding to a raw class name $C$ in a classbox $\mathsf{b}\langle n, \alpha\rangle$:

$$C_\alpha \stackrel{\text{def}}{=} \begin{cases} C^n & \text{if } \exists! n \in \mathcal{X}, (\mathsf{b}\langle n, \alpha\rangle[\mathsf{close}])C^n \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

Suppose the $\mathsf{LinkCheckerCB}$ classbox is represented by $\mathsf{b}\langle\mathsf{LinkCheckerCB},\alpha\rangle$. Then $\mathsf{Socket}_\alpha$ yields $\mathsf{Socket}^{\mathsf{SqueakCB}}$, since SqueakCB is the origin of Socket in the LinkCheckerCB classbox.

**Definition 27** An *empty classbox* with identifier $n$ is: *empty*$(n) \overset{\text{def}}{=} \mathsf{b}\langle n, \lambda\epsilon \rightarrow \emptyset\rangle$. Note that $\mathit{empty}(n)[\mathsf{close}] = \emptyset$, *i.e.*, closing an empty classbox yields an empty class namespace.

**Definition 28** We can *introduce* to a classbox $\mathsf{b}\langle n,\alpha\rangle$ a new class $C$ that subclasses $B$ (defined in a classbox $\mathsf{b}\langle m,\beta\rangle$) with $\delta$ as method dictionary by sending it the message def subclasses with.

$$\mathsf{b}\langle n,\alpha\rangle[\mathsf{def}\ C\ \mathsf{subclasses}\ B^m\ \mathsf{with}\ \delta]$$

$$\overset{\text{def}}{=} \begin{cases} \mathsf{b}\langle n, \lambda\epsilon.\{C^n \mapsto \mathsf{c}\langle\delta, B^m, \epsilon\rangle\} \rhd \alpha\epsilon\rangle, & \text{if } C_\alpha = \bot \\ \bot & \text{otherwise} \end{cases}$$

Note that the formal parameter $\epsilon$ represents the fixpoint we obtain when the classbox is finally closed. We must therefore extend $\alpha\epsilon$ with the new subclass definition, obtaining $\{C^n \mapsto \cdots\} \rhd \alpha\epsilon$. We retain $\epsilon$ as a formal parameter so that the classbox remains open (*i.e.*, $\lambda\epsilon.\cdots$). The side condition states that it is an error to introduce a class that is already defined in the classbox. Within a classbox, only decorated class names occur. The newly introduced class has the origin $n$. We also explicitly identify the origin $m$ of the superclass.

### 4.3.4 Importing Classes

**Definition 29** A classbox $\mathsf{b}\langle n,\alpha\rangle$ may *import* a raw named class from another, classbox $\mathsf{b}\langle m,\beta\rangle$, by sending it the message import.

$$\mathsf{b}\langle n,\alpha\rangle[\mathsf{import}\ C\ \mathsf{from}\ \mathsf{b}\langle m,\beta\rangle]$$

$$\overset{\text{def}}{=} \begin{cases} \mathsf{b}\langle n, \lambda\epsilon.\{C_\beta \mapsto (\mu\phi.\beta(\epsilon \rhd \phi))C_\beta\} \rhd \alpha\epsilon\rangle, & \text{if } C_\alpha = \bot \\ \bot & \text{otherwise} \end{cases}$$

Let us call the new classbox we obtain $\mathsf{b}\langle n,\alpha'\rangle$. $\alpha'$ extends $\alpha$ with the imported definition, but we must also take care that the environment of the imported class is properly extended with any pertinent definitions that occur in $\alpha'$. As before, $\epsilon$ represents the class namespace that we obtain when we take the fixpoint of $\alpha'$. We therefore pass $\epsilon$ to $\alpha$ so it is available to all the existing class definitions in $\alpha$. We must also look up the correct decorated class name $C_\beta$. Finally, we must bind this to the correct definition from $\beta$, *extended* with any new definitions from $\alpha'$.

Suppose we would simply use $C_\beta \mapsto (\mu\phi.\beta\phi)C_\beta$. This would clearly be wrong, because the class we obtain would only see other class definitions from $\beta$, and not any definitions that may have already been extended in $\alpha$. Instead, we create an *intermediate namespace* $\mu\phi.\beta(\epsilon \rhd \phi)$. $\epsilon \rhd \phi$ represents the environment of $\beta$ *extended with any new definitions from* $\alpha'$. We then pass this into $\beta$ to make it available to *all* class definitions in $\beta$. Finally we extract this definition, bind it to $C_\beta$ and use it to extend $\alpha\epsilon$.

Consider, for example, the import relationships in Figure 4.1. The classbox LinkCheckerCB imports HTMLParser from HtmlCB and HTMLEntity and its subclasses from GetLinksCB. If HTMLParser were naively imported from HtmlCB, it would not see the extensions imported from GetLinksCB.

Instead, the import operation is defined so that when HTMLParser is imported, its environment (*i.e.*, $\phi$) is extended by all definitions in LinkCheckerCB (*i.e.*, $\epsilon \rhd \phi$). So when HTMLParser is imported, it sees the extended versions of HTMLEntity and its subclasses. This is the *local rebinding* mechanism of classboxes.

Note that it is critical that HTMLEntity imported from GetLinksCB has the same origin as that expected by HTMLParser. If LinkCheckerCB or GetLinksCB were to define a *new* class HTMLEntity, then this would have a different decorated class name from the HTMLEntity originally defined in HtmlCB, and would therefore be invisible to HTMLParser.

### 4.3.5 Extending Imported Classes

**Definition 30** A classbox $\mathsf{b}\langle n,\alpha\rangle$ may *extend* a raw class named class from another classbox $\mathsf{b}\langle m,\beta\rangle$, by sending it the message extend with.

$$\mathsf{b}\langle n,\alpha\rangle[\text{extend } C \text{ with } \delta' \text{ from } \mathsf{b}\langle m,\beta\rangle]$$

$$\stackrel{\text{def}}{=} \begin{cases} \mathsf{b}\langle n, \lambda\epsilon.\{C_\beta \mapsto \delta' \rhd (\mu\phi.\beta(\epsilon \rhd \phi))C_\beta\} \rhd \alpha\epsilon\rangle & \text{if } C_\alpha = \bot \\ \bot & \text{otherwise} \end{cases}$$

where

$$\delta' \rhd \mathsf{c}\langle \delta, B, \epsilon\rangle \stackrel{\text{def}}{=} \mathsf{c}\langle \delta \rhd \delta, B, \epsilon\rangle$$

Extend works just like import, except that the imported class definition is extended with $\delta'$.

As a consequence, importing a class is the same as extending it with a nil extension:

$$\mathsf{b}\langle n,\alpha\rangle[\text{import } C \text{ from } \mathsf{b}\langle m,\beta\rangle] \;\equiv \mathsf{b}\langle n,\alpha\rangle[\text{extend } C \text{ with } \emptyset \text{ from } \mathsf{b}\langle m,\beta\rangle]$$

As should be clear from the definition, class extensions are purely local to the classbox making the extension. This guarantees *locality of changes*. Extensions become visible to other classboxes only when they are explicitly imported, or implicitly made visible by the mechanism of local rebinding (as seen in the HTMLParser example discussed above).

*Method redefinition* is supported since the $\delta'$ introduced by a class extension can redefine methods existing in the class being extended. For example, not only can the GetLinksCB classbox extend the HTMLEntity and related classes with a new getLinks method, but the LinkCheckerCB classbox can import Socket from the SqueakCB classbox and *redefine* the ping method.

### 4.3.6 Proving Classbox Properties

**Proposition 1** A method defined in a classbox is visible within this classbox.

**Proof.** Because a method is defined either when a class is (i) defined or (ii) imported, this Proof is divided in two parts.

(i) Methods defined at the same time than the class they refer to are visible within the classbox where they are effectively defined. This first part of the proof consists in showing that defining a class $C$

with a method $\mathsf{m}$ bound to a compiled method $\mathsf{CM}$ makes this method visible within the classbox (*i.e.* invoking $m$ on an instance of $C$ triggers the expected method $\mathsf{CM}$).

Without loss of generality, assume that $C$ has no superclass (*i.e.* it inherits from $\mathsf{nil}$).

$$\begin{aligned} &\mathsf{b}\langle n,\alpha\rangle[\mathsf{def}\ C\ \mathsf{subclasses\ nil\ with}\ \{\mathsf{m}\mapsto\mathsf{CM}\}] \\ &=\ \mathsf{b}\langle n,\lambda\epsilon.\{C^n\mapsto\mathsf{c}\langle\{\mathsf{m}\mapsto\mathsf{CM}\},\mathsf{nil},\epsilon\rangle\}\rhd\alpha\epsilon\rangle\ =\ \mathsf{b}\langle n,\alpha'\rangle \end{aligned}$$

Closing this classbox yields:

$$\mathsf{b}\langle n,\alpha'\rangle[\mathsf{close}]=\mu\epsilon.\alpha'\epsilon=\varphi=\{C^n\mapsto\mathsf{c}\langle\{\mathsf{m}\mapsto CM\},\mathsf{nil},\varphi\rangle\}\rhd\alpha\varphi$$

Now the instance of this class $C^n$ is $\mathsf{obj}=\mathsf{o}\langle\varphi C^n,\{\mathsf{self}\mapsto\mathsf{obj}\}\rhd\varphi\rangle$. Sending a message $\mathsf{m}$ to it yields:

$$\mathsf{obj}[\mathsf{m}]=\mathsf{m}\langle\{\mathsf{m}\mapsto\mathsf{CM}\}\mathsf{m},\{\mathsf{super}\mapsto\mathsf{c}\langle\emptyset,\mathsf{nil},\emptyset\rangle\}\rhd\varphi\rangle$$

The implementation identified for the method $\mathsf{m}$ is the result of
$\{\mathsf{m}\mapsto\mathsf{CM}\}\mathsf{m}=\mathsf{CM}$.

(ii) Methods defined when importing a class are visible within the importing classbox.

$$\begin{aligned} &\mathsf{b}\langle n,\alpha\rangle[\mathsf{extend}\ C\ \mathsf{with}\ \{\mathsf{m}\mapsto\mathsf{CM}\}\ \mathsf{from\ b}\langle m,\beta\rangle] \\ &=\ \mathsf{b}\langle n,\lambda\epsilon.\{C^n\mapsto\mathsf{c}\langle\{\mathsf{m}\mapsto\mathsf{CM}\},\mathsf{nil},\epsilon\rangle\}\rhd(\mu\phi.\beta(\epsilon\rhd\phi))C_\beta\}\rhd\alpha\epsilon\rangle \end{aligned}$$

Assuming that $C_\beta=C^p$ closing this classbox yields:

$$\mathsf{b}\langle n,\ldots\rangle[\mathsf{close}]\ =\varphi=\{C^p\mapsto\mathsf{c}\langle\{\mathsf{m}\mapsto CM\},\mathsf{nil},\varphi\rangle\}$$

The rest of the proof follows what is already shown in (i).

**Proposition 2** Importing a class makes its methods previously defined visible in the importing classbox.

**Proof.** If $\mathsf{b}\langle m,\beta\rangle C_\beta=\mathsf{c}\langle\{\mathsf{m}\mapsto\mathsf{CM},\},B\rangle\epsilon$ then

$$\begin{aligned} &\mathsf{b}\langle n,\alpha\rangle[\mathsf{import}\ C\ \mathsf{from\ b}\langle m,\beta\rangle] \\ &=\ \mathsf{b}\langle n,\lambda\epsilon.\{C_\beta\mapsto(\mu\phi.\beta(\epsilon\rhd\phi))C_\beta\}\rhd\alpha\epsilon\rangle \end{aligned}$$

Assuming that $C_\beta=C^p$ closing the resulting classbox yields:

$$\mathsf{b}\langle n,\ldots\rangle[\mathsf{close}]\ =\varphi=\{C^p\mapsto\beta C^p\}=\{C^p\mapsto\mathsf{c}\langle\{\mathsf{m}\mapsto\mathsf{CM}\},B,\varphi\rangle\}\rhd\alpha\varphi$$

Then as already shown in the first proof, sending a message $m$ to an instance of $\varphi C^p$ triggers the execution of $\mathsf{CM}$.

**Proposition 3** Within a classbox, a method redefinition takes precedence over its former implementation.

**Proof.** Within a classbox $\mathsf{b}\langle m, \beta\rangle$ a class $C$ has in its method dictionary an entry $\mathsf{m}$ bound to a first implementation $\mathsf{CM1}$. This proof consists in showing that importing $C$ in another classbox and redefining $\mathsf{m}$ bound to $\mathsf{CM2}$ hides the former implementation.

If $\mathsf{b}\langle m, \beta\rangle C_\beta = \mathsf{c}\langle\{\mathsf{m} \mapsto \mathsf{CM1}\}, B, \epsilon\rangle$ then

$$\begin{aligned}&\mathsf{b}\langle n, \alpha\rangle[\mathsf{extend}\ C\ \mathsf{with}\ \{\mathsf{m} \mapsto \mathsf{CM2}\}\ \mathsf{from}\ \mathsf{b}\langle m, \beta\rangle]\\ =\ &\mathsf{b}\langle n, \lambda\epsilon.\{C_\beta \mapsto \{\mathsf{m} \mapsto \mathsf{CM2}\} \rhd (\mu\phi.\beta(\epsilon \rhd \phi))C_\beta\} \rhd \alpha\epsilon\rangle\end{aligned}$$

Assuming that $C_\beta = C^p$ closing the resulting classbox yields:

$$\mathsf{b}\langle n, \ldots\rangle[\mathsf{close}] = \varphi = \{C^p \mapsto \{\mathsf{m} \mapsto \mathsf{CM2}\} \rhd \beta C^p\} =$$

$$\{C^p \mapsto \{\mathsf{m} \mapsto \mathsf{CM2}\} \rhd \mathsf{b}\langle \mathsf{m} \mapsto \mathsf{CM1}, B\rangle\varphi\} =$$

$$\{C^p \mapsto \mathsf{c}\langle\{\mathsf{m} \mapsto \mathsf{CM2}\}, B, \varphi\rangle\}$$

The conclusion of this proof follows the end of the very first proof. Instantiating $C^p$ and sending the message $\mathsf{m}$ executes the new implementation $\mathsf{CM2}$.

### 4.3.7 Resolving Diamond Conflicts

*Conflicts* are largely avoided. Classes that coincidentally have the same name but are introduced in different classboxes do not conflict because they have separate origins. Contradictions arising from attempts to import the same class from different classboxes of course cannot be resolved automatically. However, an important class of *indirect* conflicts is automatically resolved by the nature of the local rebinding mechanism.

Figure 4.6 illustrates a diamond pattern arising from two import chains with a common ancestor class. Classbox CB1 defines a class A which provides a method foo returning the value 1. This class is imported by CB2 where the method foo is redefined to return 2. CB2 also defines a subclass of A named B. In a similar way, classbox CB3 imports A from CB1 and redefines foo to return 3. A subclass of A named C is also defined. A fourth classbox CB4 imports B from CB2 and C from CB3. CB4 does not explicitly import class A.

In the context of CB4 invoking foo on an instance of B yields the value 2, whereas invoking foo on an instance of C yields 3. However, if CB4 would explicitly import A from any one of CB1, CB2 or CB3, then that version of A would be visible to both B and C. For example, if CB4 would import A from CB1 and redefine foo to return 4, then both instances of B and C would return 4 when foo is invoked.

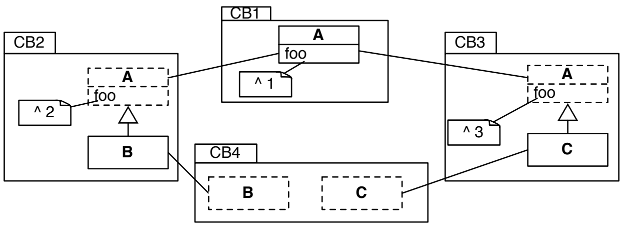


Figure 4.6: Resolving Diamond conflicts

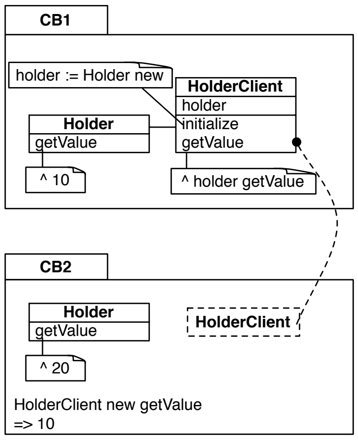


Figure 4.7: Associations are preserved

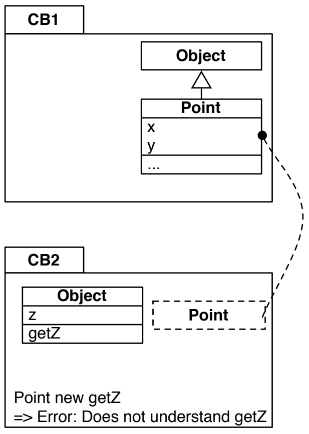


Figure 4.8: Inheritance is preserved

### 4.3.8 Interclass Relationship Preserved

A classbox is a namespace and defines a scope for contained definitions. Importing a class increases visibility for a particular class to other classboxes. However, relationships between classes like association and inheritance are preserved while importing classes.

**Preserving inter-class associations.** As stated in Section 4.1.1, an imported class may be extended with new methods, or methods that redefine existing methods. The extended class is then visible within the scope of the extending classbox, but not in the defining classbox of the extended class. As a consequence of the *local rebinding* facility, an extended version of a class is used in place of its previous versions, even if some code that rely on a previous version are invoked.

However, a new class definition does not stand for a new version of another class. For instance, Figure 4.7 describes a classbox CB1 containing two classes, Holder and HolderClient. Holder has a method getValue returning 10, and HolderClient defines a method getValue that invokes getValue on a holder. Performing the expression HolderClient new getValue within this classbox returns 10. Another classbox, CB2, imports HolderClient from CB1 and defines a **new** class Holder. Because this new Holder is a new class and not an extension of the original class provided by CB1, Holder is **not** locally rebound in CB2. Performing the expression HolderClient new getValue within CB2 returns 10.

**Preserving inheritance while importing.** As described above, creating a class does not affect the

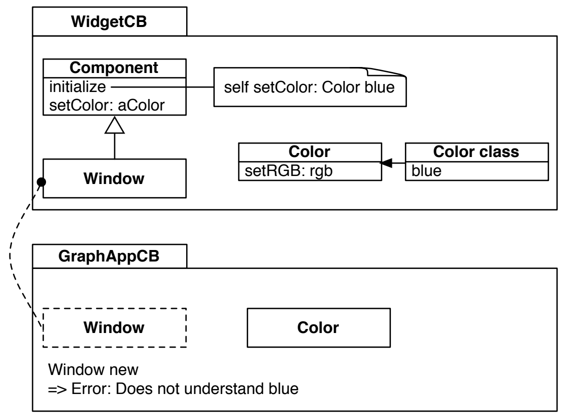


Figure 4.9: Unintended class override: In GraphApp, the class Window cannot be instantiated because of the new class Color

hierarchy of imported classes. Figure 4.8 shows a classbox CB1 defining a class Object and a subclass Point. This class is imported into another classbox CB2. CB2 also defines a new class Object that defines a getZ method. As creating a new class does not hide previous class definitions, inheritance of the imported class is not impacted by local class definition.

**Other alternative.** One alternative to the lookup of version described above is to use a lookup of class definitions. A reference to a class triggers a lookup of a class following import links. A new class definition named C overrides previous definition (or extension) of classes that have the name C. For instance, on Figure 4.7, instantiating a class HolderClient in the classbox CB2 triggers the method initialize. This alternative class lookup makes the definition of Holder provided by CB2 used when initialize of HolderClient is triggered.

In Figure 4.8 the new definition of Object in CB2 overrides the class with the same name in CB1. The imported class Point has, therefore, the new Object class as superclass. On the example, Point new getZ returns the value associated to z.

Using a lookup for class may lead to unintended class capture. Figure 4.9 illustrates this by defining a classbox WidgetCB containing a class Component. This class has a method initialize that sets the default color of a component. It gets an instance of the class Color by sending the message blue to it. The classbox GraphAppCB imports the class Window and create a new class Color that does not define the static method blue. As a consequence, instantiating Window within GraphAppCB leads to an error because the initialize methods uses the local implementation of the class Color, therefore the message blue is not understood. The class Color in GraphAppCB is an unintended override of the previous definition.

## 4.3.9 Import Explicitly Stated

Classboxes allow a class to have multiple versions at the same time, therefore an object can have different behaviors according to the context (defined by a classbox) in which it is used. In order to send messages to an object, the class of this object has to be visible (*i.e.* imported or defined) in the classbox in which it is used. For instance, if within a classbox, the message getX is sent to an instance

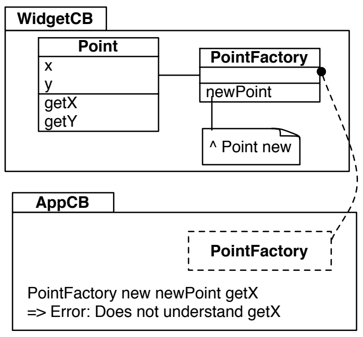


Figure 4.10: Situation where a class is not visible

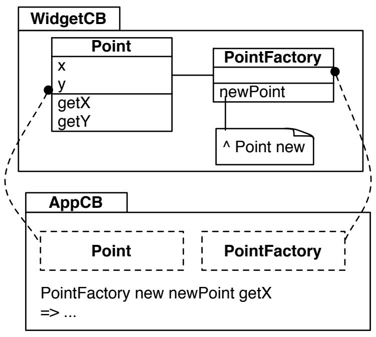


Figure 4.11: By importing Point the message getX is understood

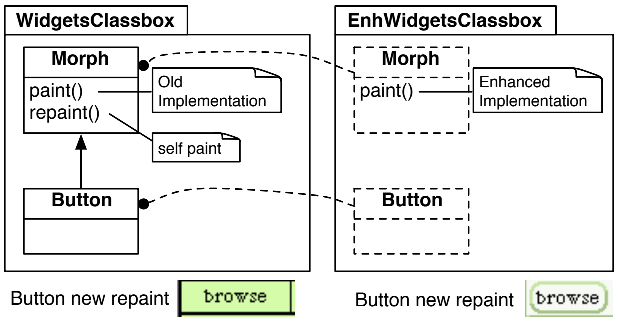


Figure 4.12: Implicitly rebinding classes within classboxes

of the class Point, this class has to be visible. If Point is not visible, then a runtime error occurs when sending a message to a point. This situation is illustrated in Figure 4.10. A new point is obtained from executing PointFactory new newPoint, and sending the message getX raises an error because the class Point is not visible within the classbox WidgetCB.

A correct version is shown in Figure 4.11. The class Point is now imported in AppCB, therefore the expression PointFactory new newPoint getX returns the default value of x (note that the initialization is not shown on the figure).

## 4.4 Classboxes in the Taxonomy

Classboxes [14, 16] is a module system that supports *local rebinding*. It allows a class defined in one particular classbox to be extended by means of method addition or redefinition in other classboxes. Moreover, the changes made by a classbox are *only* visible to that classbox and classboxes that import it. However, when methods of this class are invoked from the extending classbox, changes are visible from methods defined in the provider classbox (*i.e.* where the class is imported from).

The following example illustrates a method extension with local rebinding [14]. Figure 4.12 depicts a classbox WidgetsClassbox that defines a class Morph, which is the root of the Squeak graphic element hierarchy, and a subclass Button. Morph contains a paint() method and a repaint() that calls paint(). The classbox EnhWidgetsClassbox imports Morph and redefines the paint() method. It also imports the subclass Button. In the context of WidgetsClassbox, invoking the repaint() method on an instance of Button invokes the definition of paint() in Morph defined by this classbox. Within EnhWidgetsClassbox, invoking repaint() triggers the new implementation of paint() defined in this classbox.

Within our calculus the semantics of classboxes are expressed using the $extend$ operation. Within a classbox, it imports a class defined in another classbox and extends it with a set of definitions. This operator is defined as:

$$\begin{array}{rcl} \textit{extend} & : & \mathcal{M} \to \mathcal{M} \to \mathcal{C} \to \mathcal{D} \to \mathcal{M} \\ \textit{extend} & = & \lambda m_t.\ \lambda m_s.\ \lambda c.\ \lambda d.\ \lambda \epsilon.\ m_t\ \epsilon \| \{c^{m_s} \mapsto d \rhd\ (m_s\ \epsilon\ \ c)\} \end{array}$$

For instance, the extension between EnhWidgetsClassbox and WidgetsClassbox is stated:

*EnhWidgetsClassbox = extend* $\lambda$self. $\{\}$ *WidgetsClassbox Morph* $\{$*paint* $\mapsto \ldots\}$

The superscript (*e.g.* $c^{m_s}$) is used to identify the *originating classbox* in which a class is first defined. This makes it possible to distinguish classes that are defined from those that are imported and extended, even if they have the same name. Let's suppose that classbox WidgetsClassbox defines two classes Morph and Button, where Button makes use of Morph. If a classbox EnhWidgetsClassbox imports Button from WidgetsClassbox and defines a new class Morph, then this new class has nothing to do with the Morph originating in WidgetsClassbox and should not affect the imported Button class. The superscript identifying the originating classbox ensures that no confusion will result. If, on the other hand, EnhWidgetsClassbox imports and extends Morph from WidgetsClassbox, then this extended Morph will have the same originating classbox superscript, and will affect the imported Button class.

In the same way, new classes are defined as

$$\begin{array}{rcl} \textit{newClass} & : & \mathcal{M} \to \mathcal{C} \to \mathcal{C} \to \mathcal{D} \to \mathcal{M} \\ \textit{newClass} & = & \lambda m_t.\ \lambda sup.\ \lambda c.\ \lambda d.\ \lambda \epsilon.\ m_s\ \epsilon \| \{c^{m_t} \mapsto d \rhd \epsilon\ sup\} \end{array}$$

A shortcut to extend a class with an empty set of definition is stated as:

$$\begin{array}{rcl} \textit{import} & : & \mathcal{M} \to \mathcal{M} \to \mathcal{C} \to \mathcal{M} \\ \textit{import} & = & \lambda m_t.\ \lambda m_s.\ \lambda c.\ \textit{extend}\ m_t\ m_s\ c\ \{\} \end{array}$$

### 4.4.1 Local Rebinding

The *local rebinding* property is provided by an extension mechanism when extensions are visible by and see former definitions of the code. A change defined by some class extensions can use the former definitions, and former definitions can use the new extensions.

Classboxes offer a *local rebinding* property because within the *import* statement a *fix* operation is not performed on the classbox from which a class is imported. In gbeta and MixJuice, the *extend* operator used to express the class extension with classboxes is:

$$\textit{extend} = \lambda m_t.\ \lambda m_s.\ \lambda c.\ \lambda d.\ \lambda \epsilon.\ m_t\ \epsilon \| \{c^{m_s} \mapsto d \rhd\ (m_s\ \epsilon\ \ c)\}$$

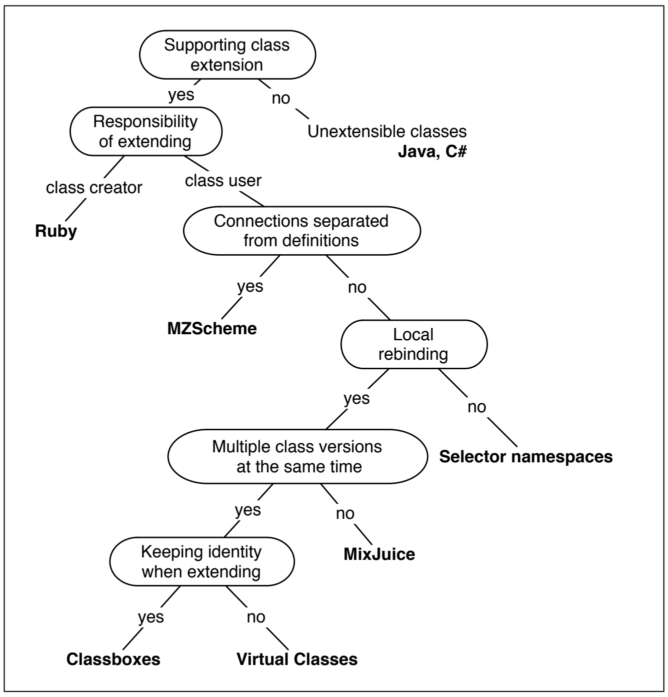

Supporting class extension
yes
no
Responsibility of extending
Unextensible classes
Java, C#
class creator
class user
Ruby
Connections separated from definitions
yes
no
MZScheme
Local rebinding
yes
no
Multiple class versions at the same time
Selector namespaces
yes
no
Keeping identity when extending
MixJuice
yes
no
Classboxes
Virtual Classes

Figure 4.13: Classboxes added to the taxonomy

The extended class is the value given by $d \rhd \ m_s \ \epsilon \ c$. The extensions $d$ override the definition of $c$ obtained from $m_s \ s$ (the module from which the class is obtained). Methods originally defined in $m_s \ s \ c$ can call methods defined in $d$. As no *fix* operation is involved, the class definition in the parent encapsulating class can introduce new refinements.

### 4.4.2 Multiple Class Versions at the Same Time

A class defined in a classbox can be refined in another classbox without conflicting with the original definition. This is a result of allowing multiple versions of the same class to coexist in the same system. Each version of a given class can have different collaborating classes present in the same system.

MixJuice offers the possibility of extending a system by defining differential modules. Such modules are then composed to form a executable system. However, one strong constraint is that only one particular version of a class can be present in a system. Therefore, a particular combination of modules may lead to some unexpected results because some modifications might be propagated to clients that rely on the original version only.

One current limitation of our formalism is that the restriction of having only one particular version of a class present in a system is not reflected by the operators described above. The notion of executable system is not defined, therefore no restriction related to the execution can be formulated.

### 4.4.3 Identity of the Extended Classes

Refining a class by subclassing it does not preserve class identity: the original and refined definitions are implemented by two distinct classes. With virtual classes (Section 2.1.6), a class is refined by creating a new class that substitutes the first one when a class lookup is performed. As a consequence, an instance of a class is not an instance of the refined class.

With classboxes, new methods can be added or redefined on an imported class. The new methods are part of the class behavior but they are only visible from the context of the classbox that defines them. As a consequence, these methods are only accessible in this classbox and in other classboxes that import the class from the extending classbox. The identity of the class is preserved. As a consequence, the set of methods understandable by an instance of a class created by a classbox $CB1$ may be enlarged if this instance is referenced by some code in a classbox $CB2$. This is expressed in our calculus by making the originating classbox explicit by means of a superscript (*e.g.* $c^{m_s}$).

## 4.5 Conclusion

This chapter outlines the classbox system. Properties of this module system are first informally described using an example and then formalized using a simple calculus. Subsequently, this module system is applied to refactoring Swing. The benefits of classboxes are multiple: duplicated code and broken extended hierarchy present in the original Swing are removed. Chapter 2 describes a classification of various module systems presented in a taxonomy. We extended this taxonomy with a new discriminating property ("keeping identity when extending") and where the classbox model is then inserted.

In the next chapter, we present an efficient implementation of classboxes in a dynamically typed environment.

# Chapter 5

# Implementation

Classboxes are a new language construct offering powerful capabilities to package class extensions. The precedent chapter presented a model of the classbox and discussed its properties. Inserting classboxes into a programming environment implies (i) the need for development tools to be classbox aware and (ii) the execution layer to behave in accordance with the classbox semantics. This chapter focuses on the latter point, and shows an implementation of classboxes in a dynamically typed language.

The local rebinding property (Section 4.4.1) fully describes the dynamic behavior of classboxes. But to implement this property a new semantics of the method lookup has to be defined. Section 5.1 provides a description of the new method lookup algorithm. An important property of this algorithm is that *import takes precedence over inheritance*. This is illustrated in Section 5.2.

Concretely, there are two different strategies to modify the method lookup: either the virtual machine has to be modified, or reflective facilities have to be used (bytecode manipulation and message passing control):

- Classboxes can be implemented by changing the method lookup algorithm of the virtual machine. This strategy is explained in Section 5.3. We adapted the method lookup and compiled a new virtual machine that is classbox-aware.
- Section 5.4 describes the second strategy to implement classboxes using bytecode manipulation and some reflective features of Smalltalk such as a reification of the method call stack.

We evaluate the impact of this extended method lookup algorithm on performance.

## 5.1 Method Lookup Overview

To achieve a classbox-aware semantics for method lookup we must define a new method lookup algorithm which is triggered for each message send. Prior to looking up the method over inheritance and multiple class versions, the *classbox path*, a set of all the involved classboxes in the computation at a given time, has to be determined. Subsequently, this path is provided to the *method lookup algorithm* to select the proper method according to the message sent.

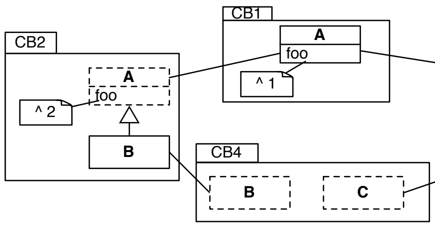


Figure 5.1: Diamond scheme where two classboxes extend the same class

**Classbox path.** The classbox path represents the set of classboxes involved in the current computation and it is obtained from the method call stack. In Figure 5.1 evaluating the expression B new foo in the classbox CB4 generates a path (CB4, CB2), and evaluating C new foo generates (CB4, CB3). This path is computed from the method call stack using reflective features of Squeak.

**Method lookup.** Figure 5.2 describes the lookup algorithm we implemented that ensures the local rebinding property. The proposed method lookup implementation requires three extra arguments (added to the method name and the receiver's class) to search over the graph of classboxes. The selector argument refers to the method name as a symbol; cls refers to the receiver's class; startbox refers to the first classbox where the initial expression is evaluated; currentbox is initialized with startbox when the algorithm is triggered and is used to keep a reference over recursive call of the algorithm; and finally path contains the chain of import for a given method call and its value is computed prior starting the algorithm.

The algorithm first checks whether the class in the current classbox implements the selector we are looking for (lines 5 to 9). If it is found, the lookup is successful and we return the found method (line 9). If it is not found, we recurse. The algorithm gives precedence to imports over inheritance, meaning that first the import chain is traversed (in lines 12 to 18) before considering the inheritance chain (in lines 19 to 30). This last part is the difficult part of the algorithm, since we need to find the classbox where the superclass is defined that is closest to the classbox we started the lookup from. Therefore the algorithm remembers the path while traversing the import chain (line 12), and uses this when determining the classbox for the superclass (line 21).

## 5.2 Import Takes Precedence Over Inheritance

Figure 5.2, lines 11-12 shows that if a class is imported (parentBox is not nil) then the lookup is pursued in the provider classbox (*i.e.* the classbox from where the class is imported from). If this class is not imported (parentBox is nil), as shown at the line 19, then the lookup continues in the superclass.

The lookup in a superclass is done only if it is stated that a class does not provide any implementation for a given message. Within the classbox model this implies that we have to run over the chain of imports to make sure that a classbox does not extend this class with the corresponding method.

Figure 5.3 illustrates this property of the algorithm by depicting an example. It shows four classboxes: GraphicCB, RoundedWindowCB, DoubleBufferCB and DoubleBufferAndRoundedCB. Each of these classes either defines extensions or simply imports classes to combine some of the extensions.

```
lookup: selector class: cls
            startBox: startbox currentBox: currentbox classboxPath: path

    | parentBox theSuper togoBox newPath |
    self
        lookup: selector
        ofClass: cls
        inClassbox: currentbox
        ifPresentDo: [:method | ^ method].
    parentBox := currentbox providerOf: cls name.
    ^ parentBox
        ifNotNil: [path addLast: parentBox.
                   self
                      lookup: selector
                      class: cls
                      startBox: startbox
                      currentBox: parentBox
                      classboxPath: path]
        ifNil: [theSuper := cls superclass.
                theSuper ifNil: [^ cls method: selector notFoundIn: cls].
                togoBox:=path detect: [:box| box scopeContains: theSuper].
                newPath := togoBox = startbox
                               ifTrue: [OrderedCollection with: startbox]
                               ifFalse: [path].
                self
                    lookup: selector
                    class: theSuper
                    startBox: startbox
                    currentBox: togoBox
                    classboxPath: newPath]
```

Figure 5.2: The lookup algorithm that provides the local rebinding

GraphicCB defines a hierarchy composed of three classes: Component provides the methods update and paint, and Window and Frame both override the method paint. Window and Frame are imported in RoundedWindowCB. This first class is extended with a new implementation of paint to make corners of windows smooth by rounding them. DoubleBufferCB extends Component, which is imported from GraphicCB, and simply imports Frame from this same classbox. Component is extended with a redefinition of paint to use double buffering. Finally, DoubleBufferAndRoundedCB combines the two characteristics by importing Component from DoubleBufferAndRoundedCB and by importing Frame from RoundedWindowCB.

In RoundedWindowCB the new implementation of paint does a super paint which executes the paint method in GraphicCB. Evaluating Frame new update in RoundedWindowCB triggers the update method contained in Component and the local definition of paint is executed, the one provided by RoundedWindowCB.

DoubleBufferAndRoundedCB combines the double buffer and the rounded facilities by importing Component from DoubleBufferCB and Frame from RoundedWindowCB. Evaluating Frame new update in DoubleBufferAndRoundedCB triggers update defined in GraphicCB which send the message paint. The implementation taken is the one provided by RoundedWindowCB because Frame is imported from it. This implementation does a super paint, which executes the paint method defined in DoubleBufferCB.

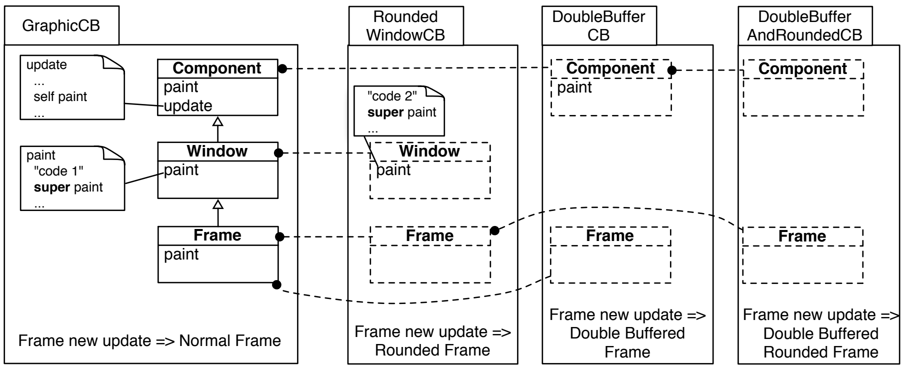

GraphicCB
update ... self paint ...
paint "code 1" super paint ...
Component
paint
update
Window
paint
Frame
paint
Frame new update => Normal Frame
Rounded WindowCB
"code 2" super paint ...
Window
paint
Frame
Frame new update => Rounded Frame
DoubleBuffer CB
Component
paint
Frame
Frame new update => Double Buffered Frame
DoubleBuffer AndRoundedCB
Component
Frame
Frame new update => Double Buffered Rounded Frame

Figure 5.3: Import takes precedence over inheritance

## 5.3 Method Lookup Performance by Modifying the Virtual Machine

As can be expected, introducing the classbox-aware method lookup mechanism introduces some run-time overhead.

Table 5.1 shows the results for some benchmarks that we performed to compare the regular method lookup performance vs. the classbox-aware lookup performance:

| **Benchmark** | **Regular lookup** | **Classbox lookup** | **Overhead** |
|---|---|---|---|
| direct call (method addition) | 5439 | 6824 | 25% |
| looked up call (method addition) | 5453 | 6940 | 27% |
| direct call (method redefinition) | 5438 | 6824 | 25% |
| looked up call (method redefinition) | 5453 | 6941 | 27% |
| opening and closing a web browser | 332 | 548 | 65% |
| opening and closing a mailreader | 536 | 760 | 41% |
| call through 1 classboxes | - | 10234 | - |
| call through 2 classboxes | - | 10357 | - |
| call through 3 classboxes | - | 10554 | - |
| call through 6 classboxes | - | 10654 | - |

Table 5.1: Benchmarks results from Squeak comparing the regular method lookup mechanism with the classbox-aware virtual machine (units are in milliseconds)

- Sending a message defined in the class of the instance (10 million times), and sending a message defined in a super class hierarchy (10 million times). We measured two sets of method calls regarding if a method is added or redefined. This distinction will be useful for comparing this approach to an alternative implementation strategy presented in Section 5.4.
- Measuring launching and closing of two applications implemented in Squeak (a web browser and an e-mail client) within the same classbox (average over 10 times).
- Performing a method call through a chain formed by classboxes extending a class.

The table shows that the time performance penalty for the new lookup scheme by itself is roughly 25 percent, where the real-world applications run about 60 percent slower. We think that this difference is due to the fact that we did not adapt the method cache in the virtual machine. Note that this particular current implementation is straightforward and does not incorporate any optimisations yet. We are however considering changing the structure of the method cache in order to take classboxes into account.

## 5.4 Method Lookup Performance by Manipulating Bytecode

To increase the performance problem from the simple approach shown in Section 5.3, we did a second implementation of classboxes. With this new implementation, there is no need to modify the VM (due to the message passing control mechanism [34] offered by Squeak) and the cost of the new method lookup greatly reduced (thanks to a cache mechanism).

With classboxes several versions of a method can coexist simultaneously. Depending on where this method is called from (*i.e.* from which classbox) the right method implementation is selected according to the method lookup algorithm described previously. When a classbox extends a class it can either be a method addition or a method redefinition. With this implementation, calling a method that has been simply added by a classbox does not impose any overhead. However calling a method that has been redefined has an extra cost: the lookup algorithm is performed. However, this result is cached. Our cache mechanism is based on the following basic assumption: *a redefined method is often called by the same object within the same classbox*. This cache uses a reification of the method call stack (using the pseudo variable thisContext in Smalltalk). The byte-code of an extended method is transformed to include 5 byte-codes that check if the caller for this method is the one that has been previously cached.

This cache mechanism is illustrated on the following example. Lets assume a method foo that simply returns the value 10:

```
foo                                          20  pushConstant: 10
    ˆ10                                      7C  returnTop
```

This method foo is encoded with 2 byte-codes. For method addition there is no need to use a cache because there is only one version of the method present in the system. If this method is redefined, a cache mechanism has to be involved to speed-up the method lookup. After a first call, the bytecode for the method foo is redefined as follows:

```
foo
    (thisContext sender == methodContextSenderCache)
        ifTrue: [ ˆ 10]
        ifFalse: [ dispatcherCache recreateMethodForClassbox.
            ˆ self foo]
```

The first line checks if the the reification of the stack frame of the current call (using thisContext) is the one that is cached (in the value methodContextSenderCache). The byte-code corresponding to this method is:

```
89  pushThisContext:
D4  send: sender
25  pushConstant: methodContextSenderCache
```

```
C6  send: ==
99  jumpFalse: 36
23  pushConstant: 10
7C  returnTop
21  pushConstant: dispatcherCache
D0  send: recreateMethodForClassbox
87  pop
70  self
D2  send: foo
7C  returnTop
```

The original method body is bolded. 5 extra byte-codes are added in front corresponding to the check.

| **Benchmark** | **Regular lookup** | **Classbox lookup** | **Overhead** |
|---|---|---|---|
| direct call (method addition) | 5435 | 5435 | 0% |
| looked up call (method addition) | 5452 | 5453 | 0% |
| direct call (method redefinition) | 5438 | 12553 | 130% |
| looked up call (method redefinition) | 5453 | 12567 | 130% |
| opening and closing a web browser | 332 | 333 | 0% |
| opening and closing a mailreader | 536 | 535 | 0% |
| call through 1 classboxes | - | 12553 | - |
| call through 2 classboxes | - | 12561 | - |
| call through 3 classboxes | - | 12550 | - |
| call through 6 classboxes | - | 12553 | - |

Table 5.2: Benchmark results from Squeak comparing the regular method lookup mechanism with the classbox-aware method lookup (units are in milliseconds)

As in Section 5.3, Table 5.2 illustrates the cost of the method lookup according to three benchmarks:

- Sending 10 million time a message to an object in four different situations: (i) direct invocation of a non redefined method, (ii) invocation of a non redefined method in a superclass, (iii) direct invocation of a redefined method, and (iv) invocation of a redefined method in one superclass.
- Measurement of two applications that does not contain any method redefinition.
- Call through a chain of classboxes.

Invoking a method that has been simply added without being redefined does not have any overhead. This is due to the cache mechanism which is involved in only for method redefinitions. This is illustrated in the third and fourth row of Table 5.2. The overhead is about 130% compared to classical method call. This is due to the cache mechanism. Note that these figures represent the worst case, where the overhead is maximal: the original method is 2 bytecodes long, and when instrumented with the cache it is 13 bytecodes long.

Because the virtual machine is left untouched, using applications that do not redefine methods does not have any cost.

"Call through classboxes" measures the cost of calling a redefined method (*i.e.* instrumented with a cache) through a chain of classboxes. It shows that there is a constant overhead that does not depend on the graph of import. Note that these benchmarks only focus on the worst-case, *i.e.* when a method is extended as the cost of executing normal methods is not changed.

## 5.5 Discussion

We briefly discuss the two implementation strategies.

**Virtual machine modification.** Modifying the virtual machine has the advantage of hiding all the machinery of classboxes from the programmer. Even by using the reflective features of Smalltalk the user cannot access to a description of the method lookup, which is valuable for security reasons. However the maintenance of particularized virtual machine is high and debugging is hampered by the lack of good debugging support. This explains why our runtime performance was not optimal (64% of slowdown for a normal use).

**Controlling send of messages.** By keeping the virtual machine untouched, the development time and the maintenance is highly improved. All the classical development tools are available and the new method lookup algorithm can be debugged using the available debugging facilities of the Smalltalk environment. However, the classbox machinery can be easily exposed using reflective Smalltalk features. As only the redefined methods trigger the new method lookup, the performance of this approach is reduced to only when it is necessary.

## 5.6 Conclusion

This chapter presented two different implementations of classboxes namely, one that modifies the virtual machine to take the new semantics of the method lookup into account, and the second approach that involves performing bytecode manipulation and is based on control of message passing.

In practice, keeping the virtual machine unmodified to benefit from classboxes has the big advantage to not have to update the modified VM whenever some enhancements to the original VM are released. Also, modifying the VM requires significant efforts to reach a stable level.

Chapter 6

# Runtime Adaptation with Dynamic Classboxes

While it is possible to design an application to be adaptable in specific ways, by using approaches such as the Strategy design pattern [42], it is difficult, if not impossible, to anticipate all the ways in which applications may need to be adapted while a system is running.

Whereas Chapter 4 treated unanticipated changes in a static setting by means of class extensions, this chapter focuses on a second kind of unanticipated changes, dynamic system adaptation.

## 6.1 Introduction

Dynamically adapting a running application means changing its behavior without stopping and restarting it [76, 79, 77, 72, 59, 32, 28]. Mobile devices and embedded systems often require dynamic adaptation [81]. Static AOP aims at modifying the control flow to an application [48]. However this has to be specified statically, at compile time. Recently, there has been a growing interest in using dynamic aspect-oriented techniques [77], class replacement [59, 32] and class extension [28] to support dynamic adaptation. However, existing implementations either limit the kinds of changes that can be applied, or require a modified virtual machine.

Without a scoping mechanism, composing several aspects that extend a class is delicate (especially when these aspects overlap each other) and requires a dedicated language (Hyper/J [73]). Assimilating an aspect as a scope that bounds the visibility of its extension makes composition easier: aspects that conflict with each other can be used at the same time because their definitions are visible in different scopes.

**Summary.** Our approach to dynamically applying changes is based on classboxes. This chapter describes an extension to classboxes (i) to support addition of instance variables and (ii) to make them fully dynamic: they can be dynamically and atomically loaded and unloaded.

Within a classbox, classes are defined or imported from other classboxes. Imported classes can then be extended. These extensions consist of adding new instance variables, and adding and redefining methods. However, such extensions have bounded visibility: variables and methods defined on a class are visible *only* visible from the perspective of this classbox.

**Contributions.** The contributions of this chapter are:

- Import relationships between classboxes can be dynamically modified and added on the fly.
- Classboxes can dynamically be replaced by a new classbox or a set of classboxes.

**Structure of the chapter.** In this chapter we motivate dynamic adaptation (Section 6.2). Then the classbox model is described by showing how it solves the adaptation problem (Section 6.3). Finally, some implementation issues are described (Section 6.4).

## 6.2 Dynamic Adaptation Needs

We motivate the need for dynamic adaptation and show how traditional approaches do not offer satisfactory solutions.

### 6.2.1 A Motivating Example

While the primary function of a cellphone is to make phone calls, nowadays all cellphones provide numerous advanced display facilities (*i.e.* colors, aliasing, 3D, animations ...). Such advanced features are considered to be *non-essential* and in case of particular situations such as device overheating or battery power shortage can be disabled to preserve the primary cellphone behavior [81]. During a phone call, disabling non-essential features to spare the battery must not interrupt the communication. Under these conditions systems such as a cellphone cannot tolerate being halted to be recompiled: they have to be adapted on the fly.

Figure 6.1 depicts the parts of a mobile cell phone related to its display capabilities. Many current cellphones offer two LCD screens: an internal one only usable when the phone is open, and an external one displaying information about incoming calls. The external screen usually has fewer capabilities than the internal one. The cellphone's operating system provides an abstraction of the hardware to the application environment. It contains several modules related to fault management, power management and resource management.

The application environment contains (i) the user interface defined by the navigation application (Navigation), (ii) a controller for the internal screen and (iii) another controller for the external screen, (iv) graphical user interface widgets like scroll bars, menus, progress bars (2DWidget) using (v) a lower-level set of graphical elements used by the widgets such as texts, lines or rectangles (GraphicalElement).

### 6.2.2 Dynamic Adaptation

A typical scenario of dynamic adaptation occurs when the fault manager receives a low battery event triggered by the hardware. The power consumption has to be reduced, so the power manager asks the resource manager to reduce its needs by downgrading some non-essential facilities like the display [81]. When the battery is full and the power consumption is not restricted, the display uses advanced

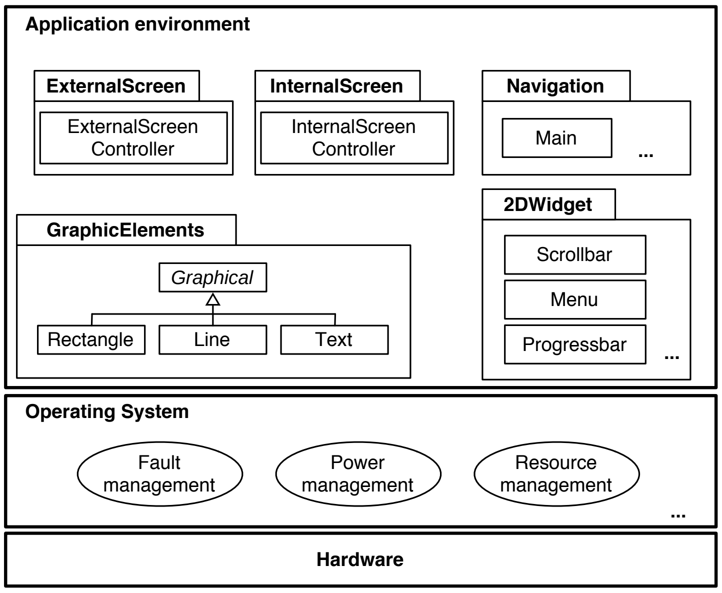


Figure 6.1: The application environment contains 5 parts: Navigation defining the main application, two controllers related to the internal and external screen (InternalScreenController and ExternalScreenController), 2DWidget offering some graphical user interfaces widgets and GraphicalElements containing low-level graphical facilities

animations and 3D rendering to display widgets to give a more attractive display to the end-user. Reducing the display facility in case of power shortage (less than 20% of the battery left) consists in keeping the internal screen fully active but removing the animation and using a colored two dimensional rendering on the external screen. When the lower threshold of 8% approaches, 3D is removed from the internal screen and the external screen becomes colorless.

**Applying changes to a hierarchy.** Switching from 3D rendering to 2D rendering is done by changing the set of methods defined on the widgets. However, if the graphical elements have to be colorless, the color concern applied to the hierarchy has to be removed from the existing hierarchy. This modification consists in removing an instance variable and a method setColor: color from a class root of the hierarchy, and providing a draw method for each of the graphical elements (subclasses of GraphicalElement). These new draw methods do not use the color variable.

Figure 6.2 presents a typical problem of extension. It describes the changes applied to a hierarchy consisting in adding or removing a color concern. Adding the concern means adding an instance variable color and a method setColor: color to the abstract class GraphicalElement. Also, each of the subclasses are extended with draw and aliasing methods that use the color variable. Removing this concern means removing the aliasing method, replacing the colored draw method by the colorless one (that does not use the color variable), and removing this variable.

**Key problems.** Dynamic adaptation of an application consists in changing the definition of the application to have a proper configuration according to its context and environment at a given moment in time. As illustrated by the above example the following points have to be addressed:

- **Dynamicity.** These changes have to be applied dynamically: a cellphone cannot be interrupted, stopped and then restarted. For instance changing the display configuration during a phone call

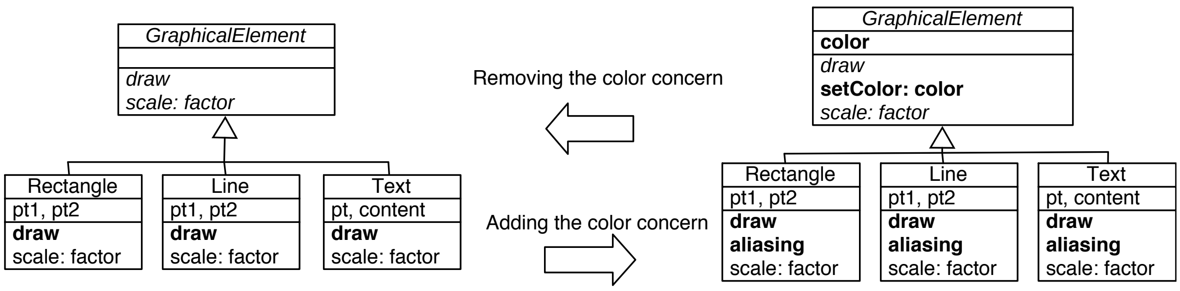


Figure 6.2: Adding and removing a color concern throughout a hierarchy of classes (bold elements indicate variation)

should not be perceived by the user. Changes have to be loaded when they are necessary and unloaded at runtime to spare memory and battery consumption [81].

- **Small runtime.** The runtime system has to be kept small and simple. This means that a compiler should not be part of the runtime. Changes should be done without requiring any source code.
- **Changes crosscut several classes.** Adapting a part of a system requires some dynamic changes, such as adding or removing state (instance variable) or fragment of behavior (set of methods), that have to be applied to a set of classes.

### 6.2.3 Scoped Changes

Due to space limitations, redundancies in the system have to be avoided. Even if one screen is colored and the other not, there should be only one hierarchy of graphical elements used by the widgets. This is achieved by scoping the changes applied to the application. Only some particular changes, like a color concern, applied to a set of classes are accessible within one part of the system (*e.g.* internal screen) whereas these extensions might not be visible within another part (*e.g.* external screen).

**Key problems.** Changes required by different parts of a system have to be scoped separately to avoid conflicts when applying different changes at the same time and to limit the visibility of the changes to the part of the system that actually needs them.

### 6.2.4 Limitation of Traditional Approaches

Extending a class by creating a subclass imposes some constraints on clients of the extended classes [39, 16]: references contained in a client have to be updated. Extending a hierarchy using subclassing also leads to code duplication and static type issues. Figure 6.3 shows how the base system containing the graphical hierarchy could be extended based on subclassing with a color concern. It is assumed that we only consider single inheritance here. Each graphical element adds the instance variable color, redefines the method draw (to take the color into account), and adds the methods setColor: color and scale: factor.

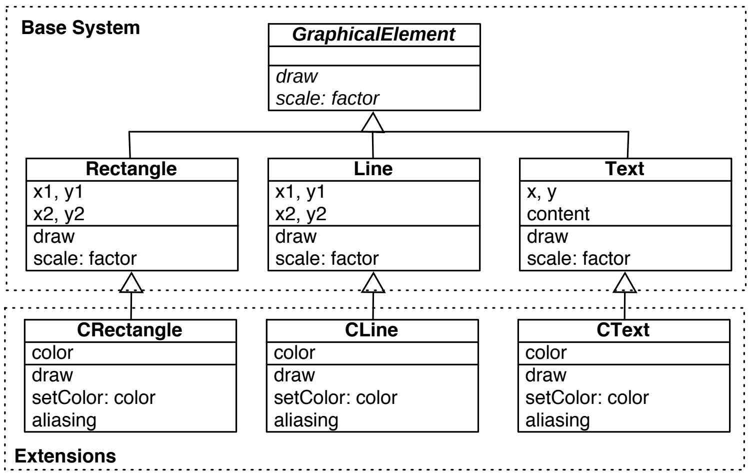


Figure 6.3: Extensions by subclassing

This leads to several drawbacks:

- **Duplication of code.** The color variable and the setColor: color method are duplicated as many times as there are graphical elements. The consequence is that maintenance becomes harder and more error-prone. For instance a future evolution that makes the color concern evolve would also need to be duplicated.
- **Clients need to be adapted.** The reuse of existing classes (originally intended to be used with the first version of the library) implies some adaptations because these classes refer to original classes (Rectangle, Line, ...) and not the subclasses representing the extensions (CRectangle, CLine, ...).
- **Statically typed languages.** In Figure 6.3 each of the nodes is subclassed, leading to a great deal of type casts in order to be type safe. For instance the aliasing method can be used only through the type of the extension (CRectangle, CLine, and CText) and not from the type defined by GraphicalElement. By using single inheritance, subclassing the class GraphicalElement define a new hierarchy with ExtendedGraphicalElement at the top, duplicate the classes Rectangle, Line, and Text.

## 6.3 Supporting Dynamic Adaptations With Classboxes

### 6.3.1 Dynamic Application of Classboxes

We extend classboxes to enable them to adapt to an application while it is running. This extension consists in making a classbox installable and removable at runtime. There is no need to stop and restart a running system. No source code is required. A classbox can be dynamically swapped by a new classbox or a set of new classboxes: the new classbox(es) must provide at least the same set of classes.

When the amount of energy left in the battery enters a different range, connections between the classboxes are modified and classboxes that are not used anymore are unloaded. For instance with less

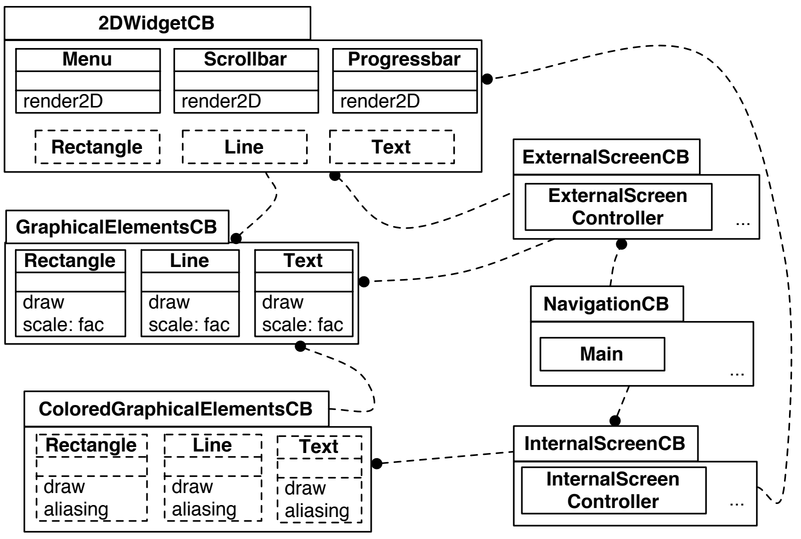


Figure 6.4: Rearranged classboxes: the external screen is colorless and the internal one colored

than 8% of the battery the internal screen uses colored 2D widgets and the external one colorless 2D widgets: the 3D facility becomes unnecessary. Classboxes are then rearranged (Figure 6.4) to make the internal screen use colored 2D widgets and the external one use colorless 2D widgets: ExternalScreenCB imports colorless graphical elements from GraphicalElementsCB, InternalScreenCB now imports the widgets from 2DWidgetCB. The classbox 3DWidgetCB becomes unnecessary and is unloaded from the memory to save some energy.

## 6.4 Implementation

Our current implementation is made in Squeak [46], an open-source Smalltalk implementation, and uses the bytecode manipulation technics (*i.e.* the second strategy described in the previous chapter). In this section we discuss some central implementation points and give some performance results.

**On-the-fly Method Switching.** If a method is replaced while it is activated, then previous calls continue with the former definition while any new call triggers the new definition. This is an approach similar to Java [32] where all invocations of old methods are allowed to complete, whereas all method calls initiated after class redefinition go to the new method. This is also true if a method calls itself after being changed. The call invokes the new method definition, and then continues with to the old definition.

**Updating Instances.** When adding new instance variables to a class the natural question regards existing instances of the modified class [80, 84] . These need to be adapted by having their size updated to the number of new variables added. New variables are initialized with an empty value (nil). Extending the state of a class is done atomically no thread or external signal can interrupt this operation. This is necessary to preserve the consistency of the running system.

**Adapting Bytecodes.** With the usual set of bytecodes used with virtual machines, accessing an instance variable of an object is done using an offset. For instance the method scale: factor in the class Rectangle accesses the pt1 instance variable by referring to it through the first field of the object.

Adding a color instance variable in GraphicalElement triggers an adaptation process of the scale: factor method because the first field now corresponds to the color instance variable. Offset references remain consistent to the variable they refer to by computing a new offset for instance variables defined in subclasses.

The variable color has to be placed at the first offset in the class GraphicalElement because it reduces overhead when performing the necessary linearization when creating instances.

This is done by manipulating the bytecodes of a method and adapting the bytecodes that represent accesses (read and write) to instance variables.

**Performance.** With our current implementation, the cost of installing and uninstalling a classbox that extends 100 classes with 500 methods is about 16 seconds. Note that this result is just a preliminary result and is obtained with our development. We are confident that the initial performance figures for installing and uninstalling classboxes can be significantly improved by more careful coding of the dynamic engine, by using an optimized compiler and by removing all the development tools that are unnecessary for an end-user application.

Having added dynamicity to classboxes did not bring any runtime cost when invoking an added method. Calling a normal method (result of a method addition) does not cost any overhead. However, there is an overhead of about 50 % when calling a redefined method. This overhead is due to the local rebinding facility: when sending a message, a computation based on the method call stack is performed in order to take the expected method implementation when invoked. Instantiating a class has no overhead.

This has to be put in contrast with other dynamic adaptation of behavior mechanism. IguanaJ [79] is 24 times slower then a normal JVM to call a method and return from it. Object creation is 25 times slower than a classical Java VM. Most of the other runtime reflective architecture like PROSE [77] or Guaraná [72] indicate delays of the same order.

## 6.5 Conclusion

Few solutions exist to support dynamic adaptation and dynamic application of changes. Classboxes allow a module to add or refine methods, and to add state to classes defined in other modules. In this chapter we extended classboxes to be dynamically replaced by a new classbox or by a set of classboxes.

A classbox offers a uniform and powerful mechanism to support a simple and structural kind of aspect. Classboxes do not offer the possibility to define joint-points like before or after. However, classboxes unify the notion of modules with that of aspects. In addition classboxes dynamicity supports dynamic adaptation by allowing changes to be installed and removed dynamically. As such, classboxes represent a important step in our quest for the *minimal* mechanisms that support aspect-oriented programming.

# Chapter 7

# Crosscutting Extensions with Traits and Classboxes

Classboxes allow classes to be extended with methods addition and redefinitions. This kind of extension is nominative, in the sense that an extension is applied to a single class. By combining classboxes with traits [35], this chapter proposes an enhancement of classboxes to make a set of methods a crosscutting extension *i.e.* applicable to different classes. This correspond to the third kind of unanticipated changes presented in Chapter 3.

*Traits* define groups of methods that can be arbitrary used by classes if some requirements are fulfilled [35]. This chapter proposes a symbiosis of classboxes with traits. As a result, class extensions defined in a classbox as a trait can be applied to any class.

## 7.1 Introduction

By importing and extending the root of a class hierarchy, an extension is propagated to all subclasses because of class inheritance. The same extension cannot be applied to multiple classes without forming a class hierarchy. The reason for this is that a class extension is nominative, *i.e.* it is applied to a designated class.

Making a class extension applicable to various classes requires to define a set of methods (*i.e.* a class extension) separately from the class to which these methods should extend. A group of methods can therefore be later on applied to a class, resulting in a class extension.

Within the spirit of mixins, traits [35] 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. With traits, classes are composed from a set of traits by specifying glue code that connects the traits together and accesses the necessary state.

**Summary.** This chapter describes the combination of traits and classboxes resulting in a system where classes can be extended with crosscutting changes 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.

**Contributions.** The contributions of this chapter are:

- It describes an approach supporting unanticipated crosscutting changes based on a symbiosis between traits and classboxes.
- The notion of class extension offered by classboxes has been enhanced by allowing a class to locally use a trait.
- Extensions modeled as traits are applicable multiple times (*i.e.* to different classes), as in the original version of classboxes.
- This combination between traits and classboxes can be applied to express a collaborative architecture where a classbox defines a collaboration and a trait defines a role.

**Structure of the chapter.** In this chapter, Section 7.2 describes traits and illustrates them with an example based on modeling geometrical objects. Section 7.3 presents the symbiosis of these two models. Then Section 7.4 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 7.5, and Section 7.6 summarizes the chapter.

## 7.2 Traits

Traits are essentially sets of methods that serve as the behavioral building blocks of classes and the primitive units of code reuse [35]. 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.

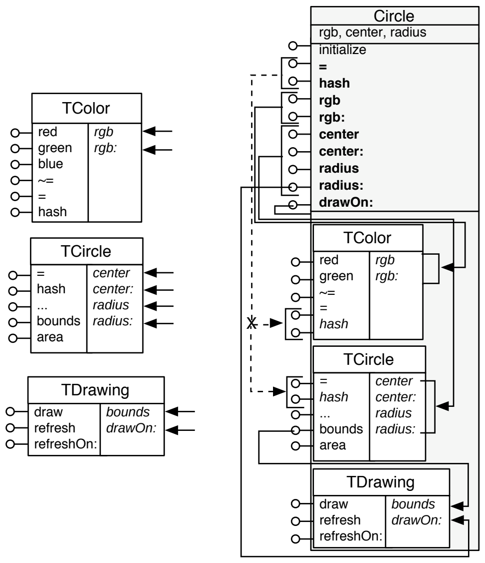


Figure 7.1: The class Circle is composed of three traits TColor, TCircle and TDrawing. A trait consists of a name, a list of defined methods (left-hand pane) and a list of required methods (right-hand pane). A conflict is resolved by redefining the methods hash and =

A *conflict* arises if we combine two or more traits that provide identically named methods 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 [35].

**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 7.1 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.

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

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

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, the aliasing (for instance, m1 → m2 defines m1 as a new name for the m2 method).

Conflict is explicitly resolved by (re)defining methods hash and = at the level of the class. Since methods at the class level take precedence over methods provided by the traits, the conflict is resolved. Note that these methods are implemented by using the new alias obtained from the traits composition (colorHash, circleHash, . . . ).

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

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

**Use of traits has to be foreseen.** 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.

## 7.3 A Traits and Classboxes Symbiosis

In this section, we present 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 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 7.3.1) and extending a class by making it use a trait (Section 7.3.2). Finally, Section 7.3.3 is dedicated to the scope of class extension visibility.

### 7.3.1 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 class can be bound to any name. Several traits with the same name cannot be simultaneously visible in a given classbox, but can live in different classboxes, just like classes.

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 applying traits, which are themselves defined or imported.

### 7.3.2 Class Extension

We refine the notion of class extension by adding the possibility of applying a trait. Figure 7.2 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.

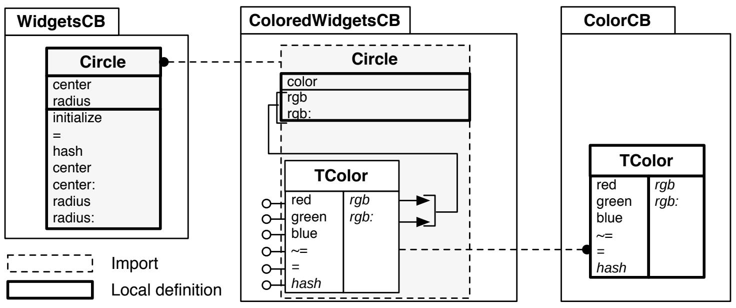


Figure 7.2: The class Circle is imported in ColoredWidgetsCB and extended by using the imported trait TColor

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
```

Traits have 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.

### 7.3.3 Visibility of Extensions

Classboxes allow the visibility of definitions to be bound to a particular scope. Contrary to the global visibility of extensions in AspectJ [49] (with inter-type) and Multijava [28, 66] (with open-class), 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 applied the trait.

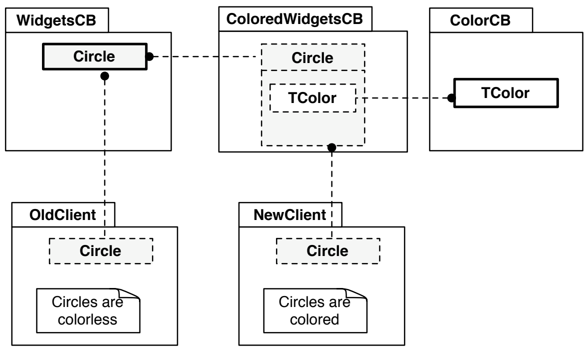


Figure 7.3: Two versions of Circle coexist at the same time. From the point of view of OldClient circles are colorless, and from the point of view of NewClient they are colored

Figure 7.3 shows an example of two clients relying on two different versions of the class Circle. Within the classbox OldClient, circles are colorless. Within NewClient however, circles are colored because in 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 [44] without the limitations of current implementations.

## 7.4 Cross-cutting Collaborations and Unanticipated Modifications

*Collaborations* have been introduced to describe functionalities that cross-cut several classes [44, 86]. 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. In this section we show how a collaboration can be expressed with a combination of traits and classboxes by regarding a role as a trait and a collaboration as a classbox.

### 7.4.1 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 [44], and then in several others like in VanHilst and Notkin [94] and Smaragdakis [86]). This graph traversal application is the canonical example to illustrate a collaboration-based achitecture.

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.

### 7.4.2 Expressing Collaborations with Traits and Classboxes

A collaboration is represented by a classbox, and a role by a trait. Figure 7.4 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 7.4 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

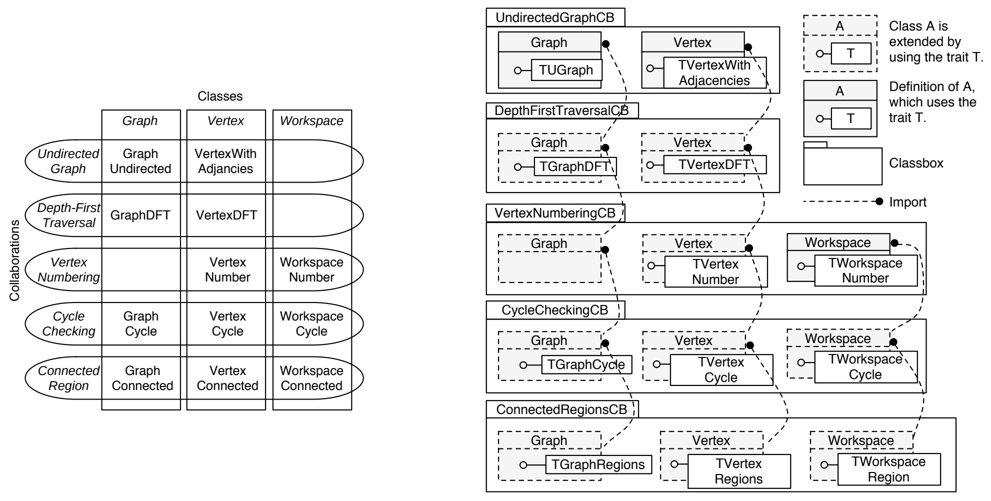


Figure 7.4: Decomposing into collaborations. Left-hand side: ovals represent collaborations, rectangles refer to classes, and at their intersection the squares refer to the roles. Right-hand side: classboxes represent the collaborations and traits represent the roles

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.

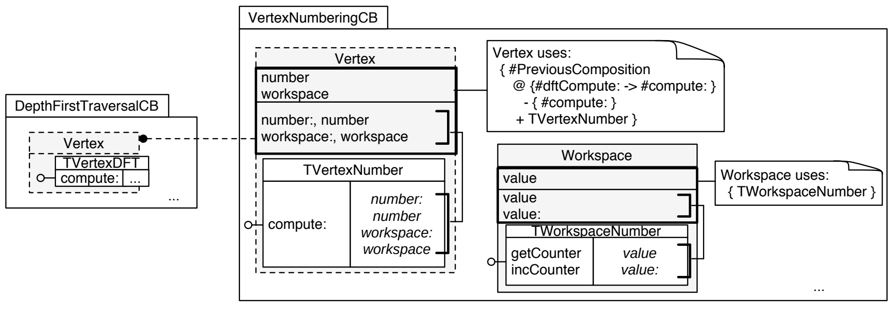


Figure 7.5: Definition of the collaboration *VertexNumbering*

Figure 7.5 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 some accessors to these variables are 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 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)
```

### 7.4.3 Advantages of Coupling the Traits and Classboxes

**Identity of participant preserved.** A *mixin* application creates a subclass [20]. Using subclassing, an unanticipated change cannot be applied without modifying former clients [39] 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 that is well-defined in advance. However, they do not help in refining an application with changes that were not initially planned. With our approach, a new collaboration layer can be added to an existing application by preserving former clients from being impacted. As a result, a collaboration defines an 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 [8] and Steyaert [88]. 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.

## 7.5 Discussion and Evaluation

**Link class-trait separated from the class definition.** In the original traits, the *use* relationship between a class and the traits it uses is specified within the definition of the class [35] 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 any programmer needing to extend the class. Hence it is possible to dynamically change the classes and the traits they use.

**Supporting incremental changes.** A class can be incrementally extended by a chain of classboxes. This class is then 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 7.5.

**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 approach, 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 7.4, 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 7.5). This method has to be provided by the class before applying TVertexNumber. Declaring required methods forces the collaborative architecture to be coherent. However, our approach does not offer any guarantee that traits that are 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.

## 7.6 Conclusion

We describe an approach supporting crosscutting changes based on a symbiosis between traits and classboxes. 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 chapter shows how this combination between traits and classboxes can be applied to express a collaborative architecture where a classbox defines a collaboration. Each collaboration contains a set of roles that are modeled with traits.

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

# Chapter 8

# Classboxes in a Statically Typed Environment

Chapter 5 describes an implementation of classboxes in the context of Smalltalk, a dynamically typed programming language. This chapter presents classboxes for one of the mainstream languages, Java.

## 8.1 Introduction

**Summary.** In this chapter we demonstrate how classboxes can be implemented in statically-typed languages like Java while minimizing the extension of the hosting language. We also describe the implementation of Classbox/J as a proof-of-concept. It is freely downloadable for MacOSX on www-.iam.unibe.ch/∼scg/Downloads/CBJ/CBJ.zip

**Contributions.** The contributions of this chapter are:

- A proof-of-concept implementation of classboxes for statically typed languages. Classbox/J consists of a minimal extension of Java: (i) package import clauses are made transitive, and (ii) packages are able to refine imported classes and export these classes to other packages.

- The original classbox model is extended with a mechanism enabling refinements to access prior definitions. The Swing refactoring towards classboxes motivates the need to invoke original methods from their redefined bodies.

**Structure of the chapter.** In Section 8.2 we present the model of classboxes for Java. In Section 8.3 we present an example illustrating how classboxes support the implementation of cross-cutting changes. In Section 8.4 we describe our Java implementation of classboxes. In Section 8.5 we conclude by summarizing the work presented in this chapter.

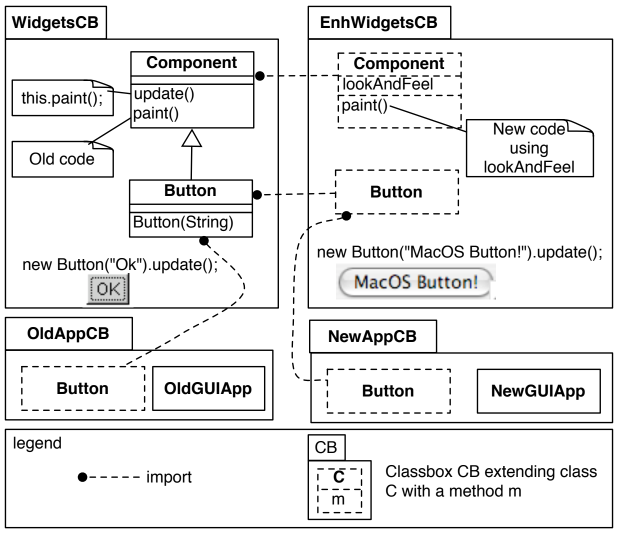


Figure 8.1: Two versions of classes Component and Button are used by two differents clients OldAppCB and NewAppCB

## 8.2 Classbox/J

In Java, a package can define new classes and it may refer to classes defined in other packages using an *import* clause. After importing a class, a package can either subclass it or reference it in a declaration. In pure Java, import statements are not transitive: a package p2 cannot import a class C from a package p1 if C was imported rather than defined in p1. In contrast to MultiJava [66], Hyper/J [73], CLOS [31] and Smalltalk [43], a Java package cannot add methods to a class defined in another package. Therefore a package can be adapted only by subclassing its member classes.

Classbox/J addresses these shortcomings by offering a means to refine classes within a well-defined scope.

### 8.2.1 Classbox/J in a Nutshell

Classbox/J is a module system for Java allowing classes to be refined with new class members, such as fields, methods and inner classes. A classbox in Classbox/J is essentially a Java package with the following three important differences: (i) imported classes can be refined by adding or redefining class members using the refine keyword, (ii) a class defined or imported within a classbox p can be imported by another classbox. This allows the import clause to be transitive, and (iii) a refined method can access its original behavior using the original keyword.

We illustrate Classbox/J with a small example based on the Swing case study.

**Refining classes.** Figure 8.1 illustrates two classboxes WidgetsCB and EnhWidgetsCB. WidgetsCB defines two classes Component and Button. EnhWidgetsCB imports them, refining Component with a new instance variable lookAndFeel and redefining the paint() method. These classboxes are implemented as follows:

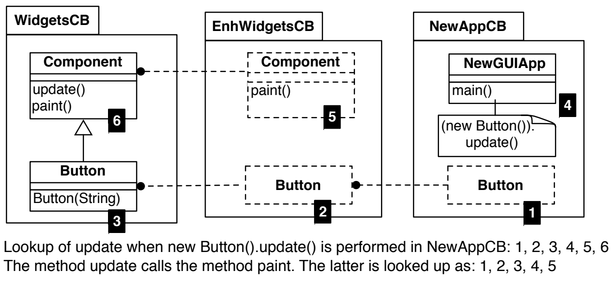

WidgetsCB
Component
update()
paint()
Button
Button(String)
EnhWidgetsCB
Component
paint()
Button
NewAppCB
NewGUIApp
main()
(new Button()).
update()
Button
1
2
3
4
5
6
Lookup of update when new Button().update() is performed in NewAppCB: 1, 2, 3, 4, 5, 6
The method update calls the method paint. The latter is looked up as: 1, 2, 3, 4, 5

Figure 8.2: Locality of changes entails a new method lookup semantics. The numbers within the black boxes indicate the steps taken in looking up a message sent to a button

```
package WidgetsCB;
public class Component {
    public void update () { this.paint(); }
    public void paint () { /* Old Code */ }
}
public class Button extends Component {
    public Button (String name) { ... }
}

package EnhWidgetsCB;
import WidgetsCB.Component;
import WidgetsCB.Button;
refine Component {
    private ComponentUI lookAndFeel;
    public void paint () { /* New code using lookAndFeel */ }
}
```

Refining a class conceptually defines a new version of it. In the previous example, two versions of Component *coexist* at the same time within the system in different scopes. The original version is accessible through WidgetsCB and the new version through EnhWidgetsCB. Class members refining an imported class are local to the refining classbox and to other classboxes that may import the refined class.

**Transitive import.** A class imported by a classbox can be transitively imported by other classboxes, whether this class is refined or not. For instance, a client of the new version of the widgets can be defined as:

```
package NewAppCB;
import EnhWidgetsCB.Button;
public class App {
    public static void main(String[] argv) {
    ... new Button().paint(); ...
    }
}
```

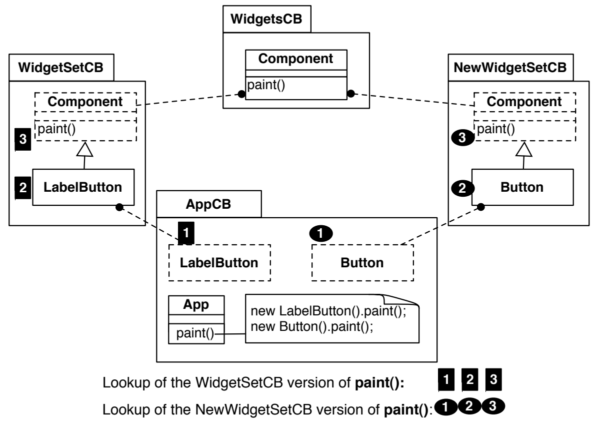


Figure 8.3: Within one classbox, different versions of the same class can be accessible. From AppCB sending the paint() message sent to a LabelButton triggers WidgetSetCB's refinement, whereas sending it to a Button triggers NewWidgetSetCB

### 8.2.2 New Method Lookup Semantics

As required by classboxes, class refinements have bounded visibility. Moreover, redefinitions have precedence over imported definitions. This behavior is obtained by a new semantics for method lookup. We illustrate this operationally.

**Import over inheritance.** Import statements between packages have to be taken into account when looking up a message. The main point is that the import clause has precedence over inheritance: before looking a method up in the superclass, the chain of imports has to be considered first.

Figure 8.2 illustrates the lookup of messages update() and paint(). When the message update() is sent to an instance of Button in the classbox NewAppCB, the lookup algorithm first searches for the implementation of update() in the classbox NewAppCB (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 (NewAppCB, in step 4). Because Component is not visible within NewAppCB 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 as in steps 1 through 5.

Note that defining new semantics for the method lookup algorithm does not necessarily mean that the virtual machine (VM) must be modified. As described in Section 8.4, the desired behavior can be obtained by inserting some code that performs dynamic run-time stack introspection where a method redefinition occurs.

**Multiple imports.** As illustrated in Figure 8.3, a diamond graph of imports may imply the use of different class refinements defined by several classboxes. In the classbox AppCB, sending the paint() message to an instance of LabelButton invokes the implementation of paint() on Component defined by WidgetSetCB. In a similar way, sending this message to an instance of Button triggers the imple-

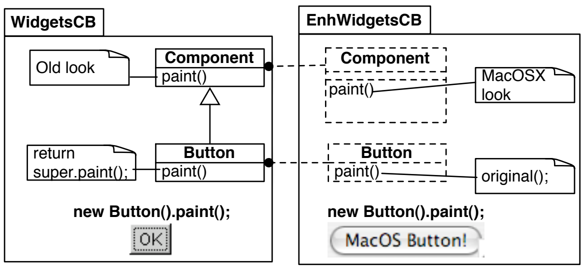


Figure 8.4: The original() construct invokes the hidden method but in the context of the classbox changes

mentation brought by NewWidgetSetCB on Component.

**Accessing the original method.** When a method is redefined, the original method is accessible using the construct original().

For instance, in the classbox EnhWidgetsCB the extension of Component could be:

```
refine Component {
    private ComponentUI lookAndFeel;
    public void paint () {
        if (lookAndFeel == nil) { original();}
        else { /* use lookAndFeel */ }
    }
}
```

The original() construct invokes the first method (*e.g.* WidgetsCB.Button.paint() in Figure 8.4) in the import chain that was redefined by the method containing the expression original() (*e.g.* EnhWidgetsCB.Button.paint()). Note that in particular super invocation in the original methods takes into account potential changes introduced by the classbox containing the original() invocation, preserving that way the method lookup semantics of classbox. In Figure 8.4, new Button().paint() displays a button having a MacOSX look since, first the method WidgetsCB.Button.paint() is executed and the super invocation invokes EnhWidgetsCB.Component.paint().

It is precisely this kind of scenario, which arises frequently in the Swing case study, that has motivated the addition of the original mechanism into the classbox model.

### 8.2.3 Properties of the Model

The model of classboxes defined in Chapter 4 exhibits several properties related to the visibility of refinements.

**Locality of Changes.** MultiJava [66] with its open-classes and Aspect/J [6] with its inter-types allow class members to be defined separately from the class they are related to. Class members are not, however, contained in a unit of scope, therefore redefinition is not allowed and composition has to be explicitly stated. With classboxes, refinements of an imported class are visible to the refining classbox and to other classboxes that import this refined class. The refined class is a new version of the original class that co-exists in the same system. Figure 8.1 shows two clients OldAppCB and NewAppCB using the old and new version of the widget framework. Any refinement introduced to WidgetsCB by EnhWidgetsCB does not impact OldAppCB. This is because changes are confined to EnhWidgetsCB and to other classboxes that may imported the classes it refines (*e.g.* NewAppCB).

**Precedence of redefinition.** Redefined class members have precedence over the imported definition. EnhWidgetsCB redefines the method paint() for Component, thus hiding the previous definition. From this classbox and other classboxes that may import Component or its subclasses, the original definition of paint() is no longer accessible. Within the classbox EnhWidgetsCB or NewAppCB, sending the message update() or paint() to an instance of Button will trigger the new definition of paint().

**Refinements along a chain of import.** With classboxes, imports are transitive: a new version of an imported class can be re-imported. Figure 8.1 shows the class Button defined in WidgetsCB that is imported in EnhWidgetsCB and from this last, are imported in NewAppCB. From the point of view of an importing classbox, there is no distinction between a class that is defined or imported in the provider classbox (*i.e.* classbox where the class is imported from). An imported class can always be refined and then re-imported, even multiple times over a chain of imports.

## 8.3 Cross-cutting Changes

Refining a class is superficially similar to subclassing: a classbox can add new interfaces, fields, methods, static fields, inner classes and constructors as well as redefine methods of an imported class. The key difference is that the changes are applied to the original class, not a subclass, but only within a well-defined scope. It is this feature that supports the introduction of cross-cutting changes. The following example shows how a *look and feel* feature is added to the root of a class hierarchy without breaking former clients, while propagating the refinements to collaborating classes. As shown in Figure 8.5, two classboxes WidgetsCB and FactoryCB define a base system which clients rely on. Since modifying these base classes would break these clients, changes cannot be directly applied to the classboxes WidgetsCB and FactoryCB, but are introduced in classbox LookAndFeelCB and used by a new client in AppCB. The rest of this section shows how classboxes allow one to incorporate these changes without having to modify WidgetsCB and FactoryCB.

The following example shows some refinements defined with classboxes on a base system that (1) does not break clients that rely on the original definitions of this system, and that (2) propagate these refinements to collaborating classes defined in other classboxes.

**Base system.** The classbox WidgetsCB defines three classes: an abstract class Component and two subclasses Button and Window. The source code of this classbox is:

```
package WidgetsCB;
public abstract class Component {
    public abstract void paint();
}
public class Button extends Component {
    public Button () { }
    public void paint() {
        System.out.println("Button");
    }
}
public class Window extends Component {
```

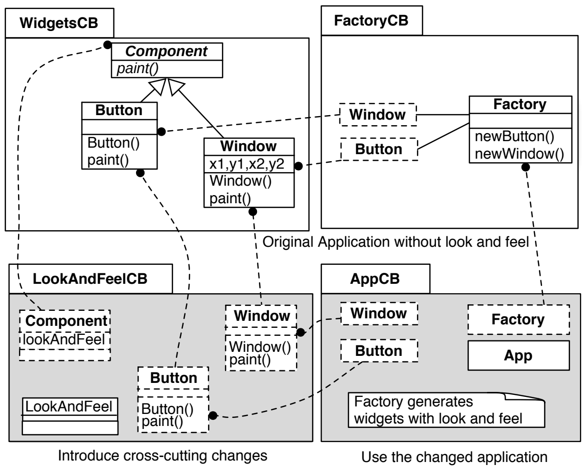


Figure 8.5: From the viewpoint of AppCB, refinements of the root of the hierarchy (Component) are propagated to the class Factory. This is a consequence of importing the version of widgets that have a look and feel

```
        int x1, y1, x2, y2;
        public Window () { x1 = 50; y1 = 50; x2= 200; y2=200;}
        public void paint() {
            System.out.println("Window");
        }
    }
```

New widgets are created using a factory. This factory is implemented in a separate classbox FactoryCB. When it was designed, the implementor of Factory relied on the version of the widgets obtained from WidgetsCB without any look and feel. The widget factory is defined as:

```
    package FactoryCB;
    import WidgetsCB.*;
    public class Factory {
        public Button newButton () { return new Button(); }
        public Window newWindow () { return new Window(); }
    }
```

**Refinement of the base system.** To introduce the changes that add a "look and feel" to the widgets, two new classboxes are added: LookAndFeelCB, which effectively defines the changes, and AppCB, which is a new client of the resulting system. In LookAndFeelCB the root class Component is refined with a lookAndFeel variable. In order for classes Button and Window to use this new variable added to their superclass, their constructor and paint() are redefined. These refinements are defined as:

```
    package LookAndFeelCB;
    import WidgetsCB.Component;
    import WidgetsCB.Button;
    import WidgetsCB.Window;
    public class LookAndFeel {
        ...
```

```
}
refine Component {
    LookAndFeel lookAndFeel; // Variable added to Component
}
refine Button {
    public Button() { // Constructor redefined
        lookAndFeel = new LookAndFeel(”ButtonMacOSX”);
        original(); // Original constructor called
    }
    public void paint() { // Method paint redefined
        System.out.println(lookAndFeel.getName());
    }
}
refine Window {
    public Window() { // Constructor redefined
        lookAndFeel = new LookAndFeel(”WindowMacOSX”);
        original(); // Original constructor called
    }
    public void paint() { // Method paint redefined
        System.out.println(lookAndFeel.getName());
    }
}
```

A small application is built in the classbox AppCB. This classbox imports the class Factory from FactoryCB and the widgets having a look and feel from LookAndFeelCB. Now when the new application uses the factory to create widgets, it gets widgets with the look and feel as defined in the LookAndFeelCB classbox, whereas the clients of the original code defined in WidgetsCB are not impacted, *i.e.* get widgets without look and feel. As AppCB imports the version of Window and Button with a look and feel, from the perspective of AppCB, this version of the widgets takes precedence over the one present in FactoryCB.

```
package AppCB;
import FactoryCB.*;
import LookAndFeelCB.*;
public class App {
    public static void main (String[] argv) {
        Factory f = new Factory();
        Window w = f.newWindow();
        Button b = f.newButton();
        //Display ”WindowMacOSX” and ”ButtonMacOSX”
        w.paint();
        b.paint();
    }
}
```

## 8.4 Implementation

We implemented a preprocessor that translates classbox definitions into pure Java files, which are then compiled using a classical compiler. While producing Java source files, classboxes are compiled away by producing a Java package for each classbox. Our implementation is freely available at www.iam-

.unibe.ch/∼scg/Research/Classboxes. It offers an executable `cbj` compiler similar to the `javac` compiler, where argument files are classbox-aware. Please note that this implementation is *naive* and serves only as a proof-of-concept for Java.

Our implementation handles three different ways of refining an imported class: (i) a new class member is added (*i.e.* not redefined), (ii) a class member other than a method is redefined, and (iii) a method is redefined. The following sections examine each of these cases. We drew a distinction between redefined methods and other redefined class members because the former are dynamically looked up when messages are sent, but not the latter (which are statically bound). We then describe how the new method lookup semantics is implemented using dynamic introspection of the method call stack (Section 8.4.4). And finally we show how the transitivity of imports is handled (Section 8.4.5) and we present some limitations and possible improvements (Section 8.4.6).

### 8.4.1 Pure Class Member Addition

Class members that are new additions (not redefinitions) are inserted into the Java class without being modified. For instance, a classbox WidgetsCB defines an empty class Component, that is refined in a classbox EnhWidgetsCB.

```
//Classbox WidgetsCB
package WidgetsCB;
public class Component {
}
```

```
//Classbox EnhWidgetsCB
package EnhWidgetsCB;
import WidgetsCB.Component;
refine Component {
    private int color;
    public int color () {
        return color;
    }
}
```

When passed to our `cbj` preprocessor, the resulting Java package used to generate pure java bytecodes is:

```
package WidgetsCB;
public class Component {
    private int color;
    public int color () {
        return color;
    }
}
```

### 8.4.2 Redefinition of Class Members Other Than Methods

For class members that are not looked up (*i.e.* variables, static fields, static initializations) a renaming is performed while compiling a classbox away. Classbox WidgetsCB defines a class Component that contains a variable color accessed by a method color1() and an inner class Color. This class is refined in a classbox EnhWidgetsCB with a new variable color, a method color2() and a new inner class Color.

```
//Classbox WidgetsCB
package WidgetsCB;
public class Component {
    Color color;
    public Color color1() {
        return color;
    }
    class Color {}
}
```

```
//Classbox EnhWidgetsCB
package EnhWidgetsCB;
import WidgetsCB.Component;
refine Component {
    Color color;
    public Color color2() {
        return color;
    }
class Color {}
}
```

The resulted Java code gathers all the class members:

```
    package WidgetsCB;
    public class Component {
        WidgetsCBColor WidgetsCBcolor;
        EnhWidgetsCBColor EnhWidgetsCBcolor;
        public WidgetsCBColor foo() {
            return WidgetsCBcolor;
        }
        public EnhWidgetsCBColor bar() {
            return EnhWidgetsCBcolor;
        }
        class WidgetsCBColor { }
        class EnhWidgetsCBColor { }
    }
```

### 8.4.3 Method Redefinition

Looking up methods that are redefined requires a new method lookup semantics (Section 8.2.2). When producing Java source code, method redefinitions are compiled into one method where each redefinition is contained in a if statement used to trigger the right definition according to the current position in the execution flow of the program (cf. following section). The method paint() contained in the class Component is redefined in EnhWidgetsCB

```
//Classbox WidgetsCB
package WidgetsCB;
public class Component {
    public void update() {
        paint();
    }
    public void paint() {
        //Original paint
    }
}
```

```
//Classbox EnhWidgetsCB
package EnhWidgetsCB;
import WidgetsCB.Component;
refine Component {
    public void paint() {
        //Enhanced paint
    }
}
```

The pure Java source code produced contains only one paint() method that gathers the two implementations of the method.

```
    package WidgetsCB;
    public class Component {
        public void update() {
            paint();
        }
        public void paint() {
            if ( ClassboxInfo.methodVisible (
                    ”EnhWidgetsCB”, ”Component”, ”paint”)) {
                //Enhanced paint
            }
            if ( ClassboxInfo.methodVisible (
                    ”WidgetsCB”, ”Component”, ”paint”)) {
                //Original paint
            }
        }
    }
```

ClassboxInfo is a generated class that (i) gathers some informations about the composition of classboxes needed at runtime like a description of the classboxes that were used to produce the Java code, and (ii) offers some methods useful to introspect the method calls stack. At runtime, when the update() method is invoked, one of the two implementations is executed according to the structure of classboxes inferred from the method calls stack.

### 8.4.4 Dynamic Introspection of the Method Call Stack

Whenever a redefined method is invoked, the method call stack is reified (by using the exception handling mechanism of Java, *i.e.* Exception.getStrackTrace()) to build the structure of the classboxes.

```
//Classbox OldAppCB
package OldAppCB;
import WidgetsCB.Component;
public class OldApp {
    public static void main (String[] argv) {
        // Original paint method invoked
        new Component().update();
    }
}
```

When the main(...) method of the OldApp is invoked, before entering the paint() method the corresponding method call stack given by Java is:

```
WidgetsCB.Component.update() //Top of the stack
OldAppCB.OldApp.main() //Bottom of the stack
```

Using this stack reification and the information about the structure of classboxes kept in ClassboxInfo, the static method ClassboxInfo.methodVisible (“EnhWidgetsCB”, “Component”, “paint”) yields false, whereas ClassboxInfo.methodVisible (“WidgetsCB”, “Component”, “paint”) returns true.

NewAppCB is a client of the refined Component:

```
//Classbox NewAppCB
package NewAppCB;
```

```
import EnhWidgetsCB.Component;
public class NewApp {
    public static void main (String[] argv) {
        // Enhanced paint method invoked
        new Component().update();
    }
}
```

In a similar way, before entering the paint() method, the method call stack is:

```
WidgetsCB.Component.update() //Top of the stack
NewAppCB.NewApp.main() //Bottom of the stack
```

Because the paint() method is redefined in the classbox NewAppCB, the new implementation has to be used: the static method ClassboxInfo.methodVisible ("EnhWidgetsCB", "Component", "paint") yields true, whereas ClassboxInfo.methodVisible ("WidgetsCB", "Component", "paint") return false.

### 8.4.5 Adapting Classbox Import to Package Import

Since class imports are transitive in Classbox/J, but not in plain Java, all transitive imports must be compiled away. In the resulting Java source code, each import statement must refer to the original package that defines this class.

For example, while producing the package corresponding to the classbox NewAppCB the import statement import EnhWidgetsCB. Component is translated into import WidgetsCB.Component because the class Component is defined in WidgetsCB.

### 8.4.6 Limitations and Possible Improvements

Since the current implementation is only intended to serve as a proof of concept, we feel it is important to raise a few points concerning the limitations of this prototype.

**Native methods.** A native method is a function written in a language other than Java. Only the signature of the method is declared within the Java class. Because such methods do not contain any Java code, they cannot be rewritten using the mechanism described above. As a consequence, native methods cannot be redefined.

**Super call in a constructor.** Constructors can be redefined as well as methods. Constructor redefinitions are compiled into one single constructor following the mechanism described in Section 8.4.3. This approach is, however, limited when a constructor performs a super call. Java enforces the constructor of the superclass to be executed *before* the constructor of the subclass: the super call has to be the first statement of the constructor. Therefore the body of a constructor cannot be embedded in a if statement.

**Debugging facilities.** Even with our current approach where classboxes are compiled away, information about classboxes needed to structure the system is available (class ClassboxInfo). This information is accessible with a debugger, however it is tedious to manually retrieve the defining classbox for a given class member. Development with classboxes would be more comfortable with a classbox-aware debugger.

**Modifying the VM.** Prior to this work, we implemented two versions of the classbox model in Smalltalk: (i) by implementing a new method lookup algorithm within the VM [16], and (ii) by using bytecode transformation and method context reification on a normal VM [14]. The cost of the former strategy is about 1.1 times slower and the latter is about 1.25 times slower (these figures were obtained by comparing the execution times of a normal Smalltalk application in a classbox and a plain environment ).

The Java VM does not provide a bytecode that reifies the context of a method call. Therefore, the latter strategy cannot be implemented in Java. By modifying the Java VM to implement a new method lookup algorithm [16], we expect to achieve a similar speedup. Whereas with this approach we would need to modify the VM (which can be tedious), the advantage is that classboxes would be transparent in term of run-time cost.

## 8.5 Conclusion

In this chapter, we presented the Java implementation of classboxes by adding a few additional language constructs to Java. A classbox is a Java package where imported classes can be refined with new class members and imported classes that are refined or not to be re-imported in other classboxes. Having a Java version of the model shows that classboxes can be applied to a statically-type language like Java.

# Chapter 9

# Conclusions

In this chapter we summarize the contributions made in this dissertation, discuss the benefits of our approach, and point to directions for future work.

## 9.1 Contributions of the Dissertation

**Chapter 2.** This chapter presents a simple calculus in which classes and module systems of various OOP languages are expressed using a set of basic operators like *hide* and *fix*. The goal is to express these different models using a common formalism in order to show different semantics of modules operators. The result is a classification of various semantics, presented within a taxonomy.

**Chapter 3.** Problems with module systems is presented in Chapter 3. These problems are illustrated on 2 concrete cases. The study of a large Java library is first described. It reveals a significant amont of code duplication, broken subtype inheritance and explicit type checks and cast. A second example is intended to exhibit the requirement for a module system to support unanticipated changes.

**Chapter 4.** *Classboxes* address the problem that classical module systems do not offer the ability to add or replace a method in a class that is not defined in that module. Classboxes offer a minimal module system for object-oriented languages in which extensions (method addition and replacement) to imported classes are *locally visible*. Essentially, a classbox defines a scope within which certain entities, *i.e.*, classes, methods and variables, are defined. A classbox may *import* entities from other classboxes, and optionally extend them *without impacting the originating* classbox. Concretely, classes may be imported, and methods may be added or redefined, without affecting clients of that class in other classboxes. Local rebinding strictly limits the impact of changes to clients of the extending classbox, leading to better control over changes, while giving the illusion from a local perspective that changes are global.

By refactoring the Swing Java library, we stress-tested the classbox model by applying it to a large case study. Our new version of Swing removes (i) the incoherence in the original Swing hierarchy and (ii) the code duplication that was introduced due to the limitations of the Swing inheritance hierarchy. Moreover, while refactoring, we found the need to extend the classbox model with a new construct that allows a previous definition of a redefined method to be accessed.

We then give a formalism of classboxes by modeling classes, objects and namespaces, and we provide operations for *instantiation*, *message sending*, *self-* and *super-calls*. We used this formalism to prove properties of classboxes.

Finally, the taxonomy presented in the state of the art is revisited to take classboxes into account.

**Chapter 5.** To implement the classbox local rebinding property, a new semantics of the method lookup has to be defined. This can be implemented in two different ways: either by modifying the virtual machine, or by using bytecode manipulation and reflective features of Smalltalk. The first implementation has the advantage to completely hide the classbox machinery from a programmer. Classbox-specific dynamic operations are implemented within the virtual machine itself. The latter implementation leaves the virtual machine untouched. Development of classboxes benefits from the classical tools offered by the Smalltalk environment (inspector, debugger, ...). However, classboxes engineering can be unraveled from the programmer by using reflection. These two proof-of-concept implementation shows that classboxes can be seriously used for an industrial activity.

**Chapter 6.** As a first extension, classboxes are used to define dynamically scoped changes. This implies that import relationships between classboxes can be dynamically modified and added on the fly. Also, classboxes can dynamically be replaced by a new classbox or a set of classboxes. The result of this extension is the ability for a classbox to offer a uniform and powerful mechanism to support a simple and structural kind of aspect made from class extensions.

**Chapter 7.** As a second extension, the notion of class extension is extended with the local use of traits. Modeled as a trait, an extension defined in a classbox can be applied multiple times to different classes. This combination between traits and classboxes can be applied to express a collaborative architecture where a classbox defines a collaboration.

**Chapter 8.** As a third extension, classboxes are applied to Java, a statically dynamic language. Concretely, Java is extended with a very small number of language constructs: (i) the import relationship is transitive. (ii) A new keyword refine is added, adding new class members to imported classes, and (iii) a second new keyword original is used to access a previous method definition. A Java package is therefore regarded as a classbox.

## 9.2 Impact of Classboxes

The impact of classboxes can be put in two different perspectives: collaborations with other university/people and new research topics triggered.

### 9.2.1 Collaborations

**Aspects and classboxes.** Work is currently in progress to make aspects be scoped within a classbox. People from the PROG group, at the *Vrije Universiteit Brussel* and in the DoCoMo Euro-Labs (Germany) are interested in combining aspects and classboxes.

**Classboxes and LogicAJ.** Developed at the University of Bonn (Germany), LogicAJ [52, 51] tackles the lack of support for aspect genericity. It defines genericity to be the ability of concisely expressing aspect effects that vary depending on the context of a join point known at weave-time.

The implementation of classbox/J is based on an analysis of the method call stack. Using the aspect terminology, the machinery used is a cflow pointcut [6]. One idea is to use LogicAJ instead of the preprocessor described in Section 8.4. The expected results are a faster runtime execution (classbox/j uses a naive implementation), and a richer notion of pointcuts (classboxes support addition and redefinition of class members).

**Classboxes and traits.** Chapter 7 is an extended version of a paper [67] done in collaboration with the Ecole des Mines de Nantes (France).

### 9.2.2 New Research Topics

**Aspect composition using explicit context.** Work with classboxes yielded a new notion of context applied to aspect. This project is currently lead by the Center for Web Research, DCC, University of Chile, Santiago (Chile).

**Changeboxes.** We plan to enhance the notion of refinement in order to enable the use of classboxes as a way to express general changes that can be applied to a system (and not just additions or redefinitions of class members). Some work is currently in progress at the University of Lugano (Switzerland).

**Explicit context to model extension.** Lumpe and Schneider have been working on using explicit contexts to express composition of general abstraction [57]. They illustrate their intention by formalizing classboxes in LambdaF based on first-class namespaces.

## 9.3 Future Work

**Typed module calculus.** The presented calculus in Chapter 2 is untyped. As a future work we plan to explore typing rules for this calculus in order to express, for instance, which compositions of modules are type safe. Virtual classes (like in Eiffel) represent an unchecked use of covariance, which is not type-safe [23], whereas the gbeta approach was always based on checking for covariance (which is possible because, in contrast to Eiffel, covariance is always explicitly declared).

We plan to explore typing rules for this calculus in order to express, for instance, which compositions of modules are type safe. Virtual classes (like in Eiffel) represent an unchecked use of covariance, which is not type-safe [23], whereas the gbeta approach was always based on checking for covariance (which is possible because, in contrast to Eiffel, covariance is always explicitly declared).

Our main focus in this Chapter was expressing import and extend relationship. Our future work is to apply our approach to other systems such as Modula-3 [27], ModularJava [30], JavaMod [4] and Nested Inheritance [70] as they offer the notion of explicit interface.

**Module systems and evolution.** The taxonomy presented in Chapter 2 classify different semantics of module operators. A really exciting extension of this formalism is to apply it to modeling evolution. We believe that inter-module operator semantics have a great impact on software evolution.

# Appendix A

# Implementation of the Module Calculus

This chapter contains one possible implementation in the Scheme programming language of the module calculus presented in Chapter 2.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Module Calculus
;;; (C) Alexandre Bergel
;;; (C) SCG, University of Bern
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Testing framework
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define *counter* 0)
; If not (equals? value expected), then raise an error.
(define (assert value expected)
  (let ((failed #f))
    (set! *counter* (+ 1 *counter*))
    (display (string-append "Test " (number->string *counter*)))
    (if (not (equal? value expected))
        (set! failed #t))
    (display (if failed " failed" " passed"))
    (newline)
    (if failed "Error, test failed!")))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Environment
;;;     An environment is a of bindings (key value)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define empty-env '())

; (select f L) returns a list of elements l that
; belongs to L and for which (f l) is true.
```

```
(define (select f L)
  (cond ((null? L) L)
        ((f (car L))(cons (car L) (select f (cdr L))))
        (else (select f (cdr L)))))

; Return keys of an environment e
(define (keys e) (map (lambda (el) (car el)) e))

; e1 override e2
(define (override e1 e2) (append e1 e2))

; Return a value associated to a name in an environment
(define (get env name) (cadr (assoc name env)))

; Extend an environment with a list of element
; We make sure that there is no duplicata.
(define (extend env el)
  (let ((name (car el)))
    (if (not (assoc name env))
        (override (list el) env)
        (error "Already existing element" el env))))

; Remove a binding from an env.
(define (exclude env name)
  (cond ((null? env) env)
        ((eqv? name (caar env)) (exclude (cdr env) name))
        (else (cons (car env) (exclude (cdr env) name)))))

(assert (keys (extend (extend empty '(a 1)) '(b 2)))
        '(b a))
(assert (exclude (extend (extend empty '(a 1)) '(b 2)) 'b)
        (extend empty '(a 1)))
(assert (get (override '((a x)(b y)) '((b z) (c w))) 'a)
        'x)
(assert (get (override '((a x)(b y)) '((b z) (c w))) 'c)
        'w)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Modules
;; Example of a module:
;;(define m (lambda (e) (add '(a 1) (add '(b 2) empty))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (mextend m el) (lambda (e) (extend (m e) el)))
(define (moverride m1 m2) (lambda (e) (override (m1 e) (m2 e))))
(define (mexclude m name) (lambda (e) (exclude (m e) name)))

; Return a function that accepts a module and return a new module
; that does not contain a binding.
(define (hide a)
  (lambda (m)
    (lambda (e)
      ((mexclude m a) (override `((,a ,(get (fix m) a))) e)))))

; Same than hide, but get a list of keys as argument
(define (hide2 L)
  (if (null? (cdr L))
      (hide (car L))
      (lambda (f) ((hide (car L)) ((hide2 (cdr L)) f)))))


(define (mkeys m) (keys (fix m)))

;; Trick to simulate a fix point
;(define (fix m) (m (m (m (m (m (m (m (m (m (m m)))))))))))
(define (fix m) (m (m m)))


(assert (get (fix (lambda (e) `((a x)(b y)))) 'a)
        'x)

(assert (let ((m1 (lambda (e) `((a 1) (b ,e))))
              (m2 (lambda (e) `((a 2)))))
          (get (get (fix (moverride m2 m1)) 'b) 'a))
        2)

(assert (let*((m1 (lambda (e) `((a 1) (b ,e))))
              (m2 (lambda (e) `((a 2))))
              (m3 (lambda (e) `((b ,(get (fix m1) 'b))))))
          (get (get (fix (moverride m2 m3)) 'b) 'a))
        1)

(assert (let*((m1 (lambda (e) `((a 1) (b ,e)))))
          (keys (fix ((hide 'a) m1))))
        '(b))

(assert (let*((m1 (lambda (e) `((a 1) (b ,e)))))
          (get (get (fix ((hide 'a) m1)) 'b) 'a))
        1)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Classes
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define GraphicsModule
  (lambda (e)
    `((Point ((x 0)
              (y 0)
              (moveBy ,(lambda (dx)
                         (lambda (dy)
                           (lambda (self)
                             (override
                               (list (list 'x (+ (get self 'x) dx))
                                     (list 'y (+ (get self 'y) dy)))
                              self))))))
      (PointFactory ((newPoint ,(lambda (self) (get e 'Point)))))))

(assert (let ((o (get (fix GraphicsModule) 'Point)))
          (+ (get o 'y) (get o 'x)))
        0)
(assert (let ((o (get (fix GraphicsModule) 'Point)))
          (set! o (((get o 'moveBy) 2) 3) o))
          (get o 'x))
        2)
(assert (let ((o (get (fix GraphicsModule) 'Point)))
          (set! o (((get o 'moveBy) 2) 3) o))
          (set! o (((get o 'moveBy) 2) 3) o))
          (+ (get o 'x) (get o 'y)))
        10)
(assert (let* ((f (get (fix GraphicsModule) 'PointFactory))
               (p ((get f 'newPoint) 'notUsed)))
          (set! p (((get p 'moveBy) 2) 3) p))
          (set! p (((get p 'moveBy) 2) 3) p))
          (+ (get p 'x) (get p 'y)))
        10)

(define extendClass
  (lambda (m)
    (lambda (sup)
      (lambda (c)
        (lambda (d)
          (lambda (e)
            (extend (m e)
                    (list c
                          (override d
                                    (get (m e) sup))))))))))

(define colorExtensions
  `((color ())
    (setColor ,(lambda (newCol)
                 (lambda (self)
                   (override
                     (list (list 'color newCol))
                     self))))))
(define ColoredGraphicsModule
  ((((extendClass GraphicsModule) 'Point)
    'ColoredPoint) colorExtensions))

(assert (let* ((f (get (fix ColoredGraphicsModule) 'PointFactory))
               (p ((get f 'newPoint) 'notUsed)))
          (set! p (((get p 'moveBy) 2) 3) p))
          (set! p (((get p 'moveBy) 2) 3) p))
          (+ (get p 'x) (get p 'y)))
        10)
(assert (let* ((p (get (fix ColoredGraphicsModule) 'ColoredPoint)))
          (set! p (((get p 'moveBy) 2) 3) p))
          (set! p (((get p 'moveBy) 2) 3) p))
          (null? (get p 'color)))
        #t)
(assert (let* ((p (get (fix ColoredGraphicsModule) 'ColoredPoint)))
          (set! p ((((get p 'moveBy) 2) 3) p))
          (set! p ((((get p 'moveBy) 2) 3) p))
          (set! p (((get p 'setColor) 'blue) p))
          (get p 'color))
        'blue)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Java
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define importClass
  (lambda (mt)
    (lambda (ms)
      (lambda (c)
        ((hide c) (mextend mt (list c (get (fix ms) c))))))))

(define graphics
  (lambda (e)
    `((Point ((x 0)
              (y 0)
              (moveBy ,(lambda (dx)
                         (lambda (dy)
                           (lambda (self)
                             (override
                               (list (list 'x (+ (get self 'x) dx))
                                     (list 'y (+ (get self 'y) dy)))
                               self)))))))
      (PointFactory ((newPoint ,(lambda (self) (get e 'Point))))))))

(define graphics2
  (((importClass (lambda (e)
                   `((Point empty)
                     (App ((main ,(lambda (self)
                                    (get e 'PointFactory))))))))
    graphics) 'PointFactory))

(assert (let* ((a (get (fix graphics2) 'App))
               (f ((get a 'main) 'notUsed))
               (p ((get f 'newPoint) 'notUsed)))
          (set! p ((((get p 'moveBy) 2) 3) p))
          (set! p ((((get p 'moveBy) 2) 3) p))
          (+ (get p 'x) (get p 'y)))
        10)

(define importPackage
  (lambda (mt)
    (lambda (ms)
      ((hide2 (mkeys ms))
       (lambda (e) (override (mt e) (fix ms)))))))

(define graphics2
  ((importPackage
    (lambda (e) `((Point empty)
                  (App ((main ,(lambda (self)
                                 (get e 'PointFactory)))))))
                   graphics))

(assert (let* ((a (get (fix graphics2) 'App))
               (f ((get a 'main) 'notUsed))
               (p ((get f 'newPoint) 'notUsed)))
          (set! p (((get p 'moveBy) 2) 3) p))
          (set! p (((get p 'moveBy) 2) 3) p))
          (+ (get p 'x) (get p 'y)))
        10)

(define private (lambda (m) (lambda (c) ((hide c) m))))

(define graphics2
  (((importClass
     (lambda (e) `((Point empty)
                   (App ((main ,(lambda (self)
                                  (get e 'PointFactory)))))))
    graphics)
   'PointFactory))

(assert (mkeys ((private graphics2) 'Point))
        '(App))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; C#
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define usingClassAs
  (lambda (mt)
    (lambda (ms)
      (lambda (a)
        (lambda (c)
          ((hide a) (mextend mt (list a (get (fix ms) c)))))))))

(define graphics
  (lambda (e)
    `((Point ((x 0)
              (y 0)
              (moveBy ,(lambda (dx)
                         (lambda (dy)
                           (lambda (self)
                             (override
                               (list (list 'x (+ (get self 'x) dx))
                                     (list 'y (+ (get self 'y) dy)))
                               self)))))))
      (PointFactory ((newPoint ,(lambda (self) (get e 'Point))))))))

(define graphics2
  ((((usingClassAs (lambda (e)
                     `((App ((main ,(lambda (self) (get e 'PF))))))))
     graphics) 'PF) 'PointFactory))

(assert (let* ((a (get (fix graphics2) 'App))
               (f ((get a 'main) 'notUsed))
               (p ((get f 'newPoint) 'notUsed)))
          (set! p ((((get p 'moveBy) 2) 3) p))
          (set! p ((((get p 'moveBy) 2) 3) p))
          (+ (get p 'x) (get p 'y)))
        10)

(define usingClass (lambda (mt)
                      (lambda (ms)
                        (lambda (c)
                            ((((usingClassAs mt) ms) c) c)))))

(define graphics2
  (((usingClass
     (lambda (e)
       `((App ((main ,(lambda (self) (get e 'PointFactory))))))))
    graphics)
   'PointFactory))

(assert (let* ((a (get (fix graphics2) 'App))
               (f ((get a 'main) 'notUsed))
               (p ((get f 'newPoint) 'notUsed)))
          (set! p ((((get p 'moveBy) 2) 3) p))
          (set! p ((((get p 'moveBy) 2) 3) p))
          (+ (get p 'x) (get p 'y)))
        10)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Ruby
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define newClassWithMixin
  (lambda (mt)
    (lambda (mixin)
      (lambda (c)
        (lambda (d)
          (mextend mt `(,c ,(fix
                              (lambda (s)
                                (override d (mixin s))))))))))

(define colorMixin (lambda (e) colorExtensions))
(define graphics
  ((((newClassWithMixin (lambda (x) empty)) colorMixin) 'Point)
   `((x 0) (y 0)
           (moveBy ,(lambda (dx)
                      (lambda (dy)
                        (lambda (self)
                          (override
                           (list (list 'x (+ (get self 'x) dx))
                                 (list 'y (+ (get self 'y) dy)))
                           self))))))))

(assert (let* ((p (get (fix graphics) 'Point)))
          (set! p ((((get p 'moveBy) 2) 3) p))
          (set! p ((((get p 'moveBy) 2) 3) p))
          (+ (get p 'x) (get p 'y)))
        10)

(assert (let* ((p (get (fix graphics) 'Point)))
          (set! p (((get p 'setColor) 'blue) p))
          (get p 'color))
        'blue)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Selector Namespace
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define snextend
  (lambda (mt)
    (lambda (ms)
      (lambda (c)
        (lambda (d)
          (mextend mt `(,c ,(override (fix d)
                                      (get (fix ms) c)))))))))

(define import (lambda (mt)
                 (lambda (ms)
                   (lambda (c)
                     (extend mt ms c empty)))))

(define english ((((snextend (lambda (x) empty))
                   (lambda (e)`((Object '())))))
                  'Object)
                 (lambda (s) (list (list 'printString 'englishVersion)
                                   (list 'printOnStream
                                         (lambda (self)
                                           (get (override s self)
                                                'printString)))))))


(define german ((((snextend (lambda (p) empty)) english) 'Object)
                (lambda (s)
                  `((printString ,(lambda (self) 'germanVersion))))))

(assert ((get (get (fix german) 'Object) 'printString) 'notUsed)
        'germanVersion)

(assert (let ((o (get (fix english) 'Object)))
          ((get o 'printOnStream) o))
        'englishVersion)

(assert (let ((o (get (fix german) 'Object)))
          ((get o 'printOnStream) o))
        'englishVersion)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Virtual Classes
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define extendEncapsulated
  (lambda (mt) (lambda (ms) (moverride mt ms))))

(define extendInner
  (lambda (mt)
    (lambda (c)
      (lambda (d)
        (lambda (e)
          (extend ((mexclude mt c) e)
                  `(,c ,(override d (get (mt e) c)))))))))

(define Graphics
  (lambda (e)
    `((Point ((x 0)
              (y 0)
              (moveBy ,(lambda (dx)
                         (lambda (dy)
                           (lambda (self)
                             (override
                               (list (list 'x (+ (get self 'x) dx))
                                     (list 'y (+ (get self 'y) dy)))
                               self)))))))
      (PointFactory ((newPoint ,(lambda (self) (get e 'Point))))))))

(define ColoredGraphics
  (((extendInner ((extendEncapsulated (lambda (x) empty))
                  Graphics)) 'Point) colorExtensions))

(assert (let* ((f (get (fix Graphics) 'PointFactory))
               (p ((get f 'newPoint) 'notUsed)))
          (set! p ((((get p 'moveBy) 2) 3) p))
          (set! p ((((get p 'moveBy) 2) 3) p))
          (+ (get p 'x) (get p 'y)))
        10)

(assert (let* ((f (get (fix Graphics) 'PointFactory))
               (p ((get f 'newPoint) 'notUsed)))
          (keys p))
        '(x y moveBy))

(assert (let* ((f (get (fix ColoredGraphics) 'PointFactory))
               (p ((get f 'newPoint) 'notUsed)))
          (keys p))
        '(color setColor x y moveBy))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Classboxes
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define cbextend
  (lambda (mt)
    (lambda (ms)
      (lambda (c)
        (lambda (d)
          (lambda (e)
            (extend (mt e)
                    `(,c ,(override d (get (ms e) c))))))))))

(define WidgetsClassbox
  ((((extendClass
      (lambda (e)
        `((Morph ((paint old-paint)
                  (repaint ,(lambda (self) (get self 'paint)))))))
     'Morph) 'Button) ()))

(define EnhWidgetsClassbox
  ((((extendClass ((((cbextend (lambda (x) empty))
                     WidgetsClassbox) 'Morph)
                   '((paint new-paint)))) 'Morph) 'Button) '()))

(assert (let ((b (get (fix WidgetsClassbox) 'Button)))
          'old-paint (get b 'paint))
        'old-paint)
(assert (let ((b (get (fix WidgetsClassbox) 'Button)))
          ((get b 'repaint) b))
        'old-paint)
(assert (let ((b (get (fix EnhWidgetsClassbox) 'Morph)))
          ((get b 'repaint) b))
        'new-paint)
(assert (let ((b (get (fix EnhWidgetsClassbox) 'Button)))
          ((get b 'repaint) b))
        'new-paint)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Units
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define link
  (lambda (mt)
    (lambda (ms)
      (lambda (a)
        (lambda (c)
          (lambda (e) (extend (mt e) `(,a ,(get (ms e) c))))))))

(define widgets
  (lambda (e)
    `((Point ((x 0) (y 0)))
      (Circle ((radius 0) (center 0))))))

(define widgetsFactory
  (lambda (e)
    `((Factory ((newPoint ,(lambda (self) (get e 'Point)))
                (newCircle ,(lambda (self) (get e 'Circle)))))))

(define compound1 ((((link ((((link widgetsFactory) widgets)
                              'Point)
                             'Point))
                     widgets)
                    'Circle) 'Circle))

(assert (let ((p (get (fix compound1) 'Point))
              (c (get (fix compound1) 'Circle)))
          (get p 'x))
        0)
(assert (let* ((f (get (fix compound1) 'Factory))
               (p ((get f 'newPoint) 'notUsed)))
          (get p 'x))
        0)


(define coloredWidgets
  (lambda (e)
    `((ColoredPoint ((color 'blue) (x 0) (y 0)))
      (ColoredCircle ((color 'blue) (radius 0) (center 0))))))

(define compound2 ((((link ((((link widgetsFactory) coloredWidgets)
                              'Point)
                             'ColoredPoint))
                     coloredWidgets) 'Circle) 'ColoredCircle))

(assert (let ((p (get (fix compound2) 'Point))
              (c (get (fix compound2) 'Circle)))
          (keys p))
        '(color x y))

(assert (let* ((f (get (fix compound2) 'Factory))
               (p ((get f 'newPoint) 'notUsed)))
          (keys p))
        '(color x y))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; MixJuice
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define mjextends (lambda (mt) (lambda (ms) (moverride mt ms))))
(define refineClass
  (lambda (mt)
    (lambda (c)
      (lambda (d)
        (lambda (e)
          (extend ((mexclude mt c) e)
                  `(,c ,(override d (get (mt e) c)))))))))

(define mjpoint
  (lambda (e)
    `((Point ((x 0)
              (y 0)
              (moveBy ,(lambda (dx)
                         (lambda (dy)
                           (lambda (self)
                             (override
                               (list (list 'x (+ dx (get self 'x)))
                                     (list 'y (+ dy (get self 'y))))
                               self)))))
              (toString ,(lambda (self)
                           (string-append
                            "point "
                            (number->string (get self 'x))
                            ", "
                            (number->string (get self 'y))))))))))

(define mjcoloredPoint
  (((refineClass ((mjextends (lambda (x) empty)) mjpoint)) 'Point)
   `((color 'blue)
     (toString ,(lambda (self)
                  (string-append "colored point "
                                 (number->string (get self 'x)) ", "
                                 (number->string (get self 'y))))))))

(assert (let* ((p (get (fix mjpoint) 'Point)))
          ((get p 'toString) p))
        "point 0, 0")

(assert (let* ((p (get (fix mjcoloredPoint) 'Point)))
          ((get p 'toString) p))
        "colored point 0, 0")
```

# Bibliography

[1] O. Agesen, L. Bak, C. Chambers, B.-W. Chan, U. Hölzle, J. Maloney, R. B. Smith, D. Ungar, and M. Wolczko. *The SELF 4.0 Programmer's Reference Manual.* Sun Microsystems, 1995.

[2] D. Ancona and E. Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, Aug. 1998.

[3] D. Ancona and E. Zucca. A primitive calculus for module systems. In G. Nadathur, editor, *Principles and Practice of Declarative Programming*, number 1702 in LNCS, pages 62–79. Springer Verlag, 1999.

[4] D. Ancona and E. Zucca. True modules for Java-like languages. In J. L. Knudsen, editor, *ECOOP 2001*, number 2072 in LNCS, pages 354–380. Springer Verlag, 2001.

[5] K. Arnold and J. Gosling. *The Java Programming Language.* Addison Wesley, 1996.

[6] AspectJ home page. http://eclipse.org/aspectj/.

[7] AWT API. http://java.sun.com/j2se/1.3/docs/api/java/awt/package-summary.html.

[8] D. Batory and B. J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering (special issue on Software Reuse)*, pages 62–87, Feb. 1997.

[9] D. Batory, J. Liu, and J. N. Sarvela. Refinements and multi-dimensional separation of concerns. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 48–57, New York, NY, USA, 2003. ACM Press.

[10] A. Bergel and S. Ducasse. Scoped and dynamic aspects with classboxes. *RSTI – L'Objet (programmation par aspects)*, 11(3):53–68, 2005.

[11] A. Bergel and S. Ducasse. Supporting unanticipated changes with traits and classboxes. In *Proceedings of Net.ObjectDays (NODE'05)*, Erfurt, Germany, 2005.

[12] A. Bergel, S. Ducasse, and O. Nierstrasz. Analyzing module diversity. *Journal of Universal Computer Science*, 2005. To appear.

[13] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 177–189, San Diego, CA, USA, 2005. ACM Press.

[14] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling visibility of class extensions. *Computer Languages, Systems and Structures*, 31(3-4):107–126, May 2005.

[15] A. Bergel, S. Ducasse, and R. Wuyts. The Classbox module system. In *Proceedings of the ECOOP '03 Workshop on Object-oriented Language Engineering for the Post-Java Era*, July 2003.

[16] A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of JMLC 2003 (Joint Modular Languages Conference)*, volume 2789 of *LNCS*, pages 122–131. Springer-Verlag, 2003. Best Award Paper.

[17] V. Bono, A. Patel, and V. Shmatikov. A core calculus of classes and mixins. In R. Guerraoui, editor, *Proceedings ECOOP '99*, volume 1628 of *LNCS*, pages 43–66, Lisbon, Portugal, June 1999. Springer-Verlag.

[18] N. Bouraqadi. Concern oriented programming using reflection. In *Workshop on Advanced Separation of Concerns – OOSPLA 2000*, 2000.

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

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

[21] G. Bracha and G. Lindstrom. Modularity meets inheritance. Uucs-91-017, University of Utah, Dept. Comp. Sci., Oct. 1991.

[22] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 282–290, Apr. 1992.

[23] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *Proceedings ECOOP '98*, pages 523–549. Springer-Verlag, 1998.

[24] C#. http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[25] P. S. Canning, W. Cook, W. L. Hill, J. C. Mitchell, and W. G. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, pages 273–280, Sept. 1989.

[26] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[27] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, Aug. 1992.

[28] C. Clifton, G. T. Leavens, C. Chambers, and T. 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.

[29] W. R. Cook. *A Denotational Semantics of Inheritance*. Ph.D. thesis, Department of Computer Science, Brown University, Providence, RI, May 1989.

[30] J. Corwin, D. F. Bacon, D. Grove, and C. Murthy. MJ: a rational module system for Java and its applications. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 241–254. ACM Press, 2003.

[31] 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.

[32] M. Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001*, Oct. 2001.

[33] DrScheme. http://www.drscheme.org/.

[34] S. Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[35] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *Transactions on Programming Languages and Systems*, 2005. To appear.

[36] E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[37] E. Ernst. Propagating class and method combination. In R. Guerraoui, editor, *Proceedings ECOOP '99*, volume 1628 of *LNCS*, pages 67–91, Lisbon, Portugal, June 1999. Springer-Verlag.

[38] E. Ernst. Family polymorphism. In J. L. Knudsen, editor, *ECOOP 2001*, number 2072 in LNCS, pages 303–326. Springer Verlag, 2001.

[39] R. B. Findler and M. 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.

[40] M. Flatt and M. Felleisen. Units: Cool modules for hot languages. In *Proceedings of PLDI '98 Conference on Programming Language Design and Implementation*, pages 236–248. ACM Press, 1998.

[41] 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.

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

[43] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

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

[45] Y. Ichisugi and A. Tanaka. Difference-based modules: A class independent module mechanism. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, Malaga, Spain, June 2002. Springer Verlag.

[46] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. 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, Nov. 1997.

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

[48] G. Kiczales. Aspect-oriented programming: A position paper from the xerox PARC aspect-oriented programming project. In M. Muehlhauser, editor, *Special Issues in Object-Oriented Programming*. ?, 1996.

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

[50] G. Kniesel. *Darwin – Dynamic Object-Based Inheritance with Subtyping*. PhD thesis, CS Dept. III, University of Bonn, Germany, 2000.

[51] G. Kniesel and T. Rho. Generic aspect languages - needs, options and challenges, jfdlpa 2005. In *Proceedings of JFDLPA 2005*. Hermes Paris, Sept. 2005.

[52] G. Kniesel, T. Rho, and S. Hanenberg. Evolvable pattern implementations need generic aspects, proc. of ecoop'2004 workshop on reflection, aop and meta-data for software evolution. In *Proc. of ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution*. Springer Verlag, June 2004.

[53] B. B. Kristensen, O. L. Madsen, B. Moller-Pedersen, and K. Nygaard. The BETA programming language. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, Cambridge, Mass., 1987.

[54] W. LaLonde and J. Pugh. Subclassing $\neq$ Subtyping $\neq$ Is-a. *Journal of Object-Oriented Programming*, 3(5):57–62, Jan. 1991.

[55] X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[56] R. E. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technlogies. In *Proceedings ECOOP 2005*, 2005.

[57] M. Lumpe and J.-G. Schneider. Classboxes – An Experiment in Modeling Compositional Abstractions using Explicit Contexts. In M. Barnett, S. Edwards, D. Giannakopoulou, G. T. Leavens, and N. Sharygina, editors, *Proceedings of ESEC '05 Workshop on Specification and Verification of Component-Based Systems (SAVCBS '05)*, pages 47–54, Lisbon, Portugal, Sept. 2005.

[58] O. L. Madsen and B. Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 397–406, Oct. 1989.

[59] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic java classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 337–361. Springer-Verlag, 2000.

[60] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned java. In *Proceedings OOPSLA 2001, ACM SIGPLAN Notices*, pages 211–222, Oct. 2001.

[61] S. McDirmid and W. C. Hsieh. Aspect-oriented programming with jiazzi. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 70–79, New York, NY, USA, 2003. ACM Press.

[62] 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.

[63] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[64] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[65] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99. ACM Press, 2003.

[66] T. Millstein, M. Reay, and C. 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.

[67] F. Minjat, A. Bergel, P. Cointe, and S. Ducasse. Mise en symbiose des traits et des classboxes : Application à l'expression des collaborations. In *Proceedings of LMO 2005*, volume 11, pages 33–46, Bern, Switzerland, 2005.

[68] E. Miranda, D. Leibs, and R. Wuyts. Parcels: a fast and feature-rich binary deployment technology. *Computer Languages, Systems and Structures*, 31(3-4):165–182, May 2005.

[69] H. Mössenböck. *Object-Oriented Programming in Oberon-2*. Springer-Verlag, 1993.

[70] N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 99–115. ACM Press, 2004.

[71] Ocaml. http://caml.inria.fr/.

[72] A. Oliva and L. E. Buzato. The design and implementation of Guarana. In *USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, 1999.

[73] H. Ossher and P. Tarr. Hyper/J: multi-dimensional separation of concerns for java. In *Proceedings of the 22nd international conference on Software engineering*, pages 734–737. ACM Press, 2000.

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

[75] L. J. Pinson and R. S. Wiener. *Objective-C*. Addison Wesley, 1988.

[76] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for java. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 100–109. ACM Press, 2003.

[77] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147. ACM Press, 2002.

[78] C. Prehofer. Feature-oriented programming: A fresh look at objects. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 419–443, Jyväskylä, June 1997. Springer-Verlag.

[79] B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of European Conference on Object-Oriented Programming*, volume 2374, pages 205–230. Springer-Verlag, 2002.

[80] F. Rivard. *Évolution du comportement des objets dans les langages à classes réflexifs*. PhD thesis, Ecole des Mines de Nantes, Université de Nantes, France, 1997.

[81] T. Saridakis. Managing unsolicited events in component-based software. In *Workshop on Component Models for Dependable Systems*, Aug. 2004. To appear.

[82] Y. Sato and S. Chiba. Loosely-separated "sister" namespaces in java. In *Proceedings ECOOP 2005*, 2005.

[83] Scala home page. http://lamp.epfl.ch/scala/.

[84] M. Serrano. Wide classes. In R. Guerraoui, editor, *Proceedings ECOOP '99*, volume 1628 of *LNCS*, pages 391–415, Lisbon, Portugal, June 1999. Springer-Verlag.

[85] D. Simmons. Smallscript, 2002. http://www.smallscript.com.

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

[87] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM TOSEM*, 11(2):215–255, Apr. 2002.

[88] P. Steyaert, W. Codenie, T. D'Hondt, K. D. Hondt, C. Lucas, and M. V. Limberghen. Nested mixin-methods in agora. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 197–219, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[89] Swing api. http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/package-summary.html.

[90] S. T. Taft. Ada 9x: From abstraction-oriented to object-oriented. In *Proceedings OOPSLA '93*, volume 28, pages 127–143, Oct. 1993.

[91] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of ICSE '99*, pages 107–119, Los Angeles CA, USA, 1999.

[92] D. Thomas and A. Hunt. *Programming Ruby*. Addison Wesley, 2001.

[93] M. Torgersen. The expression problem revisited — four new solutions using generics. In M. Odersky, editor, *Proceedings ECOOP 2004*, LNCS, Oslo, Norway, June 2004. Springer-Verlag.

[94] M. VanHilst and D. 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.

[95] M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. In *Proceedings OOPSLA '96*, pages 359–369. ACM Press, 1996.

[96] Cincom Smalltalk, Sept. 2003. http://www.cincom.com/scripts/smalltalk.dll/.

[97] A. Wirfs-Brock and B. Wilkerson. An overview of modular Smalltalk. In *Proceedings OOPSLA '88*, pages 123–134, Nov. 1988.

[98] N. Wirth. *Programming in Modula-2*. Springer-Verlag, Berlin, 1983.

[99] M. Zenger. Evolving software with extensible modules. In *International Workshop on Unanticipated Software Evolution*, Malaga, Spain, June 2002.

[100] M. Zenger. Type-safe prototype-based component evolution. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, pages 470–497, Malaga, Spain, June 2002. Springer Verlag.

# Curriculum Vitæ

## Personal Information

| | |
|---|---|
| Name: | Alexandre Bergel |
| Nationality: | Française |
| Date of Birth: | March 23, 1978 |
| Place of Birth: | Nice, France |

## Education

| | |
|---|---|
| 2002 - 2005: | *Ph.D. in Computer Science* in the Software Composition Group, University of Bern, Switzerland. Subject of the Ph.D. thesis: "Controlling visibility of Class Extensions" |
| 2000 - 2001: | *Master in Computer Science* at the University of Nice-Sophia Antipolis, France and University of Adelaide, South-Australia, Australia. Subject of the Master thesis: "Définition et compilation d'un langage à affectation unique vers un réseau de processus" |
| 1999 - 2000: | *Maîtrise d'Informatique*. University of Nice-Sophia Antipolis. |
| 1998 - 1999: | *Licence d'Informatique*. University of Nice-Sophia Antipolis. |
| 1996 - 1999: | *DEUG Mathématique, Informatique, et Applications aux Sciences*. University of Nice-Sophia Antipolis. |