Enabling White-Box Reuse in a Pure Composition Language

Diplomarbeit

der Philosophisch-naturwissenschaftlichen Fakultät

der Universität Bern

vorgelegt von

Andreas Schlapbach

Dezember 2002

Leiter der Arbeit:

Prof. Dr. O. Nierstrasz

Nathanael Schärli

Institut für Informatik und angewandte Mathematik

Abstract

Inheritance is a key concept of object-oriented programming languages, features such as conceptual modeling and reusability are largely accredited to it. While many useful components have been, and will be, developed in this paradigm, the form of white-box reuse offered by inheritance has a fundamental flaw: reusing components by inheritance requires an understanding of the internals of the components. We can not treat components of object-oriented languages as black-box entities, inheritance breaks encapsulation and introduces subtle dependencies between base and extending classes.

Component-oriented programming addresses this problem by shifting away from programming towards software composition. We build applications by scripting components. Instead of overriding the internals of a component, we focus on composing its interfaces only. This form of black-box reuse leads to a flexible and extendible architecture with reusable components.

In this master's thesis we propose a migration strategy from class inheritance – a white-box form of reuse – to component composition as a black-box form of reuse. We present a language extension that gives us the power of inheritance combined with the ease of scripting. It enables us to reuse Java components using inheritance in JPiccola – a small, pure and general composition language implemented on the Java platform – at a high level of abstraction. Using the services provided by the language extension we can seamlessly generate interfaces and subclasses from JPiccola. This capability greatly increases the number of components scriptable from JPiccola. To validate the usefulness of our language extension we demonstrate how we can script various Java components by defining services and compositional styles. We thus turn white-box components of Java into black-box components in Piccola.

Acknowledgments

First of all, I would like to thank Prof. Dr. Nierstrasz, head of the SCG, for giving me the opportunity to work in his group and for fruitful discussions about software composition, Piccola and scripting in general.

Furthermore, I would like to thank the Piccola people of the SCG: Dr. Franz Achermann helped me to get my master's thesis started. Too bad that he had to leave university for the Real World. Nathanael Schärli gave me guidance and support throughout my work even when he was thousands of kilometers away. Thanks a lot and I must say I enjoy your sense of humor. With Stefan Kneubühl I spent some fine hours of hacking and profiling JPiccola.

Thanks also go to Matthias Rieger with whom I talked – apart from design, typography and films – a lot about the meta aspects of writing a master's thesis. Merrin Dimmock Jaggi helped me to improve my English.

Finally, I would like to thank Bram Moolenaar for making a wonderful piece of software even better.

> To me `vi` is zen.
> To use `vi` is to practice zen.
> Every command is a koan.
> Profound to the user, unintelligible to the uninitiated.
> You discover truth every time you use it.
>
> – Satish Reddy

> Etz chaim hee lamachazikim bah. *(It's a tree of life for those who grasp it.)*
>
> – Hebrew proverb

Andreas Schlapbach

December 2002

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Inheritance is a key concept of object-oriented languages [Weg87]. Inheritance is often regarded as the feature that distinguishes object-oriented programming from other modern programming paradigms, and many of the alleged benefits of object-oriented programming, such as improved conceptual modeling and reusability, are largely accredited to it [Tai96]. Today, object-oriented programming is the dominant paradigm and many frameworks and powerful components have been and will be developed.

However, in recent years there has been growing critique that object-oriented programming has failed to provide software architectures based on reusable and extendible components [NTdMS91, PS96]. More specifically, it has been claimed that inheritance is not ideally suited for composing components [WS96]. This form of white-box reuse focuses on adapting a mismatched component by either changing or overriding its internal specification. In this respect, inheritance breaks encapsulation [Sny86]. In order to correctly reuse a component using inheritance we have to understand its internals and be extremely careful not to break subtle dependencies between base and extending classes.

To overcome these problems, a new paradigm of software developing has been proposed [NM95, Szy98]: component-oriented programming shifts away from programming towards software composition. Instead of overriding internals of a component, it focuses on adapting only the interface of a component and on scripting these components [Ous98]. In this paradigm, we treat components as black-box entities that provide a clean interface and hide their internals completely. Applications are built by scripting these components. This approach makes the architecture explicit and provides an open, flexible and reusable piece of software.

We would like to have the best of both paradigms: the power of inheritance combined with the ease of scripting. We would like to easily use components written in a object-oriented language using inheritance and turn them into components with an interface adhering to

a compositional style and well-defined requirements. These components we then script seamlessly.

1.2 Goal

In this master's thesis, we explore how we can integrate inheritance as a white-box form of reuse into JPiccola, a small, pure and general composition language implemented on the Java platform [AN01]. Our goal is to use inheritance to gain access to the components provided by Java frameworks, and then wrap and script them from Piccola.

We address a number of questions: what are the roles of inheritance? How do we combine white-box and black-box form of reuse? How do we integrate our approach seamlessly into JPiccola? How well suited are existing components for this form of reuse? What are the requirements and what technical problems do we face if we generate and load Java classes at runtime?

1.3 Contribution

We reason about the roles of inheritance in object-oriented languages. We show that subclassing as a form of code reuse can break encapsulation by introducing subtle dependencies between base and extending classes. We thus argue that we should minimize the use of inheritance and propose a migration strategy from class inheritance – a white-box form of reuse – to component composition as a black-box form of reuse.

We introduce a language extension that enables white-box reuse by inheritance in JPiccola. The key idea of the language extension is to generate Java classes whose objects delegate all method calls to Piccola services. This language extension enables us to reuse Java components by inheritance from JPiccola and control their behavior entirely in JPiccola.

We discuss the technical problems faced when we want to subclass at runtime on the Java platform. We analyze how other Java-based scripting languages address these problems and highlight the Java Virtual Machine and the structure of a Java class file. We especially look in detail at the class loading and verifying mechanism of the Java Virtual Machine.

We validate our language extension by scripting various components of Java frameworks. We illustrate how we can use Java components in JPiccola and turn them into services and define compositional styles on top of them. We thus script Java frameworks from JPiccola. In other words, we show how to use our language extension to turn black-box components into white-box components. Additionally, we discuss the characteristics that distinguish a mature black-box framework from a bundle of classes.

1.4 Structure

The rest of this master's thesis is structured as follows: in Chapter 2 we present a framework for component based software development based on the maxim *Applications = Components + Scripts + Glue*. We distinguish between white-box and black-box forms of reuse and we argue that while inheritance is a key concept of object-oriented languages it hinders reuse. We claim that a shift towards component-oriented programming with a focus towards scripting of components is needed. Nevertheless, we would like to reuse the many components written in object-oriented languages. We would like to have the best of both worlds: the power of inheritance combined with the ease of scripting. We proceed along two paths: in Chapter 3 we introduce Piccola, a small, pure, general purpose composition language. We show how we can bootstrap from a minimal set of abstractions and validate our claim that the primitive abstractions defined by Piccola are sufficiently expressive to model higher level abstractions. Chapter 4 discusses the infrastructure needed to support inheritance in JPiccola, a Java based implementation of Piccola. We illustrate how we can generate classes on the fly and load them into the Java virtual machine using a custom class loader. In Chapter 5 we introduce our approach for white-box reuse in JPiccola and discuss implementation issues. This approach we use in Chapter 6 to script various kinds of Java components and present services and styles built on top of them. In Chapter 7 we conclude the thesis and discuss lessons learned.

Chapter 2

Component-based Software Development

Modern applications must be flexible and extendible to adapt to changing requirements. Object-oriented methodology has come a long way towards achieving these goals, helping to develop expressive models that reflect the objects of the problem domain. However, it only partially succeeded in providing software architectures based on reusable and extendible components [PS96].

A new approach to software development is needed [NM95, ND95]: the requirements of open systems can only be adequately addressed by adopting a *component-oriented* as opposed to a purely object-oriented software development approach with a shift away from programming towards software composition.

This chapter is structured as follows: in Section 2.1 we introduce a conceptual framework for component based software development which is based on the concepts of *components*, *scripts* and *glue code*. In Section 2.2 we take a look at possible forms of reuse. We distinguish between black-box and white-box adaptation techniques of reuse. We take a look at inheritance as a prominent form of white-box reuse in object-oriented languages and we show that although inheritance is a powerful concept, it is not well suited for software composition. We thus argue in Section 2.3 that we should combine the power of inheritance with the ease of scripting and propose a migration strategy from class inheritance to component composition.

2.1 A Framework for Component-based Software Development

We propose a framework built on the following terminology [Sch99, SN99, AN01]: a *component framework* is a collection of software components and architectural styles that

determines the interface components may have and the rules governing their composition. A *software component* itself is a static abstraction with plugs and can be seen as a kind of black-box entity that hides its implementation details. A connector connects required ports of a set of components to provided ports of components. A *software architecture* describes a software system as a configuration of components and connectors. An *architectural style* is an abstraction over a family of software architectures. It defines a vocabulary of component and connector types and a set of rules defining how components and connectors can be combined.

In the next section, we introduce a conceptual framework for software composition based on this terminology.

2.1.1 Applications = Components + Scripts + a Drop of Glue

Complex software systems are required to be open, flexible compositions of heterogeneous and distributed software components rather than monolithic heaps of code. This requirement places a strain on old-fashioned software technology and methods that are based on the maxim

> Applications = Algorithms + Data.

For well-defined and delimited problems this equation still has its relevance. Imperative programming languages are based on this view of software composition and focus at top-down decomposition of a problem space.

Object-oriented programming languages go one step further by encapsulating state and behavior into objects. Applications are made up of objects that communicate by message sending, leading to the maxim

> Applications = Objects + Messages.

While object-oriented languages succeed in implementing components, they have a rather limited support for expressing composition abstractions, e.g., adding coordination abstractions to distributed systems. For component-based software engineering, we need an approach that clearly separates between computational and compositional entities. Therefore, we propose a component-oriented style based on the maxim

> Applications = Components + Scripts.

Components must conform to architectural styles that determine the plugs each component may have, i.e., exported and imported services, the connectors that may be used to compose them and the rules governing their composition. Scripts define specific connections

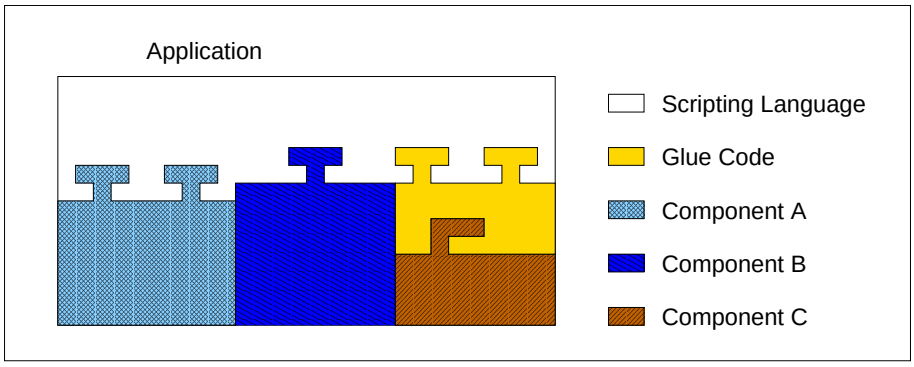

Figure 2.1: A framework for component-based software development

of the components. In this paradigm, building an application means choosing the right components and plugging them together following the rules that govern their composition.

Unfortunately, in reality *as-is* reuse of components is very unlikely to occur. In the majority of cases, components have to be adapted in some way or another to match an architectural style. We use glue code to overcome compositional mismatch. Glue code makes mismatched components composable by adapting their interface in a way that they conform to a given architectural style. We thus extend our equation as follows:

$$\text{Applications} = \text{Components} + \text{Scripts} + \text{Glue}.$$

Figure 2.1 illustrates our framework for component-based software development: in this example, an application consists of three components A, B and C which are plugged together using a scripting language. While the first two components adhere to the given architectural style (depicted by the plugs in form of a 'T'), the last component does not. We overcome this compositional mismatch using glue code.

In our framework components, scripts and glue code are key concepts. In the next sections we introduce them one by one in more detail.

2.1.2 Components

A software component is a *static abstraction with plugs*[ND95]. By *static*, we mean that a software component is a long-lived entity that can be stored in a software base, independently of the applications in which it has been used. By *abstraction*, we mean that a component puts a more or less opaque boundary around the pieces of software it encapsulates. *With plugs* we express that there are well-defined ways to interact and communicate with the component.

Components are black-box entities that provide services to other components which, nonetheless, may require certain services to work. They provide their functionality through a set

of interfaces and have well-defined requirements. Components can be of arbitrary size, they can be very fine grained like a button of a GUI framework or they can be very coarse grained like a PDF-viewer embedded into a web browser.

Fine grained components tend to be homogeneous, since these components are typically composed to form an application built within a single programming environment. Components of a GUI framework are good examples of this type of components. In contrast, using the pipes and filter architecture provided by a UNIX shell, heterogeneous components written in arbitrary languages can be scripted as long as they obey to this compositional style.

Szyperksi [Szy98] claims that software components are binary units of independent deployment[1]. In fact, most of the existing component models are based on binary components, e.g., COM, Corba or Enterprise JavaBeans. However, this property is not a crucial feature of a software component: the Standard Template Library of C++ is a source code based component providing templates to build powerful data structures.

Components are mainly written in system programming languages that are strongly typed. Strongly typed languages help manage the complexity of a software component by providing type annotated APIs. Compilers use type information to detect type errors at compile time and exploit this information to generate optimized code, improving the performance of the components.

2.1.3 Scripts

We define *scripting as a high-level binding technology for component-based systems* [Sch99]. This definition accurately captures the main purpose of scripting, namely, to build applications by composing a set of existing components at a high-level of abstraction.

Based on this definition, we define the term *scripting language*: *A scripting language is a high-level language to create, customize, and assemble components into a predefined software architecture* [Sch99]. To adhere to this definition a scripting language must provide means to work with encapsulated pieces of software and define interfaces to it. These components may be written in a foreign (system programming) language. A scripting language must thus have powerful constructs to interoperate with this language. In other words, it must provide external extensibility.

We can divide scripting languages into special-purpose and general-purpose scripting languages. An example of the former category is JavaScript, a language embedded into web browsers and used for dynamic manipulation of web pages. An example of the latter cate-

[1]Szyperski's argument for binary components is not about protecting intellectual property or about information hiding, his main concern is to increase the ease of reusability by minimizing dependencies and knowledge required to use components [Szy02]: he wants them to be units of deployment – units that can be dropped into a system and put to work. Further, he argues that source code based components can not guarantee this as we have to provide additional information to get these components to work.

gory is Python, an interpreted, object-oriented, high-level programming language used in various application domains.

Scripting languages represent a completely different style of programming than system programming languages [Ous98]. Scripting languages are often dynamically and weakly typed and offer support for introspection. They are mainly interpreted and provide automatic memory management. High-level programming, like rapid prototyping, is favored over execution speed. Therefore, scripting languages offer built-in data abstractions such as powerful data and control structures.

Combining the strength of the two language types gives use the best of both worlds: we use system programming languages to create exciting components which we glue together using scripting languages.

2.1.4 Glue Code

In an ideal world, there are components available for any task applications have to perform, and these components can be simply plugged together. In practice, such components often do not exist [Bos99]. We therefore need glue code to adapt components that do not fit the compositional requirements of a framework or system.

We define *glue code* as the part of an application that *overcomes compositional mismatch*. Overcoming compositional mismatch means that we extend a component so that it becomes plug-compatible to an architectural style. Compositional mismatch can occur at any level: at the architecture platform level, where components are not designed for the platform they are supposed to run on. At the cross-platform level, where components are running on different components platform. At the interaction level, where components use different protocols. At the architectural level, where components make different assumptions about the architecture on which they are supposed to run, leading to *architectural mismatch* [GAO95]. Last but not least, versioning conflicts can lead to compositional mismatch as well.

A number of techniques for overcoming compositional mismatch have been developed for adapting components. These adaptation techniques can be categorized either as black-box or white-box techniques: white-box techniques focus on adapting a mismatched component by either changing or overriding its internal specification, whereas black-box techniques only adapt its interface. In the next section, we examine the advantages and disadvantages of these two approaches more closely.

2.2 Forms of Reuse

2.2.1 White-Box versus Black-Box Adaptation Techniques

White-box adaptation techniques require us to understand the internals of the reused component. To extend or reuse a component, we have to be familiar with the implementation details of the component. Two often used white-box adaptation techniques are:

Copy & Paste. Copying the code of the part of the component that provides the functionality needed and pasting it where appropriate is a very simple form of reuse. Although it gets the job done, it has some major drawbacks: it leads to maintenance problems as the same code is spread among several places and, to make matters worse, the code often gets slightly adapted which makes code reuse virtually impossible.

Inheritance. Inheritance makes the state and behavior of the reused component available to the reusing component. Depending on the language model, all internal aspects (as in Smalltalk or Python) or only part of the aspects (as in C++ of Java) become available to the reusing component. The advantage of inheritance for reuse is that the reused code stays in one place. Its main disadvantage is that its user must have a detailed understanding of the internals of the base classes and of the subtle dependencies that can arise between base and extending classes. We discuss reuse by inheritance in more detail in Section 2.2.3.

Black-box adaptation techniques ideally only require us to understand the interface of a component, the internals of the component are of no concern to us. The component can be reused *as-is*. Some of the adaptation techniques used are:

Wrapping techniques. A common technique to overcome compositional mismatch uses wrappers that pack the original component into a new one with a suitable interface. Wrappers can have the form of *adaptors* which bridge incompatible interfaces, or *transformers* which modify mismatching interaction protocols. Other glue abstractions are bridges, proxies, and mediators.

Reflective approach. This approach uses the reflective capabilities of a language to intercept service invocations and to manipulate them according to the requirements of the receiver of the corresponding invocation. In particular, if service invocation is based on message sending – as in Smalltalk – messages can be transformed, delayed, or even delegated.

Standardization. Whereas wrappers and other glue techniques mainly focus on overcoming compositional mismatch, standardization tries to avoid mismatch by restricting the kind of components that can be used in a system. As an example, *software buses* define a standardized communication protocol for exchanging data, take care of correct message handling and perform necessary data conversions.

We can use the black-box and white-box adaptation techniques described above to reuse components. In general, black-box reuse of components should be favored over white-box adaptation techniques. The deficits of reuse of components by copy and paste are vast: it lacks code reuse and because of duplicated code complicates maintenance. We do not treat this adaptation technique any further.

The reason why reuse by inheritance is problematic is not so obvious. After all, inheritance is crucial to object-oriented programming languages. In the next section we briefly highlight the different roles of inheritance, examine inheritance as a means for code reuse and discuss the problems it possesses as a white-box adaptation technique.

2.2.2 Roles of Inheritance

Inheritance plays various roles in object-oriented programming. At different levels it serves different purposes [LP91, Tai96]:

Subclassing. At the implementation level, inheritance is used as a mechanism for sharing code and representation and helps in avoiding duplicated code.

Subtyping. At the design level, inheritance defines a substitutability relationship: an instance of a subtype can stand in for an instance of its supertype.

***Is-a* Relationship.** At the conceptual level, inheritance represents a conceptual specialization relationship: it describes one kind of object as a special kind of another.

The different roles of inheritance are unrelated, they often even conflict. For example, subclassing and subtyping are orthogonal roles. A well-known example highlighting this orthogonality can be found in the Smalltalk class hierarchy: here, the class `Collection` serves as a superclass of the class `Set` which itself is inherited by the class `Dictionary`. At the implementation level this inheritance hierarchy makes sense as these classes are used to store values and code can be shared between them. However, viewed from the conceptual level the inheritance hierarchy is wrong: a dictionary is simply not a set.

Analyzing the different roles of inheritance is a fascinating yet exhausting enterprise. Meyer, for example, identifies and classifies as many as 17 different forms of inheritance [Mey97]. A full treatment of inheritance is not the aim of this thesis. Nevertheless, we would like to discuss two distinguishing criteria in the rest of this section: first, single versus multiple inheritance, and second, implementation inheritance versus interface/subtype inheritance. This discussion should help us to better understand the Java programming language.

Multiple Inheritance versus Single Inheritance

We can distinguish object-oriented programming languages by their support for multiple inheritance. Multiple inheritance is the ability of a class to inherit from more than one superclass. In contrast, if a class can have at most one superclass then the language supports single inheritance only.

Multiple inheritance is a powerful construct, so powerful that it can lead to very complex inheritance hierarchies which can be hard to manage and understand. But more fundamentally, the concept of multiple inheritance also poses theoretical problems. The following example is taken from Sakkinen [Sak92]. Let us define five classes:

```
class A  { ... } ;
class B: public virtual A { ... };
class C: public B { ... };
class D: public B { ... };
class E: public B, public C, public D { ... };
```

Every E object now has three distinct B subobjects and they share a common A subobject. This is badly inconsistent with the notion of B being a subclass of A.

Multiple inheritance complicates the programming language supporting it [Sny86]. Name clashes and ambiguities introduced in the object model must be resolved by the language in order for multiple inheritance to work. Many object-oriented programming languages, e.g., Smalltalk or Java, therefore do not support it.

Implementation Inheritance versus Interface/Subtype Inheritance

Subtype inheritance – also known as interface inheritance – is the most common form of inheritance. Every subclass is considered to be a subtype of its superclass. A language supporting subtype inheritance considers an object to conform to the type of its class or any of its superclasses.

Implementation inheritance is the ability of a class to inherit part or all of its implementation from another class. Most languages do not support pure implementation inheritance, instead they support implementation and subtype inheritance: a class that inherits from another class is always considered to be a subtype of its superclass.

C++ supports single pure implementation and single subtype inheritance as well as both forms of multiple inheritance. Implementation inheritance is enforced by *private inheritance*, in which methods of a base class are made private in a derived class.

Similarly to C++, Python supports multiple implementation and interface inheritance. However, Python offers no support for access control: all attributes and methods are publically accessible. In Python privateness is expressed by convention: any method beginning with an underscore is understood to be private.

Java supports a mix of single and multiple inheritance. Its designers decided that multiple implementation inheritance creates too many problems and stuck to a single inheritance model [GM96]. Instead they introduced *interfaces* in addition to single class inheritance. An interface is a specification of methods that an object declares it implements. An interface does not include instance variables or implementation code, only declarations of constants and methods. The concept of an interface is borrowed from the Objective-C concept of a *protocol* [Com02]. While a class can inherit from one superclass only, it can implement as many interfaces as it chooses to. To summarize, Java provides two separate inheritance mechanisms: Single implementation inheritance combined with multiple inheritance of interfaces.

2.2.3 Reuse by Inheritance Considered Harmful

In the previous section we have sketched the different roles of inheritance and discussed two distinguishing criteria. In this section we concentrate on one role, i.e., the subclassing role of inheritance as a means of code reuse.

The subclassing role of inheritance denotes the ability of a class or an object to be defined as an extension or specialization of another class or object. The benefit of this role of inheritance is to promote code reuse. Code shared by several classes can be placed in their common superclass, and new classes can profit from this common code base and functionality.

Avoiding multiple inheritance helps in solving some of the problems of inheritance. However, from the point of view of component reuse, inheritance *per se* hinders reuse: inheritance contradicts the idea of encapsulating an object's implementation [Sny86, WS96]. The aim of encapsulation is to separate the external aspects of an object which are accessible to other objects, from the internal implementation details of the object which are hidden from other objects. The aim of subclassing is to define a class or an object as an extension or a specialization of another class or object. These two aims conflict: subclassing can break encapsulation by accessing internal implementation details of an object.

A well-known example illustrating this conflict is the *Fragile Base Class Problem*: classes in the foundation of an inheritance hierarchy of an object-oriented system can be fragile. The slightest attempt to modify these foundation classes may damage the whole system.

The fragile base class problem comes in two forms [Szy98]: the *syntactic* and the *semantic* base class problem: the syntactic fragile base class problem concerns maintaining binary compatibility of compiled classes with new binary releases of superclasses. This problem is solvable: if we guarantee not to break binary compatibility, we can circumvent the syntactic base class problem. The semantic base class problem in contrast is more difficult to address.

The following example is an adapted version of a well-known example from Mikhajlov and Sekerinski [MS97]. Let us define a class `Bag` with a member variable `_list` and two

methods: `add(self, x)` for adding elements and `addAll(self, list)` for adding lists. Important to note is that `addAll(self, list)` depends on `add(self, x)`: Elements are added one by one by iterating over the list.

The class `CountingBag` inherits from `Bag`, adds a member variable `_counter` functioning as a counter and overwrites `add(self, x)` that increments the counter with every element added. The method `addAll(self, list)` uses `add(self, x)` and the counter gets correctly incremented even when lists are added.

```
class Bag:
   def __init__(self):
      self._list = []
   def add(self, x):
      self._list.append(x)
   def addAll(self, list):
      for x in list:
         self.add(x)

class CountingBag(Bag):
   def __init__(self):
      Bag.__init__(self)
      self._counter = 0
   def add(self, x):
      self._counter = self._counter + 1
      Bag.add(self, x)
```

If we want to improve the performance of the `Bag` class, we can rewrite `addAll(self, list)` to concatenate the two lists directly, no longer depending on `add(self, x)`. The new `addAll(self, list)` method might now look like this:

```
  def addAll(self, list):
      self._list += list
```

This change breaks the semantics of the `addAll(self, list)` method inherited by the class `CountingBag`: the variable `_counter` is no longer incremented when `addAll(self, list)` is called. Neither recompiling the whole class hierarchy nor extensively testing the `Bag` class reveals the problem. The semantic of the `Bag` class has not changed. The problem only becomes apparent when both classes are tested.

This example clearly demonstrates the subtle dependencies that can arise in inheritance hierarchies. Harmless changes to base classes can break extending classes. This problem has major drawbacks.

Let us assume that `Bag` is part of a framework of a vendor A, and `CountingBag` is an extension of vendor B. As vendor A wants to improve the performance of his framework, he modifies the class according to the changes discussed above. When he tests his framework

no problems arise, so he distributes his new framework. Suddenly, the class `CountingBag` breaks without any change to its code. In order for vendor B to fix the problem he has to know how the internals of class `Bag` have changed, which can be difficult if vendor A releases his framework in binary form only. Once vendor B has fixed his class he must release a new version, which can be a major undertaking.

This situation contradicts the requirements of an open and distributed system. In an open system it must be possible to replace components easily. Changes to a sub system should not affect the functioning of the whole system. Instead, it must be possible to plug out the old component and plug in the new one without having to worry about subtle dependencies between the system's components and without having to know anything about the internals of the components.

2.3 Power of Inheritance and Ease of Scripting

In the framework presented in Section 2.1, we write applications by composing components through scripts. Scripting is a black-box adaptation technique for component reuse. Components can have various forms, objects are just one possible form. If we have a piece of software that is not scriptable using our architectural style, we can use glue code to overcome compositional mismatch.

In Section 2.2 we looked at different forms of reuse and sketched the different roles inheritance plays in object-oriented languages. One role is subclassing which has a major drawback: it contradicts the idea of encapsulating an object's internals and introduces subtle dependencies in an inheritance hierarchy that hinder effective reuse.

We would like to have the power of inheritance combined with the of ease of scripting. Using glue code we would like to reuse objects by inheritance and turn them into a component with an interface adhering to an architectural style and well-defined requirements. More specifically, we would like to proceed as follows:

1. *Use inheritance to gain access to the components.* Reuse of components written in an object-oriented language is often only possible by inheritance, so we *have* to have a way to deal with it. Our aim is to wrap the components with a well-defined interface adhering to a compositional style to make them accessible from JPiccola.

2. *Script the wrapped components.* Once we have wrapped the components in JPiccola, we compose them by scripting. Now we can switch from a class inheritance based form of reuse to component composition.

The main aim of our approach is thus to minimize the use of inheritance to avoid the problems associated with it. We are proposing a *migration strategy* from class inheritance – a white-box form of reuse – to component composition as a black-box form of reuse. Or to put it in more flowery terms: we are using inheritance as a ladder to overcome an obstacle; we need it to get to the top, but once we are there, we leave it behind as we do not need it anymore.

2.3.1 Related Work

We intend to inherit from a Java class in JPiccola. This is a simple form of *cross language inheritance*. In this section we briefly address related work.

Various Java-based scripting languages support cross language inheritance. We discuss them in Section 4.1. Jython [PR02], a Java-based implementation of the Python scripting language, makes great efforts to seamlessly support inheritance between Java and Python. We analyze its cross language inheritance mechanism in detail in Section 4.1.2.

The first major attempt to support general cross language inheritance has been provided by IBM's *System Object Model (SOM)* [Lau94]. It provides a rich, extensible, object model with complete runtime support for its semantics. SOM embodies object oriented programming features such as implementation inheritance, encapsulation and polymorphism, as well as, advanced capabilities including metaclasses, user intercept and control of method dispatch and dynamic class construction. SOM provides these capabilities in a language independent way, e.g., it provides full subclassing across languages. In SOM, it is possible to implement an object using one language, then subclass the object using another language and use that class to build an application in yet a third language.

More recently, Microsoft has released its *.NET* framework. The functionality of .NET is based on the capabilities of the *Common Language Runtime (CLR)* [MG02]. The CLR is explicitly designed to support a wide range of programming languages, i.e., imperative, object-oriented, scripting and declarative [MPY01, SPM02] languages[2].

The CLR consists of a *Common Type System (CTS)* that provides a rich type system [GS01] common to all languages, an *instruction set* designed to support different classes of programming languages [MG02][3] and *metadata* that defines a common mechanism for exchanging type information.

The CLR provides the necessary foundation for language interoperability by specifying and enforcing a common type system and providing metadata. Because all languages targeting the CLR follow the rules of the CTS for defining and using types, their usage is consistent across languages. Metadata enables language interoperability by defining a uniform mechanism for storing and accessing type information. Compilers store type

[2] see: research.microsoft.com/project7.net/project_7.htm

[3] For example, it includes a *tail call* instruction that is crucial for the efficient implementation of functional programming languages.

information as metadata, and the CLR uses this information to provide services during execution; the CLR can thus manage the execution of multi-language applications because all type information is stored and retrieved in the same way, regardless of the language the code was written in.

Key to the cross-language inheritance capabilities of SOM as well as .NET is a common type system. By expressing the types of the various languages in a common type system, it provides the bridging mechanism for inheritance between the different type models of the languages.

We do not attempt to model such a type system in Piccola. In fact, we try to minimize the use of inheritance and use it as a means of migration only.

2.3.2 Next Steps

To pursue the migration strategy proposed in this section, we proceed along two paths: we introduce a scripting language and we discuss what is needed to integrate inheritance into this language.

In Chapter 3 we introduce Piccola, a small, pure and general-purpose composition and scripting language. It has a small syntax and a minimal set of features needed for specifying different styles of software composition. We introduce the key concepts behind Piccola, agents, forms and channels, and illustrate how we can define higher level abstractions based on these primitives.

JPiccola is an implementation of Piccola on top of the Java platform. While we have explored black-box reuse extensively using Piccola it is not possible to reuse Java components through subclassing at runtime. In Chapter 4 we investigate the infrastructure needed to support dynamic subclassing in JPiccola.

Chapter 3

Piccola

In the previous chapter we introduced a framework for component-based software development based on the maxim *Applications = Components + Scripts + Glue*. In this chapter we introduce Piccola, a small, pure, general purpose composition language supporting this view of software composition. Based on *forms*, *agents* and *channels*, Piccola provides a simple, but expressive formal model that enables us to define composition abstractions.

This chapter is structured as follows: in Section 3.1 we give a concise overview of Piccola's syntax and its semantics, which has been extensively covered in [Sch01, Ach02], and sketch its implementation. In more detail we treat Piccola's bridging mechanism which is responsible for interfacing with external components provided by the underlying host language.

Piccola gains most of its expressive power from the underlying host language. In Section 3.2 we demonstrate how we can model primitive data types in Piccola based on the data types provided by the host language. Additionally, we illustrate how we can model control flow structures in Piccola and discuss their differences to constructs found in common programming languages.

3.1 The Piccola Language

Piccola is a *small*, *pure* and *general purpose* scripting language based on the π-calculus. By small we mean that Piccola has only a small syntax and provides a small set of primitives for specifying different styles of software composition. Piccola is a pure composition language, all computation is performed by external components and it only provides a small set of primitives needed to express the necessary composition abstractions. Contrary to special-purpose composition languages which only support the scripting of one specific component model, Piccola is a general-purpose composition language as it supports composition of components corresponding to different compositional styles.

Piccola is based on *forms*, *agents* and *channels*. Forms are extensible records unified with services. Agents are autonomous entities that communicate by sending information along channels. This minimal set of abstraction is sufficiently expressive to define high-level composition abstractions.

Conceptually, Piccola is built on top of the π-calculus, a calculus for communicating and mobile systems devised by Milner [Mil99]. Piccola expressions are encoded into the π-calculus, the soundness of this encoding is proven in [Ach02]. Encoding of Piccola into the π-calculus gives us a higher level of expressiveness compared to the π-calculus while it retains a strictly defined formal model which is small enough to be mathematically tractable.

3.1.1 Syntax of Piccola

The abstract syntax of Piccola is given in Table 3.1. The most important class of terms are *form expressions*. These expressions evaluate to *forms* which are the unifying concept in Piccola.

The keyword `root` denotes the static namespace where we look up *identifiers*. Constant *literals* are numbers and strings. A backslash (\), optional parameters and a colon (`:`) followed by a form define an *anonymous service*. *Projection* of `A` on a label `B` we express using `A.B`.

An *application* `A B` denotes the result of applying a service A to a form B. Additionally, Piccola supports *infix* as well as *prefix* applications which enable us to define certain compositional styles, e.g., pipes and filter in a natural way.

A *collection* is encoded by matching tokens $op^{\{}$ and $op^{\}}$. Valid tokens are sequences of `{` or `[` and `}` or `]`, respectively, where `{ ... }` is used to denote a list and `[ ... ]` is used to denote a set.

Parentheses are used to increase the precedence of a form expression[1]. We use *sandbox* to specify a static namespace in which we evaluate the subsequent form. Alternatively, we can use *quote* (') to extend the static namespace instead of replacing it. In fact, the quote expression `'E, F` is just syntactic sugar for the sandbox `root = (root, E),F`. While a anonymous abstraction is specified by a *anonymous service*, we define a *service binding* by named abstractions. Finally, to define a recursive form we use the keyword `def` which is implemented using fix-points.

Like Python, Piccola uses indentation and newlines instead of parentheses and commas to group form expressions. This design decision contributes to Piccola's small syntax: there are only two keywords `def` and `root` and built-in data types or flow control structures are completely missing. In Section 3.2 we demonstrate how we can define higher level constructs expressed in this small syntax.

[1] The complete precedence rules are given in [Ach02].

Form ::=	
root	*current namespace*
identifier	*label*
literal	*constant literal*
\ [*Param*] : *Form*	*anonymous service*
Form . *identifier*	*projection*
Form Form	*application*
Form op Form	*infix application*
op Form	*prefix application*
Form , *Form*	*extension*
$op^{\{}$ [*FormList*] $op^{\}}$	*collection*
([*Form*])	*parentheses*
root = *Form* [, *Form*]	*sandbox*
[**def**] *Label* [*Param*] : *Form* [, *Form*]	*service binding*
[**def**] *Label* = *Form* [, *Form*]	*binding*
' *Form* [, *Form*]	*quote*
FormList ::=	
[*FormList* ,] *Form*	*collection composition*
Param ::=	
identifier[*Param*]	
([*identifier*]) [*Param*]	
Label ::=	
identifier	*simple label*
Label . *identifier*	*nested label*

Table 3.1: Piccola Language Syntax

3.1.2 Semantics of Piccola

Forms

Forms are the only first class values of Piccola, every first-class entity is represented as a form. Forms are to Piccola what objects are to object-oriented languages. A form supports five basic operations: *extension*, *projection*, *application*, *restriction*, and *inspection*.

Extension. A form is an immutable and unordered set of bindings of labels to values. Forms can be nested and extended with other forms. The empty form is denoted by `()` and contains no bindings.

```
aCircle =
    center =
        x = 5
        y = 5
    radius = 3
    diameter(): 2 * radius
```

The binding `diameter` defines a named abstraction we call *service*. A service is a form with a hidden label that gives access to an agent that represents it. The binding above is just syntactic sugar for an anonymous lambda abstraction bound to a label:

```
diameter = \(): 2 * radius
```

We can use extension to create default bindings. If the form `Point` does not contain a label `x`, we can provide a default one, otherwise the value of `x` provided by `Point` is used.

```
xCoord(P):
    println (x=0, P).x  # default: x=0

xCoord(x=9)
```

Projection. We extract bindings of a form by projecting on a certain label. If there is no such label, a runtime error occurs.

```
aCircle.radius      # returns 3
aCircle.center      # returns (x=5, y=5)
aCircle.diameter    # returns PiccolaService
```

Application. A service is represented as a form. The application of a service, `F G` invokes the service represented by the form `F` with the argument form `G` and yields the resulting form.

```
aCircle.diameter()  # returns 6, the empty form is passed as an argument
xCoord(aCircle.x)   # prints 5
xCoord aCircle.x    # ditto
```

Restriction. Using restriction we can remove a binding from a form. We can use restriction combined with inspection to iterate over all labels of a form.

```
println aCircle

# prints (center = (x = 5, y = 5), diameter = PiccolaService, radius = 3)

radiusLabel = label(radius=())
aCircle2 = radiusLabel.restrict(aCircle)
println aCircle2

# prints (center = (x = 5, y = 5), diameter = PiccolaService)
```

Inspection. Using inspection we can investigate the structure of a form. The primitive service `inspect()` is Curried. As first argument it takes the form to be inspected, as second argument it takes a form containing three services. Depending on the structure of the form, either the `isEmpty()`, the `isService()` or the `isLabel()` service is invoked.

```
Cases =
    isEmpty():    println "Form is empty"
    isService():  println "Form is a service"
    isLabel():    println "Form is a label"

inspect () Cases         # prints: Form is empty
inspect (\: ()) Cases    # prints: Form is a service
inspect (a = 98) Cases   # prints: Form is a label
```

Agents and Channels

The semantics of Piccola are given in terms of communicating agents. There are two predefined abstractions necessary to control these agents: one to asynchronously evaluate a `do()` service by a new agent and one to synchronize running agents. The `run()` primitive evaluates the `do()` service of a form by starting a new agent. The term `newChannel()` creates a new channel. Channels provide atomic `send()` and `receive()` services to communicate forms. The sender cannot detect when and whether the form sent is received by a communicating agent. An agent receiving a form from a channel blocks unless someone sends a form to it. If one or more forms are sent, then an arbitrary one of them is received. There is no ordering on the forms communicated along a channel.

A feeble attempt to model a log utility is given below: the service `log` starts an agent that sends a message to a channel and `printLog` blocks on this channel, waiting for a message to arrive.

```
ch = newChannel()
log msg channel:
    run(do:  channel.send msg)

printLog channel:
    run(do:  println channel.receive())

log "logged on" ch
log "logged off" ch
printLog ch          # prints either 'logged on' or 'logged off'
```

As the order of the messages sent is not preserved, this log utility is feeble indeed.

Scopes

Piccola features two types of namespaces: the static namespace accessible by the **root** keyword and the namespace of the current form. Defining a binding extends the `root` as well as the current namespace. Often however, we want a binding to be local to the current scope only. We achieve this effect by *quoting* (') a binding. If we are only interested in the side effect of defining a binding, we can use *double quoting* which prevents the current and the root namespace from being extended.

expression	current namespaces	**root** namespace
`a =`	ϵ	ϵ
`x = 42` # *binding*	$x \mapsto 42$	$x \mapsto 42$
`'y = x` # *quoted binding*	$x \mapsto 42$	$x \mapsto 42, y \mapsto 42$
`''z = y` # *double quoted binding*	$x \mapsto 42$	$x \mapsto 42, y \mapsto 42$

Piccola is statically scoped, but offers *dynamic scoping* on demand. Without dynamic scoping some kinds of coordination and control abstractions, e.g., exception handlers, are next to impossible to define. Therefore, whenever a service is called in Piccola, the form `dynamic` – containing the dynamic scope – is passed implicitly together with the actual parameters. If the client extends this dynamic context with additional services, these are then available to the called abstraction.

Labels

We can use restriction to remove bindings from a form by calling `restrict()` on a *first-class label* which we obtain by using the built-in service `label()`. First-class labels enable us to

extend forms with labels using the `bind()` service. We restrict labels invoking `restrict()` and we project on forms using `project()`. Furthermore, we can test whether a form contains a certain label using the `exists()` service.

```
aLabel = label (color = ())      # Returns first class label color
form = aLabel.bind("blue")       # Binds the label 'color' to 'blue'
println form                     # Prints: (color = blue)
binding = aLabel.project(form)   # Project on the label 'color' of the form
println binding                  # Prints: blue
println aLabel.exists(form)      # Prints: true
form = aLabel.restrict(form)     # Removes the label color from the form
println form                     # Prints: ()
```

Using these services we can built up arbitrary forms and reason about forms at run-time.

3.1.3 Interfacing to External Components

Piccola is a *pure* composition language. There are no predefined primitives for any computation, every computation is performed by external components provided by the host language[2]. In order to integrate an external component seamlessly we need to adapt it in Piccola to a given style. We extend its interface with certain bindings. But when we send this component back to another external service, we can not pass the wrapped component but we must pass the original, i.e., the unwrapped component. In short, we need a *bridging strategy* [Sch01] between Piccola and its external components.

We can access an external component as a form. We call this form the *peer form*. We denote the peer form for any external component c by $p(c)$. All provided services of the external component are services of the peer form. When we invoke a service of a peer form $p(c)$ in Piccola, the service of c is invoked and the argument (which is a form) is converted to an external component again. Therefore we need a function e to convert every form F into an external component $e(F)$. If the form F is a peer form $p(c)$ then converting this form back to a component must provide the original external component, written $e(p(c)) = c$.

We define two functions *up* and *down* [Meu98] which describe the effect of upping an external component into a form, and downing a form which returns the external component:

$$\begin{aligned} Up(c) &:= p(c) \cdot peer{\mapsto}p(c) \\ Down(F) &:= e((peer{\mapsto}F \cdot F); peer) \end{aligned}$$

If F does not contain the *peer* label then $Down(Up(c) \cdot F) = c$ holds. This property guarantees that we can modify any upped component and invoke an external service that receives the original component as an argument.

[2]It is important to note that from a theoretical point of view, Piccola is computational complete. *In theory*, every computation performed by the external components can be performed by Piccola as well.

We use the term *external form* to denote a form that represents an external object within Piccola, and we use the term *plain form* for all other Piccola forms. We require that an external form has a nested structure consisting of two parts. The top level part represents the Piccola interface. Additionally, this form contains a label `peer` that is bound to a sub form representing the identity of the external object, which is called *peer form* or just *peer*. Any form corresponding to this structure is considered an external form.

Host objects are instantiated via *peer classes*. Peer classes give access to all static methods and fields of a class and contain a factory service to create *peer objects*. In JPiccola, Java objects are the external components performing the computation. Peer classes are created by the predefined service `Host.class(String)` with the name of the class as its argument. Peer objects are created by calling `new()` with optional arguments on peer classes.

Invoking a service `serve` on a Java peer object o with the argument form F triggers the following three steps to find the appropriate method to call on a Java object: we first define a form-tuple from the argument, then we create a tuple of Java objects, and finally we associate a type-tuple with the argument to resolve overloaded methods. More precisely:

1. If the form F has bindings `val1`, `val2` up to `val`n, we create a tuple containing these bindings. If the form has no binding `val1` we create the 1-tuple (F). If the form is the empty form, we create the empty tuple ().
2. We convert the form tuple into a tuple of objects $(o_1, \ldots, o_n)$ by applying *Down* on each component of the tuple.
3. We construct a tuple of types in order to resolve overloaded methods. The type t_i of each Java object o_i of the tuple is its class. If however the form F has a binding $type_i$, then the type t_i is $F.type_i$.

Next we search for the Java method with the signature `serve`$(t_1, t_2, \ldots, t_n)$, pass the arguments and invoke it. If we do not find such a method, we raise an exception.

The fact that there are primitive types in Java complicates the lookup procedure. In JPiccola, we use Java objects only. Thus, whenever a primitive Java type is upped we convert it to the corresponding Java object, e.g., `int` is transformed into a `java.lang.Integer`. This conversion bears the problem that when we down such an object we no longer know whether we have to convert the object back to a primitive type or not.

We can address this problem in two ways: the first solution marks the fact that the object is actually a converted primitive Java data type in the `peer` form. We can use this information when we down a form to transform it back to the appropriate Java type. The drawback of this solution is that it introduces an additional binding in every `peer` form. The second solution focuses on extending the procedure of finding the correct Java method to call: if a signature does not match, we can try to convert the object's type to primitive data types, hoping to find a matching signature. This approach leads to a complicated method lookup. Note that in a pure object oriented language such conversion problems would not occur.

3.1.4 Implementation

Applications	*Components + Scripts*
Architectural Styles	*GUI composition, pipes and filters*
Core Libraries	*Data types, services and abstractions*
JPiccola VM	*Forms, Agents and Channel + Bridge*

Figure 3.1: Layered Implementation of Piccola

Piccola is implemented based on a layered approach [AN01]. The bottom layer provides a virtual machine on top of which agents asynchronously communicate forms through shared channels. The virtual machine implements the Piccola calculus and the bridge that is responsible for interfacing with external components. This layer is entirely written in the host language. Hence JPiccola is written in Java.

The next layer defines the core libraries of Piccola. We introduce primitive values, like numbers and strings, higher-order abstractions over agents, forms and channels, and basic composition abstractions, e.g., control abstractions (e.g., if-then-else, try-catch) or coordination abstractions (e.g., blackboards, futures).

At the third layer, libraries of compositional styles are defined, e.g., styles defining push-flow or pull-flow streams, GUI composition, or GUI event composition. A style may also define coordination abstractions to manage interactions between components or glue abstractions to adapt external components to a particular style or to bridge gaps between different styles.

All the services defined in the previous layers we can use at the top layer to create applications by scripting the components adhering to a compositional style.

3.2 Bootstrapping JPiccola

Piccola gains its computational power by interfacing to the underlying host language. There are no data types, no control flow structures and no composition services built into the language. The purpose of this section is to demonstrate how we can build up such constructs in JPiccola by wrapping external components and by defining higher level services on top of them.

The boot-up process of JPiccola is divided into the following two steps:

1. *Setup an initial, minimal form.* Built into the JPiccola virtual machine is an initial

```
Host =
    host = upped ch.unibe.piccola.Host
    run(): PiccolaService
    protect(): PiccolaService
    newChannel(): PiccolaService
```

Figure 3.2: Initial, minimal boot-up form

form containing a minimal number of bindings and services. These services provide the hooks to bootstrap JPiccola.

2. *Process the boot-up script.* Load and process the boot-up script (called `prelude.picl`). All data types, control structures, basic services and higher level abstractions are built up in this step.

The initial form constructed in Step 1 is given in Figure 3.2 which defines only one binding and three primitive services. The label `Host` provides access to the wrapped Java `Host` object which offers services to bridge to Java objects from JPiccola and provides access to internals of the JPiccola virtual machine, e.g., its parser and its compiler. The semantics of the three services are:

run. Invoking the service `run()` with a service binding `do()` creates a new agent executing this service as described in Section 3.1.2.

newChannel. Invoking the service `newChannel()` creates a new channel with two services `send()` and `receive()` as described in Section 3.1.2.

protect. If we pass an external form as an argument to a Java method the form is automatically downed, e.g., the raw Java object is passed to it. This is reasonable, as Java methods require Java objects as their arguments. However, there are situations when we want to pass the form containing the Java object as an argument, e.g., when we want to store forms into a Java collection. Using the service `protect()` on a form protects it from being unwrapped by adding it as a `peer` binding to a new form.

Using this initial, minimal form we can bootstrap JPiccola[3]. In the next two sections, we show how we can bootstrap data types and flow control services using this minimal form only.

[3]This bootstrap process is quite tricky as we have to be careful not to use any bindings or services before we have defined them. Debugging can be especially difficult as exception handling or even the ability to print out the content of a form are missing initially.

```
# Boolean type from Java.
BooleanOp = Host.class "ch.unibe.piccola.bridge.BooleanOp"
wrapBoolean X:
    'def this =
        peer = X.peer
        _&_ Y: BooleanOp.and(val1 = X, val2 = Y)     # AND
        _|_ Y: BooleanOp.or(val1 = X, val2 = Y)      # OR
        _&&_ Y: BooleanOp.select(                    # lazy AND
            val1 = X
            val2 = (true = Y, false: this)) ()
        _||_ Y: BooleanOp.select(                    # lazy OR
            val1 = X
            val2 = (true: this, false = Y)) ()
        _==_ Y: peer.equals(val1 = Y)              # equality
        select Cases: BooleanOp.select(            # returns either
            val1 = X                               # X.true or X.false
            val2 = Cases) ()
        not: BooleanOp.not(X)                      # negation
        !_ = not                                   # negation as a
                                                   # prefix operator
    this
```

Figure 3.3: Piccola Boolean

3.2.1 Introducing Data Types

In this section, we investigate how we can integrate data types into Piccola. As an example, we define Piccola Booleans with the services given in Figure 3.3. Note that we define a thin wrapper around the Java Boolean type only, all the services resort to the Java class `BooleanOp` to accomplish their task.

Using the `wrapBoolean` service we can wrap Java Booleans as forms with an interface completely defined in Piccola:

```
aString = Host.class("java.lang.String").new("Hello World")
aString.startsWith("H")    # returns a wrapped Java Boolean type by
                           # automatically calling wrapBoolean (see below)
```

This approach gives us great flexibility to built up our own data structures. Nevertheless, we still have to manually wrap the Java objects which is cumbersome for frequently used data structures. To ease this task, there are hooks in the JPiccola virtual machine which automatically call the wrapping services defined in the boot-up script for often used types like strings, numbers and booleans. This approach combines the freedom to define the wrappers entirely in Piccola with the ease of using these types. If we now

call `aString.startsWith("H")` the returned Java Boolean is automatically wrapped as a Piccola Boolean.

3.2.2 Introducing Flow Control Services

Piccola has no built-in flow control services. In this section we explore how we can bootstrap flow control services in Piccola by modeling control structures found in common programming languages. At a first glance the services defined look familiar, but as they are built up from more primitive operations, they often behave slightly different than expected.

A *if-then-else* Service

In the previous section we showed how to define Booleans in Piccola. Based on these Booleans we now define a *if-then-else* service.

```
if condition cases:
    'cases = (then: (), else: (), cases)
    condition.select(true = cases.then, false = cases.else)()

# test if-then-else service
if (2==2)                    # prints: true
    then: println "true"
    else: println "false"
```

At the heart of this service is the `select()` service provided by the Piccola Booleans. Depending on whether a condition is true or false, either the `then()` or the `else()` service is invoked. Additionally, we provide default `then()` and `else()` services. This enables us to invoke the `if()` service with a `cases` form containing an `else()`, respectively a `then()` service only. We use form extension to achieve this flexibility: if the `cases` form provides its own `then()` or `else()` services, they override the default ones.

Note that `if`, `then` and `else` are not keywords but services that do not take any argument. This allows us to omit the parentheses when defining them.

A *switch* Service

Many programming languages provide a *switch* construct to prevent deeply nested *if-then-else* statements. Next we look at how we can model a similarly behaving service in Piccola.

The difficulty of programming a *switch* service arises from the fact that we can not know beforehand how many cases a programmer defines. We therefore rely on the meta programming capabilities of Piccola. Invoking `Host.metaForm()` on a form returns its form label which we can inspect. The idea applied here is: if the `cases` form has a `condition`

label we project on that label and invoke its service. Otherwise we project on the `default` label which always exists as we provide a default one.

```
switch condition cases:
    'cases = (default: print "'" + condition + "' not handled", cases)
    'metaForm = Host.metaForm(cases)
    if metaForm.exists(condition)
        then: metaForm.project(condition)()
        else: metaForm.project("default")()

# Test switch service
switch ("german")            # prints: Guten Tag.
    german:  println "Guten Tag."
    french:  println "Bonjour."
    english: println "Good morning."
    default: println "Good morning."
```

Note that we use Piccola labels as labels for the cases of the `switch()` service. Therefore, our switch construct only works for strings. Trying this approach for integers does not work, as integers are not valid labels in Piccola.

A *while* Service

A *while* construct loops until its termination condition is true. Next, we try to model this behavior using a recursive function:

```
while X:
    def loop():
        if (X.condition())
            then:
                X.do()
                loop()
    loop()
```

We have to consider that forms are purely functional. The following construct never terminates as the value of `x` is statically bound.

```
x = 0                    # does not work, loops forever!
while                    # prints 000000...
    condition: (x<4)
    do:
        print x
        x = x + 1
```

We can use channels to model side effects by storing a value using `newVar()` [Ach02]: the service `newVar()` defines a form with a `set(X)` and a `get()` service. The service `set(X)` sends `X` to a channel. The service `get()` is blocked on this channel. When `X` is sent to the channel, `get()` reads from the channel, temporarily stores the value, resends it to the channel and returns the temporarily stored value.

Using `newVar()` to model side effects, the `while()` service now behaves as expected as `x` is correctly incremented.

```
x = newVar()
x.set(0)
while                    # prints: 0123
    condition: (x.get() < 4)
    do:
        print x.get()
        x.set(x.get() + 1)
```

A *for* Service

Having modeled a `while()` service we can easily build a Curried `forTo()` service based on it, again using `newVar()` to model side effects.

```
forTo start end statements:
    x = newVar()
    x.set(start)
    while
       condition: (x.get() < end)
       do:
           statements.do (x.get())
           x.set(x.get() + 1)

# test forTo-service
forTo 0 10               # prints: 0123456789
   do x: print x
```

A *for* Service – Continuation Style

Continuations are a powerful construct most prominently used in Scheme [Dyb87]. At any point during the evaluation of an expression, there exists a continuation that contains the computation left to complete. In a nutshell, a continuation represents the *rest of the computation.*

We can use a continuation based approach to model the `forTo()` service: First, we create a helping service called `loop()` that loops until the termination condition passed in is true. Next, we create a Curried service called `forTo()` that takes the range to loop over (from `start` to `end`) and a form `statement` that contains the statements we iterate over.

We start the iteration by creating a form consisting of a `counter`, the termination `condition` and the `continuation` which defines what we still have to compute. That is, we evaluate the `statements` form passed into the `forTo()` service, increase the counter by 1 and finally pass this form with the modified counter to the `loop()` service.

```
# Continuation based for service
loop x:
     if(x.condition(x))
          then: x.continuation(x)

forTo start end statements:
   loop
       counter = start
       condition x: x.counter <= end
       continuation x:
           statements.do x.counter
           loop(x, counter = x.counter+1)
```

The form extension operation offers a natural way to extend a form with new bindings, probably overriding earlier bindings. We use this property here to update the `counter` label of the `statements` form with the increased value.

Breaking out of a loop

Some programming languages allow the programmer to break out of a loop by using a *break* keyword. Breaking out of a loop can obscure control flow, but conceptually trying to model such a behavior in Piccola is challenging. In this section we discuss two possible approaches.

The first approach requires the introduction of a construct similar to `call/cc` found in Scheme. `Call-with-current-continuation` obtains its continuation and passes it to a procedure expecting one argument. The continuation itself is represented by a procedure of one argument. Each time this procedure is applied to a value, it returns the value to the continuation of the `call/cc` application. Thus when the continuation procedure is given a value, it returns the value as a result. The example below illustrates that behavior.

```
(* 7
   (call/cc
      (lambda (k)
         (* 5 (k 6 ))))  # returns 42
```

`Call/cc` is an extremely powerful construct that enables us to implement nonlocal exits, backtracking or multi-tasking [Dyb87]. However, as continuations can potentially have infinite extent their implementation is tricky. A virtual machine has to be designed from the start with such a concept in mind as a purely stack-based implementation of continuations is insufficient [CHO88].

We take another, albeit less powerful approach. We can use Piccola's agents and channels to model a breaking behavior based on a three layered approach.

On the bottom layer we define a service `OrJoin`. This service starts two agents that run the scripts passed to them.

```
OrJoin X:
    'receptor = newChannel()
    run (do: receptor.send X.left())
    run (do: receptor.send X.right())
    receptor.receive()
```

The next layer defines a service `breakOut` that uses the `OrJoin` service to spawn two agents `left` and `right` and synchronizes them by a channel. Additionally, we define a `break` service in the root context which simply calls the `break` service in the dynamic namespace.

The `breakOut` service works as follows: initially the `right` agent runs while the `left` agent is blocked on the channel. If the agent encounters a break while executing its script it sends a message to the channel and blocks. When the blocked `right` channel receives this message it unblocks and continues to run the continuation.

```
break X: dynamic.break X
breakOut X:
    'passControl = newChannel()
    OrJoin
        left:
            dynamic.break:
                passControl.send 0      # Send an arbitrary message
                newChannel().receive()
            X.do()
        right:
            X.continue passControl.receive()
```

On top of this layer, we can now define a loop that enables use to break out of it whenever the service `break` is invoked:

```
whileLoop:      # Let's define a while loop that uses break..
    counter = newVar(0)
    while
        condition: (counter.get() < 4)
        do:
            if (counter.get() != 2)
                then: counter.set(counter.get()+1)
                else: break()
            println "Counter: " + counter.get()

rest:           # and a continuation..
    println "The rest, moving on."

breakOut        # and run it.
    do:
        whileLoop()
        rest()
    continue:
        rest()
```

Note that we can use the `OrJoin` abstraction defined on the bottom layer to model an exception handling mechanism in Piccola [AN01]. We again start two agents. The first one executes the service passed to it and, if an exception occurs, sends it to a channel. The second agent blocks on this channel, waiting to handle the exception. If no exception occurs this agent terminates silently, otherwise it receives the exception and continues execution.

3.2.3 Discussion

In this section we looked at how we can model control structures using Piccola's primitives and its ability to bridge to external components, e.g., the Java Boolean types. We would like to highlight the following two points:

- *Agents, forms and channels are powerful primitives.* The `OrJoin` service presented in the last section is a powerful abstraction as it enables us to model a weak form of `Call/cc` and an exception handling mechanism. It nicely demonstrates the power of Piccola's primitives and supports our claim that they are sufficiently expressive.
- *Using the layered approach of Piccola as a design principle.* The control structures introduced in this section illustrate the layered approach we take in Piccola. Initially, we define general abstractions based on the primitives of Piccola and data types based on external components. Using these building blocks we can construct new

services which themselves serve as building blocks for new services. Nice examples demonstrating this point are the `if-then-else` service used in the `while` service or the `OrJoin` service as the underlying building block of an interruptible loop and an exception handling mechanism.

3.3 Conclusion

In these chapter we introduce Piccola, a small, pure and general purpose scripting language. Piccola is based on agents that communicate by passing forms along channels. Furthermore, we present our bridging strategy that enables us to seamlessly access external components from Piccola. Finally, we validate our claim that the primitive abstractions defined by Piccola are sufficiently powerful to model higher level abstractions.

Many compositional styles and abstractions have been developed for Piccola, e.g., object-oriented programming abstractions, coordination styles, GUI composition, pipes and filter styles [Sch99, AKN00, Ach02]. These results support our view of component-oriented software development as the process of writing applications by scripting components.

All these styles are based on composition as a black-box form of reuse. However, as we have seen in Chapter 2, inheritance is a prominent and powerful form of reuse in object-oriented language. We would like to introduce this white-box form of reuse into JPiccola. Therefore, in the next chapter we discuss how to dynamically generate classes on the Java platform and in Chapter 5 we introduce a language extension to JPiccola that enables us to support inheritance on a high level of abstraction.

Chapter 4

Dynamic Class Generation

In Section 2.2.3 we show that subclassing as a white-box form of reuse is problematic. However, in object-oriented programming languages inheritance is a key concept. The question we address in this chapter is how we can handle this form of reuse in a Java based scripting language. We would like to have the capability to dynamically subclass from an existing class. Therefore, we discuss the technical challenges faced to integrate inheritance into a Java based scripting language.

This chapter is structured as follows: in Section 4.1 we look at Java based scripting languages and investigate how they deal with subclassing from Java classes and the implementation of Java interfaces. Section 4.2 focuses on the architecture of the Java Virtual Machine and in Section 4.3 we illustrate how we can use this architecture to dynamically generate Java classes.

Classes are generated using byte code which can be quite cumbersome. Therefore, in Section 4.4 we take a look at the byte code engineering library BCEL which offers a high level view on byte code generation.

4.1 Java Based Scripting Languages

There are many Java based scripting languages. The promise of using the Java platform are manifold. The new scripting language runs on any platform that provides a Java Virtual Machine. Improvements to the virtual machine, the language or the libraries are immediately available to the scripting language. Using reflection, the numerous Java libraries can be used without having to write any glue code. Finally, compiling the scripting language to Java byte code can lead to a performance similar to that of code written in Java.

The web site *Programming Languages for the Java Virtual Machine*[1] lists more than 20 Java based scripting languages which we categorize as follows:

Special Purpose Languages. These languages are explicitly built for a special problem domain. For example, automating Web tasks (WebL, PolyJsp, Resin) or scripting network components (Netscript) or server-side scripting (iScript) are among the tasks addressed. Scripting arbitrary Java objects is not the design goal of these languages.

Implementation of Scripting Languages. Many existing scripting languages have been ported to the Java platform: Ruby (JRuby), Python (Jython), Tcl (Jacl) or JavaScript (Rhino, EcmaScript). Their aim is to implement the existing scripting language as closely as possible.

Interestingly, there are also several attempts to implement Java as a scripting language (Beanshell, DynamicJava). While largely preserving the syntax and semantics of the statically typed Java language, their aim is to enhance it with typical features of scripting languages such as dynamic typing.

Typically, these implementations try hard to seamlessly integrate any Java objects into the scripting language and make them easily and consistently accessible.

New General Purpose Language. The Java platform is used to implement a new general purpose language with its own syntax and semantics. Seamless access to Java objects is not the main issue here.

In order to understand how to integrate inheritance into a Java based scripting language we concentrate on the implementations of scripting languages in Java and investigate how they deal with inheritance. We want to answer the following two questions:

1. How is the problem of subclassing handled in Java based scripting language?

2. How are interfaces implemented in Java based scripting language?

To answer these questions we take a look at two scripting languages, i.e., BeanShell and Jython. BeanShell is designed as a dynamically typed scripting language with the goal of following Java's syntax as closely as possible. Jython is an implementation of the Python language in Java.

We briefly introduce these languages and look at a small code snippet. A complete coverage of the two scripting languages is not the aim of this chapter, we concentrate on answering the two previously posed questions.

[1] see: `grunge.cs.tu-berlin.de/~tolk/vmlanguages.html`

4.1.1 BeanShell

Overview

BeanShell [Nie02][2] bridges the standard Java language into the scripting domain in a natural way by allowing the developer to relax types where appropriate. It is possible to write BeanShell scripts that look exactly like Java code. Additionally, it is possible to write scripts that look more like scripts of a traditional scripting language, while still maintaining the framework of the Java syntax.

The approach BeanShell takes to scripting is interesting in two ways: first, it offers a way of combining the power of a scripting language with the power of a system language. Using BeanShell, we can rapidly build prototypes taking advantage of dynamic typing and the lack of a compile phase while writing programs. Once we are convinced of our code, we add types to it and – ideally – compile it, giving us all the benefits of compiled code. Second, from an educational point of view, it offers an alternative way of teaching Java.

Example

On one hand, Beanshell closely follows Java syntax and semantics. On the other hand, it offers us the advantages of a dynamically typed language by allowing us to omit type declarations. In this case, BeanShell does the type inferences at runtime.

As an example, popping up a frame with a button in it can be written adhering to Java's syntax:

```
JButton button = new JButton( "My Button" );
JFrame frame = new JFrame( "My Frame" );
frame.getContentPane().add( button, "Center" );
frame.pack();
frame.setVisible(true);
```

The code above is valid Java code. If we want to omit types, we can rewrite the same example as follows:

```
button = new JButton( "My Button" );
frame = new JFrame( "My Frame" );
frame.getContentPane().add( button, "Center" );
frame.pack();
frame.setVisible(true);
```

In this case, BeanShell infers the correct types for us at runtime.

[2] see: `www.beanshell.org`

```
foo() {
    int a = 42;
    bar() { print("The bar is open!"); }
    bar();
    return this;
    }
// Construct the foo object
fooObj = foo();
// prints "the bar is open!"
// Print a variable of the foo object
print ( fooObj.a )
// 42
// Invoke a method on the foo object
fooObj.bar();
// prints "the bar is open!"
```

Figure 4.1: Using a method closure to model an object.

Handling of Subclassing

The current version of BeanShell does not support the generation of classes at runtime. Instead, BeanShell enables us to script objects as *method closures*, similar to the way it is done in Perl, JavaScript, and other object capable scripting languages. An example of using a method closure is given in Figure 4.1.

Within a method closure we can define variables as well as methods and call them inside this scope. Returning `this` in a closure gives us access to the method scope and enables us to access variables and call methods defined therein.

Conceptually, method closures very much behave like real objects, but they are not true objects in the Java sense. We cannot pass them as objects to Java methods as they are not instances of a Java class. Method closures can only be used to model objects within BeanShell.

Handling of Interface Implementation

Scripting an interface in BeanShell works by looking for scripted methods to implement the methods of the interface type. A Java method invocation on a script that implements an interface causes BeanShell to look for a corresponding scripted method with the same signature. BeanShell then invokes that method, passes along the arguments and finally passes back any return value.

BeanShell uses the `Proxy` class that was added to the Java as of version 1.3. Using the `Proxy` class we can dynamically create a proxy class. A dynamic proxy class is a class that

implements a list of interfaces specified at runtime when the class is created. A proxy interface is such an interface that is implemented by a proxy class. Each proxy instance has an associated invocation handler object, which implements the interface `InvocationHandler`. A method invocation on a proxy instance through one of its proxy interfaces is dispatched to the invoke method of the instance's invocation handler and passes the proxy instance, a `java.lang.reflect.Method` object identifying the method that was invoked, and an array of type `Object` containing the arguments. The invocation handler processes the method invocation and the result of the method invocation on the proxy instance is the value returned.

The code given next illustrates how we can create a proxy for some interface `Foo` using the `Proxy` class. We have to pass the class loader, the list of interfaces for the proxy class to implement and the invocation handler to dispatch method invocations to:

```
Foo f = (Foo) Proxy.newProxyInstance(Foo.class.getClassLoader(),
                                     new Class[] { Foo.class },
                                     handler);
```

The handler processes a method invocation on a proxy instance and dispatches to its `invoke` method which expects the proxy, a method and an array consisting of the arguments of the method as arguments and returns the value of the method invocation.

Using this proxy mechanism, BeanShell is able to define arbitrary Java interfaces without having to generate any byte code.

4.1.2 Jython

Overview

Python [Lut96] is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built-in data structures, combined with dynamic typing and dynamic binding, make it very attractive for rapid prototyping and gluing existing components together. Python supports objects, classes, single and multiple inheritance and modules as well as a runtime meta-object protocol.

Similar to Smalltalk, Python uses objects as a unifying concept. Everything is an object. Objects are first-class values and, therefore, any abstraction can be used as a first-class value. Functions are objects too, and using a lambda construct it is possible to define anonymous functions which can be passed as arguments.

Jython [PR02][3] is an implementation of the Python programming language written in Java, enabling Python programs to interact seamlessly with any Java components. Jython integrates Java and Python tightly, by offering subclassing of Java classes in Jython and dynamic compilation of Python code to Java byte code.

[3] see: `www.jython.org`

Example

Jython tries to implement Python as closely as possible while giving full access to Java classes. The following example subclasses `java.awt.Color`, implements the interface `java.lang.Comparable` and overrides the method `java.lang.Object.toString`.

```
from java.awt import Color
from java.lang import Comparable

class ComparableColor(Color, Comparable):
    def __init__(self, color):
        Color.__init__(self, color.getRGB())

    def _RGBTuple(self):
       return (self.getRed(), self.getGreen(), self.getBlue())

    def compareTo(self, color1):
        rgb0 = self._RGBTuple()
        rgb1 = color1._RGBTuple()
        return cmp(rgb0, rgb1)

    def toString(self):
        return "#%02x%02x%02x" % self._RGBTuple()
```

This example nicely illustrates the hybrid nature of the created Python class. Looked at from the side of Python, this class inherits from two classes: `java.awt.Color` and `java.lang.Comparable`. Looked at the class from the Java side, this class inherits from one class `java.awt.Color` and implements an interface `java.lang.Comparable`.

In Python, constructors are defined in a `__init__` method and are automatically called when a new instance is created. In the previous example the classes inherited are in fact Java classes, so at creation time Java objects thereof have to be instantiated. Therefore Jython automatically calls the constructor of the parent Java class at the end of the `__init__` method.

Furthermore, the class `ComparableColor` overrides the methods `compareTo(Object)` as well as `toString()`. Whenever these methods are called (on the Python as well as on the Java side) the calls get redirected to the code written in Python. For example, when the Java `Collections.sort()` method is called, the collection is sorted according to the criteria defined in the Python `compareTo(self, color1)` method.

Jython takes great effort to support inheritance of Java classes by incorporating them into the Python inheritance model. There is only one short coming: Python supports multiple inheritance which is limited in Jython as it is not possible to subclass from multiple Java classes. This is a limitation of Java's class file format (see Section 4.2.1).

Handling of Subclassing

A Jython object can inherit from at most one Java class and a set of Java interfaces, in addition to an arbitrary number of Python classes. When Jython encounters a `class` statement at runtime, it creates a Python class object and binds it to the class name. If the Jython class inherits from a Java class, Jython also dynamically creates a proxy class in Java, which is associated to the Python class.

The Java proxy class extends the specified Java parent class. If the Java inheritance is indirect through another Jython class, it extends the proxy class of that Jython class.

The proxy class is dynamically created using byte code and loaded using a custom class loader. We investigate in detail how this mechanism can be implemented on the Java platform in the rest of this chapter.

Handling of Interface Implementation

The Java proxy class that is created whenever a Java class is specified as a parent class implements any Java interfaces in the base classes list. If only interfaces are specified, `java.lang.Object` is used as the Java parent class.

As Jython is able to dynamically generate classes and load them into the Java Virtual Machine, implementing interfaces comes almost for free. All there is to do is to dynamically create a class that implements the given interfaces.

4.1.3 Conclusion

BeanShell nicely illustrates how tightly we can integrate Java into a scripting language without generating classes at runtime. Using the `Proxy` class of the `java.lang.reflect` package we can implement Java interfaces. Subclassing of Java classes, however, is not possible. This limitation drastically reduces the interoperability with the Java platform, as we can not use subclassing as a powerful means of reuse.

Jython, in contrast, offers a powerful mechanism of subclassing from Java and Python classes. Subclassing from a Java class, as well as implementing arbitrary Java interfaces, is possible. Only multiple inheritance from Java classes is not possible due to a limitation in Java's class file format.

In order for Jython to integrate Python and Java this tightly, an infrastructure to dynamically generate byte code and load it into the Java Virtual Machine using a customized class loader is needed.

If we want to benefit from dynamic subclassing in a scripting language, we must understand how the Java Virtual Machine machine works. Therefore, in the next section we take a close look at the Java Virtual Machine and the structure and execution of its byte code.

4.2 The Java Virtual Machine

The Java Virtual Machine [LY99] is the cornerstone of the Java platform. It is the component of the platform responsible for its hardware and operating system independence, the small size of its compiled code, and its ability to protect users from malicious programs.

The Java Virtual Machine is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at runtime. The Java Virtual Machine does not assume any particular implementation technology, host hardware, or host operating system.

The Java Virtual Machine knows nothing of the Java programming language, only of a particular binary format, the *class file* format. A class file contains Java Virtual Machine instructions (or byte codes) and a symbol table, as well as other ancillary information.

For the sake of security, the Java Virtual Machine imposes strong format and structural constraints on the code in a class file. However, any language that can be expressed in terms of Java Virtual Machine operations can be run by the Java virtual machine.

4.2.1 Class File Format

The class file format is the system-independent binary format used to represent a compiled class or interface. Each class file contains the definition of a single class or interface only.

A class file consists of a single `ClassFile` structure as given in Figure 4.2[4]. It starts with a magic number (`0xCAFEBABE`) and some versioning information. The constant pool contains the text segments of the class. The access flag defines access permissions and other properties of this class, e.g., whether this class is an interface, abstract or final. The name of the class is next. Every class has exactly one superclass and can implement 0 to n interfaces, defined by the next three entries. Two arrays give a complete description of all the fields and the methods of the class. The last array defines attributes of the class, such as the actual code of a method, references to source file line numbers or information about whether certain elements of a class are deprecated.

Figure 4.3 shows the structure of a simple `HelloWorld` class. Its simplified class file structure is given in the left of the figure. The rounded box illustrates a simple method which prints "Hello, world" to the `System.out` stream. The array of methods in the class file structure contains a reference to this method structure, which itself has a reference to a code attribute, containing the actual code. The code itself has various references into the constant pool illustrated by the text in the shadowed rectangle. For example,the "Hello, world" string is stored in the constant pool.

[4] `u1`, `u2` and `u4` represent one-, two-, or four-byte quantities.

```
ClassFile {
        u4 magic;
        u2 minor_version;
        u2 major_version;
        u2 constant_pool_count;
        cp_info constant_pool[constant_pool_count-1];
        u2 access_flags;
        u2 this_class;
        u2 super_class;
        u2 interfaces_count;
        u2 interfaces[interfaces_count];
        u2 fields_count;
        field_info fields[fields_count];
        u2 methods_count;
        method_info methods[methods_count];
        u2 attributes_count;
        attribute_info attributes[attributes_count];
    }
```

Figure 4.2: The class file structure.

4.2.2 Execution

The Java Virtual Machine dynamically loads, links, and initializes classes and interfaces. *Loading* is the process of finding the binary representation of a class or interface type with a particular name and creating a class or interface from that binary representation. *Linking* is the process of taking a class or an interface and adding it to the runtime state of the Java Virtual Machine so that it can be executed. *Initialization* of a class or an interface consists of executing the initialization method of a class or an interface.

If the initial attempt to execute the method `main` of a class reveals that the class is not loaded – that is, the virtual machine does not currently contain a binary representation for this class – the virtual machine uses a *class loader* to attempt to find such a binary representation. If this process fails, an exception is thrown.

After the class is loaded, it must be initialized before `main` can be invoked. A type (class or interface) must always be linked before it is initialized. Linking involves *verification*, *preparation*, and, optionally, *resolution* of a class.

Verification checks that the loaded representation of a class is well formed and contains a proper symbol table. Furthermore, it checks that the code that implements a class obeys the semantic requirements of the Java Virtual Machine. If a problem is detected during verification an exception is thrown.

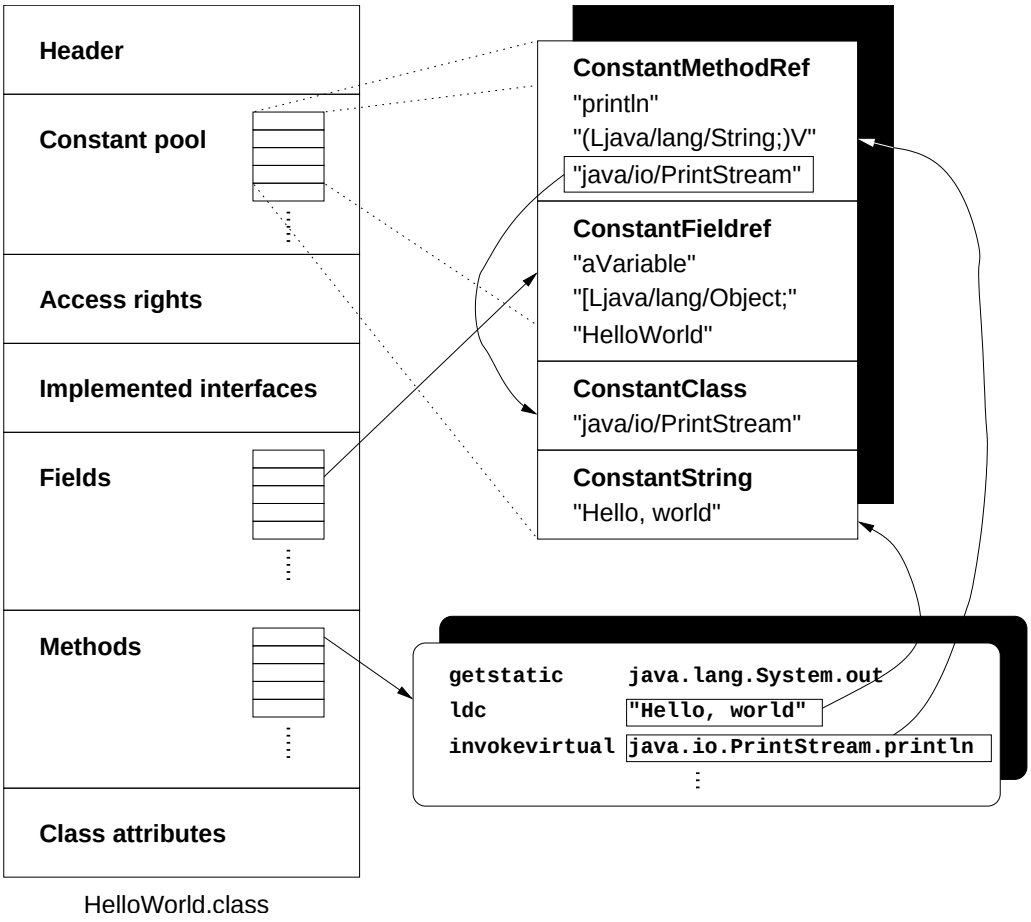

Figure 4.3: The class file format of the `HelloWorld` class.

The preparation process involves allocating any data structures that are used internally by the virtual machine, e.g., method tables, and creating static fields for a class or an interface and initializing them with default values.

Resolution checks symbolic references from one class to other classes and interfaces by loading the other classes and interfaces that are referenced and checks that these references are correct. As long as the semantic is retained, this process can be done lazily.

At the level of the Java Virtual Machine, every constructor appears as an instance initialization method that has the special name `<init>`. A class or an interface has at most one class or interface initialization method and is initialized by invoking that method. The initialization method of a class or an interface is static and takes no arguments. It has the special name `<clinit>`.

A new frame is created each time a method is invoked. Additionally, a new operand stack and a set of local variables for use by that method are allocated. The new frame is referred to as the current frame, and its method is known as the current method. When a method is invoked, a new frame is created. On method return, the current frame passes back the result of its method invocation to the previous frame. This frame is then discarded and the previous frame becomes the current one.

4.2.3 Instruction Set

The Java Virtual Machine is stack-oriented, with most operations popping one or more operands from the operand stack of the current frame or pushing results back onto the operand stack.

The instruction set of the Java Virtual Machine often distinguishes operand types by using distinct byte codes for operations on its various data types. For example, the instructions chosen to operate on type data (`iconst_0, istore_1, iinc, iload_1, if_icmplt`) are all specialized for type `int`. Furthermore, the Java Virtual Machine takes advantage of the likelihood of certain operands (`int` constants -1, 0, 1, 2, 3, 4 and 5 are encoded by the `iconst_<i>` instructions) by making those operands implicit in the opcode.

Currently, the Java Virtual Machine Specification defines 212 instructions which can be roughly grouped into the following subsets [Dah01]:

Stack operations. Constants can be pushed onto the stack either by loading them from the constant pool with the `ldc` instruction or with special "short-cut" instructions where the operand is encoded into the instructions, e.g., `iconst_0` or `bipush` (push byte value).

Arithmetic operations. The instruction set of the Java Virtual Machine distinguishes its operand types using different instructions to operate on values of specific type. For example, arithmetic operations starting with `i` denote an integer operation. Among these instructions, `iadd` adds two integers and pushes the result back on the stack.

Control flow. There are branch instructions like `goto` or `if_icmpeq`, which compares two integers for equality. There is also a `jsr` (jump sub-routine) and `ret` pair of instructions that is used to implement the `finally` clause of `try-catch` blocks. Exceptions may be thrown with the `athrow` instruction.

Branch targets are coded as offsets from the current byte code position, i.e., with an integer number.

Load and store operations. Depending on their type, there are special load and store operations, e.g., local integer variables using `iload` and `istore`. There are also array operations like `iastore` that stores an integer value into an array.

Field access. The value of an instance field may be retrieved with `getfield` and written with `putfield`. For static fields, there are `getstatic` and `putstatic` counterparts.

Method invocation. Methods may either be called via static references with `invokestatic` or be bound virtually with the `invokevirtual` instruction. Super class methods and private methods are invoked with `invokespecial`.

Object allocation. Class instances are allocated with the `new` instruction, arrays of basic type like `int[]` with `newarray`, arrays of references like `String[][]` with `anewarray` or `multianewarray`.

Conversion and type checking. For stack operands of basic type there exist casting operations like `f2i` which converts a float value into an integer. The validity of a type cast may be checked with `checkcast` and the `instanceof` operator can be directly mapped to the equally named instruction.

4.3 Dynamic Class Loading

Dynamic class loading is an important feature of the Java Virtual Machine [Gon98, LB98]. It gives Java the ability to load software components at runtime. The Java dynamic class loading incorporates the following four features:

1. *Lazy loading.* Classes are loaded on demand as late as possible.
2. *Type-safe linkage.* Dynamic class loading must not violate the type safety of the Java Virtual Machine and does not require additional runtime checks to guarantee type safety.
3. *User-definable class loading policy.* Class loaders are first-class objects.
4. *Multiple namespaces.* Class loaders provide separate namespaces for different software components.

4.3.1 Class Loaders

The purpose of a class loader is to support dynamic class loading. Classes are distributed using a machine-independent, standard, binary representation known as the class file format. Classes can come from various, potentionally hostile sources, e.g., they can be downloaded from the Internet. The class loader is responsible for resolving a symbolic reference to a class by loading a class file and creating the class type.

A class loader L that loads a class C is the *defining* class loader of a class. The actual class type is fully qualified by a tuple {C,L}, consisting of the class C and its class loader L. Two types in the Java runtime environment are equal if, and only if, both the class types are equal and their defining class loaders are identical.

The root of the class hierarchy is an abstract class called `java.lang.ClassLoader`. The class `java.Security.SecureClassLoader`, introduced in JDK 1.2, is a subclass and a concrete implementation of the abstract superclass and itself is a superclass of various other class loaders.

As class loaders are themselves classes, a chicken-and-egg question arises: Where does the first class loader come from? It comes from a *bootstrap* class loader that bootstraps the class loading process. Even though responsible for the loading of system classes it does not manifest itself in the Java context, as it is generally written in C.

Obviously, class loaders are ordinary objects too, they are instances of subclasses of the class `java.lang.ClassLoader`. These instances interact by delegation: when the Java Virtual Machine asks one class loader to load a class, this class loader either loads the class itself or asks another class loader to do so.

The following class resolution is executed whenever the Java Virtual Machine searches for classes:

1. Check if the class has already been loaded.
2. Delegate, if the current class loader has specified a delegation pattern, otherwise delegate to the bootstrap class loader.
3. Call a customizable method to find the class elsewhere.

Step 3 provides a way to customize the mechanism that looks for classes. A custom class loader can override this method to specify its own lookup mechanism.

4.3.2 Verification of Class Files

The Java language was designed from the ground up with security in mind. Type-safety and lack of pointer arithmetic are two of the best known security features of Java. Lesser

known is the fact that before any class file is executed, it has to pass a thorough verification process in order to guarantee that its code is well behaved.

A *class file verifier* is responsible for checking these constraints and guaranteeing that a given code is well behaved. For any code that passes verification, the interpreter knows that the code adheres to constraints such as:

- There are no stack overflows or underflows.
- All register accesses and stores are valid.
- The parameters to all byte code instructions are correct.

In order to guarantee these constraints the Java Virtual Machine Specification [LY99] describes a four pass verification process: pass one consists of loading a class file into the Java Virtual Machine and pass two verifies that the loaded class file information is consistent. Pass three verifies that the program code is well-behaved and, finally, pass four verifies constraints that conceptually belong to pass three but are delayed to runtime for performance reasons.

Pass one is simple: checks are conducted to ensure that the class file is not truncated, that it starts with the correct magic number and that all attributes are recognized and have correct length.

Passes two and three are performed during resolution of a class file. Resolution defines the transformation of a symbolic reference to an actual reference in the virtual machine. This pass performs all verification that can be performed without looking at the actual byte code. For example, it ensures that a final class is not subclassed and that final methods are not overridden. Further checks assure that all classes except `java.lang.Object` have a direct superclass and that all references have valid name, class and type descriptors.

Actually verifying the byte code is done on passes three and four. *Static* constraints can be verified easily. They guarantee that no instruction can access or modify a local variable at an index equal or greater than the number of local variables a method allocates, that all references to the constant pool have an appropriate type, and that execution can not fall off the end of code.

Structural constraints – better named *dynamic* constraints – must be assured using a *data flow analyzer*, an algorithm sketched by Sun based on control flow graphs. While its correctness is hard to prove, intuitively, its job is to guarantee that the actual byte code is well behaved, e.g., at any given point during execution the operand stack never exceeds the maximal size and the types on it do not change during execution.

The final pass – pass four – performs verifications that in principle could be performed in pass three. Instead, as a performance enhancement, they are delayed until runtime as they could trigger the loading of additional class files. The task of this pass is to ensure

that the referenced method or field exists, that it has the correct signature and that the executing method has the correct access rights.

If a class file passes these four verification passes the constrains imposed on the class structure hold and prove that its binary representation is structurally correct.

Note that there are tests that nevertheless have to be performed at runtime. They are not part of the verification process, e.g., the `java.lang.ClassCastException` is thrown when an instruction attempts to cast an object to a subclass of which it is not an instance.

4.4 Dynamic Generation of Java Byte Code

The previous section indicates that writing valid byte code can be difficult and cumbersome. Many aspects have to be considered: for example, the Java Virtual Machine specification requires that every method code declares the maximal depth of the operand stack as well as the maximal number of local variables it uses. On a lower level, the handling of the binary opcodes of each instruction can be quite tricky. Therefore, a byte code framework should hide the tedious details from the programmer and provide a higher level view of the byte code engineering task.

There are quite some byte code frameworks we are aware of: BCEL, gnu.bytecode, Trove, Jikes Bytecode Toolkit, or Serp[5]. While all these frameworks provide full byte code engineering support, we chose BCEL for several reasons: it is actively developed as part of the Apache Jakarta project, open source and well documented. Most importantly however, it has been successfully used in scientific projects which include adding genericity to Java [BD98] or implementing the functional language Funnel [Ode00].

4.4.1 Byte Code Engineering with the BCEL API

The BCEL API (Byte Code Engineering Library), formerly known as JavaClass, is a toolkit for the static analysis and dynamic creation or transformation of Java class files. It enables developers to implement the desired features on a high level of abstraction without having to handle all the internal details of the Java class file format and thus re-invent the wheel every time.

The BCEL API abstracts from the concrete circumstances of the Java Virtual Machine and the details how to read and write binary Java class files. The API consists of the following three parts:

[5] BCEL is available at `jakarta.apache.org/bcel`, gnu.bytecode is part of Kawa, the Java-based Scheme system at `www.gnu.org/software/kawa`, Jikes Bytecode Toolkit is available at `alphaworks.ibm.com/tech/jikesbt`. Trove is part of the TeaServlet template engine available at `opensource.go.com/Trove` and, finally, Serp is a available at `serp.sourceforge.net`.

1. A package containing classes that describe "static" constraints of class files, i.e., that reflect the class file format and are not intended for byte code modification. Its classes may be used to read and write class files. This property is especially useful for analyzing Java classes that are accessible in binary form only. Key to this part of the library is the `JavaClass` class.

2. A package to dynamically generate or modify `JavaClass` objects. It may be used to insert analysis code, to strip unnecessary information from class files, or to implement the code generator back-end of a Java compiler.

3. Various code examples and utilities like a class file viewer, a tool to convert class files into HTML, and a converter from class files to the Jasmin assembly language [MD97].

All of the binary components and data structures declared in the Java Virtual Machine specification [LY99] are mapped to classes. The top-level data structure is the `JavaClass` class, which most often is created by a `ClassParser` object that is capable of parsing binary class files. A `JavaClass` object basically consists of fields, methods, symbolic references to the superclass and to implemented interfaces. The class `ConstantPool` serves as a central repository. `Field` and `Method` model fields and methods, and can have various attributes, e.g., the class `Code` is used to describe the code of a given method. An overview of the `JavaClass` part of the BCEL API is given in Figure 4.4.

The `ClassGen` part of the API supplies an abstraction layer to create or transform class files dynamically (see Figure 4.5). The generic constant pool, for example, is implemented by the class `ConstantPoolGen` and provides methods for adding different types of constants. Accordingly, the class `ClassGen` offers an interface to add methods, fields and attributes. The class `Type` abstracts from the concrete details of the type signature and the classes `FieldGen` and `MethodGen` abstract from the details of field and method creation.

Every instruction of the Java Virtual Machine is modeled by a class. This may look somehow odd at first sight, but it provides a high-level view upon control flow.

Instructions are inserted in a `InstructionList`. When the instruction list of a method is constructed we can transform it to real byte code, with BCEL taking care of many details such as offset generation and instruction encoding.

To further facilitate the creation of byte code, the BCEL API provides *instruction factories*. For example, pushing constants onto the operand stack can be achieved in different ways: `iconst_m1`, ..., `iconst_5` for numbers between $-1, \ldots, 5$, other possibilities are `bipush` (for values between -128 and 127), `sipush` (for values between -32768 and 32767), or `ldc` (load constant from the constant pool). The instruction `PUSH` relieves the programmer from the boring job of picking the right one.

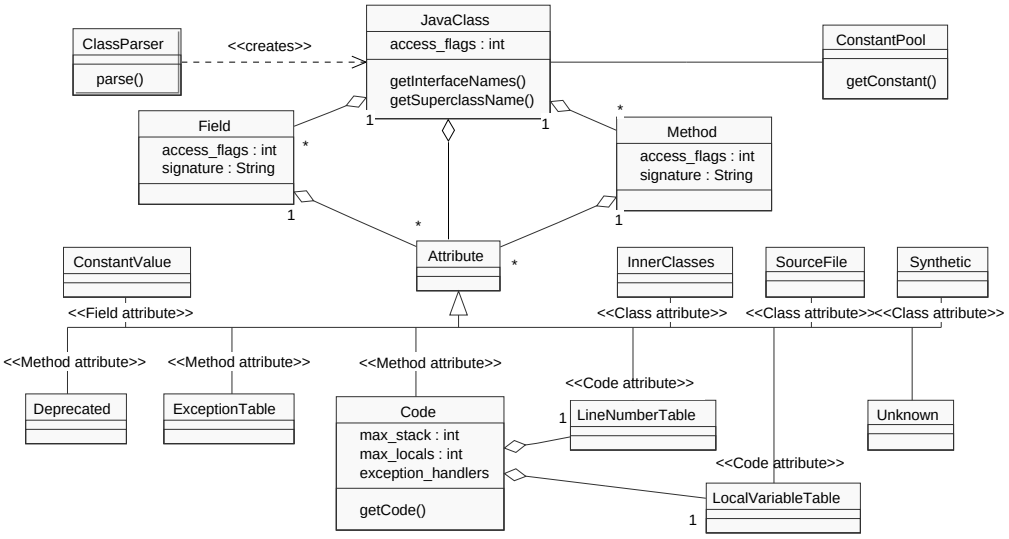

Figure 4.4: UML diagram for the `JavaClass` part of the BCEL API.

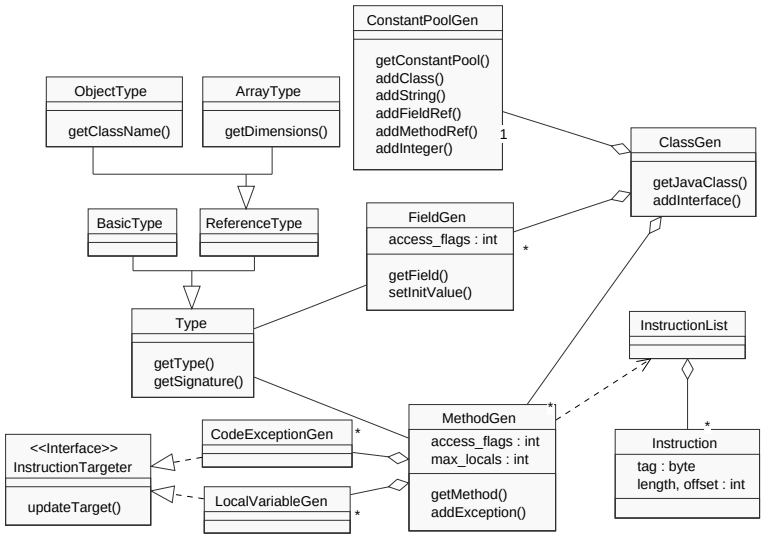

Figure 4.5: UML diagram of the `ClassGen` part of the BCEL API.

4.4.2 Using BCEL– An Example

In this section, we illustrate how to generate a class file using BCEL. The simple example given in Figure 4.6 implements the same functionality as the Java code presented next:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

If we compile this Java code, we get the following byte code:

```
Method HelloWorld()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

Method void main(java.lang.String[])
   0 getstatic #2 <Field java.io.PrintStream out>
   3 ldc #3 <String "Hello World">
   5 invokevirtual #4 <Method void println(java.lang.String)>
   8 return
```

Having the Java as well as the byte code in front of us, it is easy to understand the code given in Figure 4.6. First, we create a `ClassGen` object. We add an empty constructor and the description of the `main` method to the class. Next, we create the instructions that belong to this method. We create an `InstructionList` object and by using an `InstructionFactory` object, we add the instructions one by one.

We have to set the maximum depth of the operand stack and the maximal stack size for every method. In the simple example given in Figure 4.6, we set the maximum depth of the operand stack by hand. However, BCEL can compute the maximal stack depth and size by performing control flow analysis.

Last we add the generated method `main` to the `ClassGen` object. The class is now constructed and can be transformed to byte code.

4.4.3 JustIce

Creating class files from scratch can be tedious and frustrating. It is easy to create class files that violate the constraints imposed by the class verifier. Trying to load them into the Java Virtual Machine forces it to throw an exception and quit. Unfortunately, the exception is often of little value to the programmer, as it does not inform the programmer where exactly, and in which verification pass, a constraint violation has occurred.

```
/* Generate a class file */

ClassGen cg = new ClassGen("HelloWorld", "java.lang.Object",
                           "<generated>",
                           ACC_PUBLIC | ACC_SUPER,
                           null);
ConstantPoolGen cp = cg.getConstantPool();
InstructionList il = new InstructionList();

/* Generate an empty constructor */

cg.addEmptyConstructor(ACC_PUBLIC);


/* Generate a method description */

MethodGen mg = new MethodGen(ACC_STATIC | ACC_PUBLIC,
                             Type.VOID,
                             new Type[] {new ArrayType(Type.STRING, 1)},
                             new String[] {"argv"},
                             "main", "HelloWorld",
                             il, cp);

/* Generate byte code using an instruction factory */

InstructionFactory factory = new InstructionFactory(cg);

il.append(factory.createFieldAccess("java.lang.System", "out",
                            new ObjectType("java.io.PrintStream"),
                            Constants.GETSTATIC));
il.append(new PUSH(cp, "Hello World"));
il.append(factory.createInvoke("java.io.PrintStream", "println",
                            Type.VOID, new Type[] {Type.STRING},
                            Constants.INVOKEVIRTUAL));
il.append(factory.createReturn(Type.VOID));

/*  Set the maximum depth of the operand stack */

mg.setMaxStack(2);
cg.addMethod(mg.getMethod());
il.dispose();

/* Finalize the created class and dump it to a file */

try {
    cg.getJavaClass().dump("HelloWorld.class");
} catch (java.io.IOException e) {System.err.println(e);}
```

Figure 4.6: Writing a `HelloWorld` class using BCEL.

A recent addition to BCEL greatly facilitates the work of the byte code engineer. JustIce [Haa01] is a free class file verifier, written using the BCEL API. One of its design goals is to be verbose: whenever a verification error occurs, exact information about the offending code, its location and the verification pass responsible for the error is given. This information is of great value when we test and debug generated class files.

Chapter 5

A Language Extension for White-Box Reuse in JPiccola

In this chapter we present a language extension that enables us to integrate white-box reuse into JPiccola. At a high level of abstraction, we can generate arbitrary classes, e.g., we can subclass from Java classes and implement Java interfaces. The language extension consists of two parts, the Java part which is responsible for generating a class and loading it into the Java Virtual Machine and the Piccola part which is responsible for providing seamless access to the functionality of the Java part of the language extension.

This chapter is structured as follows: in Section 5.1 we argue why we need this language extension for JPiccola. In Section 5.2 we give an overview of the language extension and its basic concepts. In the next two sections we introduce the two parts of the language extension: in Section 5.3 we describe the Java part of the language extension. We describe the structure and the functioning of a generated class, how we load it into the Java Virtual Machine and how we can use reflection to facilitate class generation. In Section 5.4 we describe the Piccola part of the language extension. We illustrate how we can define the structure of a class at a high-level of abstraction and how we can access the functionality provided by the Java part.

5.1 Motivation for the Language Extension

There are two reasons why we need this language extension. From a practical point of view, in the previous version of JPiccola an easy and elegant way to reuse Java components by inheritance did not exist. If we wanted to subclass a Java class, we had to leave JPiccola, write a Java wrapper class mainly consisting of boiler plate code, compile it, add it to the class path of JPiccola and restart it. Reusing components by inheritance was tedious and limited. Furthermore, each wrapper class had to be included in the JPiccola bridge and bloated the JPiccola distribution.

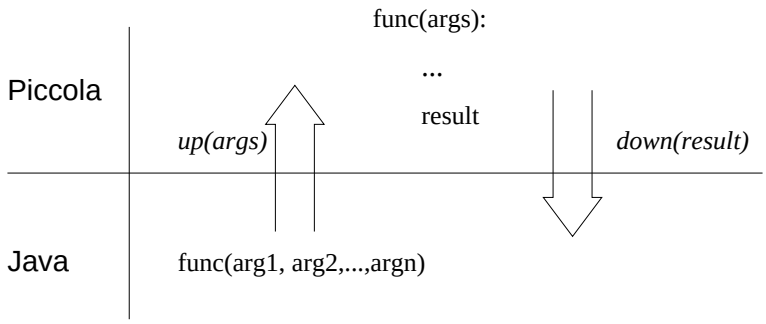

Figure 5.1: Passing arguments between Java and Piccola.

More importantly, from a theoretical point of view, if we want to use frameworks written in Java, we often have to provide objects of a certain type from JPiccola. As an example, if a framework requires an interface of type I then we have to provide an object that implements this interface I. To create such an object, we need a class implementing this type. Furthermore, if we want to do this dynamically, we need a mechanism to generate a class on the fly. To accomplish this on the Java platform, we need to dynamically generate byte code and load it in to the Java Virtual Machine.

Note that we use the term *language extension* in the sense that we drastically extend the class of components easily accessible from JPiccola. The language extension is implemented at the virtual machine layer of JPiccola: we extend its bridge with the functionality to generate and load Java classes dynamically. We neither introduce any new language constructs nor do we change the semantics of Piccola.

5.2 Overview of the Language Extension

In this section, we give an overview of our language extension. The key idea behind the language extension is to create Java objects that delegate all calls to their methods to Piccola services.

More specifically, we dynamically create subclasses of Java classes and load them into the Java Virtual Machine. Instances of these classes delegate any call to a method to a corresponding Piccola service. We pass the arguments to the service which processes the call. If the service returns a result we pass it back to Java. In the terminology introduced in Section 3.1.3 we *up* the arguments of the Java call and *down* the result of the Piccola script. This is illustrated in Figure 5.1.

The benefit of our approach is that the generated class is a real Java class that can be passed as an argument to a Java method. Its behavior we nonetheless completely define from JPiccola. We can thus use inheritance in JPiccola and combine its power with the ease of scripting.

In the rest of this section, we investigate how to implement this delegation scheme: first, we define the behavior of a dynamically generated class and, second, we sketch the structure of such a class.

5.2.1 Behavior of a Generated Class

Let us define a Java Class with one constructor and six public methods as defined below:

```
public class A {
    public A() {...}
    public void a() {...}
    public void b(Object obj) {...}
    public void c(int i) {...}
    public void d(Object obj_1, Object obj_2, Object obj_3) {...}
    public int e() {...}
    public int f(int i, int i) {...}
}
```

We want to delegate each Java method call to a Piccola service. We provide a form with services to delegate method calls to. We choose to use a simple mapping between Java calls and services. We delegate a call to a method with the name `name` to an equally named service called `name`. Additionally, we want to control constructor calls from Piccola. Therefore, we define a specially called service `initialize` which we map to the constructor call.

For the previously defined class `A`, the form `handler` with Piccola services to which we map Java method calls looks like this:

```
handler =
    initialize(): ...
    a(self): ...
    b(args): ...
    c(args): ...
    d(args): ...
    e(): ...
         result
    f(args): ...
         result
```

Next, we have to devise a strategy how to pass the arguments of a Java method to its corresponding Piccola service. We apply the following strategy: if the method does not take an argument, we pass a reference to the current object to the service only. This is similar to the way `self` is passed, as the first argument, to a method in Python and provides us access to the Java object. If the method has one or more arguments we have

to distinguish between two situations: if the argument is a non primitive type, we can just add it to the list of arguments. Otherwise, we convert it to the corresponding object type before adding it to the list of arguments.

Similarly, we have to distinguish between primitive and non primitive return types. In JPiccola we deal with Java objects only. If the return type of the Java method is a Java object we can just pass it down. Else the return type is a primitive type and we have to convert the object to a primitive type.

Additionally, we want a class to behave lazily in the sense that if a function on the Java side is never called we do not have to define a service. However, if there is a call to a Java method but no service is defined, an exception is thrown. If we are not interested in a Java call we nonetheless have to provide an empty service handling it.

We considered dropping this constraint, so that if there was no service handling a Java call, we would silently ignore this. The benefit of this approach would be that we would have to define services only for the calls we are interested in, ignoring the rest. However, this has the drawback that if we forget to define a service by mistake, we are never notified of our shortcoming. For this reason we rejected this idea.

Finally, we want to be able to change the behavior of a method at runtime by changing the Piccola services mapped to a Java object.

5.2.2 Structure of a Generated Class

In this section we look at the actual structure of the generated classes to implement the behavior described in the previous section.

We pass arguments of a Java method call to a Piccola service by creating an instance of `ArgumentList`. First, we add a reference to the object to the list. Then we add the arguments one by one. If the argument is a primitive data type, we convert it to a Java object.

The method `getService(String)` returns the service mapped to this method. We try to execute the service represented by this form by calling `callback(ArgumentList)`. The list of arguments is upped and passed to the Piccola service. If the form does not represent a Piccola service, we throw an exception.

When the script of the Piccola service terminates it may return a result which we down and pass to Java. We are only using object types in JPiccola, so if the return type of the Java method is primitive, we have to transform the object type to a primitive type before passing it back to Java.

The example given next illustrates the code we generate to handle a call to the method `f` of class `A`[1]:

```
public int f(int i, int j) {
    ArgumentList args = new ArgumentList();
    args.add(this);
    args.add(new Integer(i));
    args.add(new Integer(j));
    return ((Integer) getService("f").callback(args)).intValue();
}
```

The service to handle this Java method call might look like this:

```
f(args):
    'i = args.at(1)      # Remember that a reference to the object
    'j = args.at(2)      #   is stored at the index 0.
    i+j                  # Returns the sum of i and j.
```

The structure of a generated class is straightforward (see Figure 5.3): the class consists of a member variable `m_delegate` that holds a reference to the associated form, two constructors, a method `initialize` to initialize an object and a method `setHandler` to set the form corresponding to it, plus all the methods we override. A detailed listing of a subclass of the class `A` is given in Appendix A.1.

Overloaded Methods

A Java class can have overloaded methods, i.e., it can have several methods with the same method name but different signatures.

Let us assume we have a class `B` with overloaded methods:

```
public class B {
    public void overloaded(int i) { ... }
    public void overloaded(String str) { ... }
    public void overloaded(String str_1, String str_2) { ... }
}
```

We map all calls to these three methods to one Piccola service. Within the script of this service we can distinguish which Java method was called by analyzing the size of the argument list and the type of their arguments. Note that we always pass a reference to the object at index 0, so if a method has n arguments the size of the argument list is $n + 1$.

[1]Note that we actually create byte code, we use the Java Code listing here to better illustrate our approach only. The full Java source code for the subclass of the class `A` is given in Appendix A.1.

```
overloaded(args):
    if (args.size() == 3)
        then:
           println "overloaded(String str_1, String str_2) called."
        else:
           if (args.at(1).getClass() == "java.lang.String")
                then: println "overloaded(String str) called"
                else: println "overloaded(int i) called"
```

More on `this` and `super`

In Java, the keyword `this` references to the object for which the instance method was invoked or to the object being constructed [GJS96]. The object referenced by `this` is a real object, i.e., we can pass it as an argument or return it. At the byte code level, the first local variable of a current frame always contains a reference to the current object. We can access it using the `aload_0` instruction.

Additionally, we considered adding access to `super` as well. The semantics of `super` in Java are to affect the lookup procedure for a method [MD97]. Let `C` be the direct superclass of the current class. Then if `C` contains a declaration of a method with the correct signature, this method is invoked and the lookup procedure terminates. Otherwise, the procedure is recursively performed using the direct superclass of `C`. If no matching method is found, an exception is thrown.

We decided against including a reference to the superclass because, at runtime, we do not know what `super` refers to as the keyword `super` has a static lookup. This lookup is illustrated in Figure 5.2: the direct superclass of `C` is `B`, but calling `method()` on an object of class `C` prints `A::method`. The class `C` inherits the method implementation of class `B` whose superclass is `A`, so `super` references to `A` even when called by an object of class `C`.

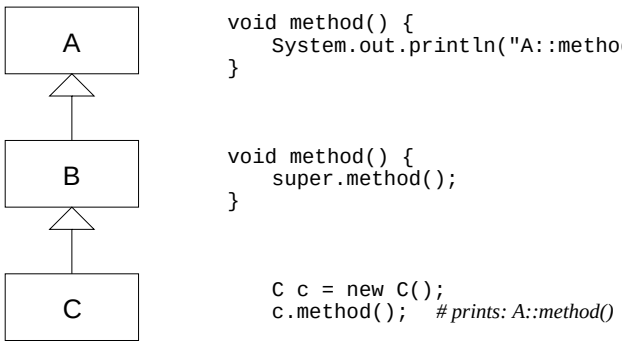

Figure 5.2: Static lookup of `super`.

Changing the Behavior of a Class at Runtime

In every generated class we include a Java method `setHandler(IForm)` that sets the form with the services to which we map the Java calls. Let us assume we have a class `X` with one method `version()`. Using the `setHandler` method, we can change the `version()` service at runtime:

```
handler =
   initialize(): ()
   version():  1
X.setHandler(handler)
X.version()             # prints: 1

handler = (handler, version():  2)    # override old version service
X.setHandler(handler)
X.version()             # prints: 2
```

This mechanism enables us to can change the behavior of a generated class without having to generate a new class.

The Darker Side of Java Byte Code

The overall structure of a generated class is fixed and easy to implement. However, we have to distinguish carefully between primitive types and object types and the instructions we generate to support them. We have to create different byte code instructions to collect the arguments to pass to Piccola and different byte code instructions to pass the result back to Java. The fact that `double` and `long` take up two slots in the local variable stack further complicates the code generation. We have to consider this when calculating the next free local variable. Additionally, we have to generate different code to handle arrays.

We rely heavily on the BCEL library to generate a class. Using its instruction factories, we can express the structure of a class at a high level of abstraction, e.g., we create the empty class file using one line of code.

Unfortunately, the BCEL library and the Java reflection package use different data structures to describe properties of a class, e.g., the access modifiers are expressed differently. Therefore we have to introduce some supporting classes which transform the data structures back and forth.

5.3 Java Part of the Language Extension

In this section we take a closer look at the Java part of the language extension. We present its public interface and briefly sketch its implementation.

5.3.1 Public Interface

The access point to the Java part of the language extension is the class `ByteCodeFactory` that uses the *singleton pattern* to provide access. The class provides the following four methods to generate a class:

```
public void generateClass(String className,
                          String superClassName,
                          ArrayList interfaceNameList,
                          int modifiers,
                          ArrayList methodDeclarationList)

public void generateSubClass(String superClassName)
public void generateInterface(String interfaceName)
public void generateInterfaces(ArrayList interfaceNames)
```

The method `generateClass` enables us to define arbitrary classes: we define its name, its superclass, the possible empty list of interfaces that it implements, its class modifiers and a list defining its methods. The definition of a method consists of its name, its modifiers, the return type and a list of formal parameters.

Most of the time, we do not want to define an arbitrary class. We do not want to add new methods to a class, we only want to subclass from an existing class or implement an interface with a fixed set of methods. In this case, defining the signatures of the methods by hand is superfluous. We can obtain this information using the Java Reflection package.

The `generateInterface` method uses reflection to get all the information needed to implement an interface. It takes the name of the interface class as its argument. Similarly, the method `generateInterfaces` allows us to implement a list of interfaces.

The purpose of the `generateSubClass` method is to create a direct subclass of a given class. Using reflection we can identify the methods we have to override. We have to be careful not to override public functions with special method modifiers, e.g., we must exclude methods that are *final* as overriding final methods is prohibited.

Additionally, the `ByteCodeFactory` class provides us hooks to load a dynamically generated class into the Java Virtual Machine and store it on disk. The two responsible methods are:

```
puplic Class loadClass()
public void saveClass(String path, String fileName)
```

In Section 5.4, we will illustrate how we use these methods from JPiccola.

5.3.2 Implementation

We use a variation of the *factory pattern* to actually generate the class. Instead of returning an object, our factories return Java byte code instructions to generate parts of the class.

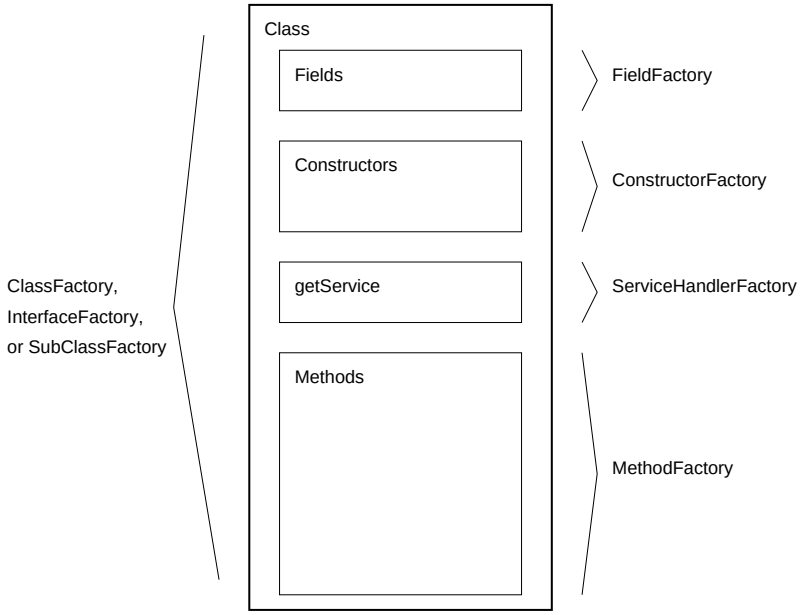

Figure 5.3: Schemata of the class generating process.

The class generation process is sketched in Figure 5.3. The generation is initiated by the `ClassFactory` or the reflection-based classes `InterfaceFactory` and `SubClassFactory`. These classes construct an empty class and then control the other factories which actually build the class.

The `FieldFactory` adds a member variable that stores a reference to the form associated to an instance of the generated class. The `ConstructorFactory` generates two constructors, a default one and one taking a form as its argument. The `ServiceHandlerFactory` creates the `getService` method which is responsible for projecting on the service corresponding to a method. Finally, the `MethodHandlerFactory` generates the methods which redirect the calls to Piccola and pass the result returned by the service back to Java as sketched in Section 5.2.2.

Once the class is fully generated, we transform it to binary form and either save it to disk or load it into the Java Virtual Machine, using an instance of the `PiccolaClassLoader`. Apart from actually loading the class, our class loader contains a cache which stores created classes.

5.4 Piccola Part of the Language Extension

In this section we illustrate how we use the Java part of the language extension from JPiccola. We distinguish between two type of services (see Table 5.1): services that create a class and services that create an instance of the class generated.

class	*instance*	*description*
`newClassClass`	`newClass`	Generate an arbitrary class
`newInterfaceClass`	`newInterface`	Generate an interface
`newInterfacesClass`	`newInterfaces`	Generate a list of interfaces
`newSubClassClass`	`newSubClass`	Generate a direct subclass

Table 5.1: Services to generate a Java class from JPiccola.

The first type of services generates a Java class of a given type. The second type of services uses the first type of services to generate a class and, additionally, creates an instance of the generated class, builds a form containing the services to map to the Java methods, and associates this form to the object.

5.4.1 Defining Arbitrary Classes

We define two services `newClassClass` and `newClass`. The first service generates the specified class only. The second service creates an object of the specified class and sets the form that maps Java method calls to Piccola services.

Both services allow us to define an arbitrary Java class from JPiccola. The services are Curried, and take four forms as arguments: `name`, `superClassName`, a list of `interfaceNames` and `methods`. The last form is a nested form. Their labels define the name of the method. Bound to these labels are forms containing the `returnType`, the `argumentTypes` and the `body` with the script that is executed when the service is invoked. Optionally, we can define an `initialize` service that is called whenever an object is created. If we create a class only, the `body` form and the `initialize()` service can be omitted.

```
newClass
    name = Name of the class
    superClassName = Name of superclass, default: java.lang.Object
    interfaceNames = [List of interface names, default: ()]
    initialize(): Service mapped to class' constructor, default: ()
    methods =
        <methodName> =
            returnType = Return type, default: void
            argumentTypes = [List of arguments type, default: void]
            body = script of the service, default: ()
        ...
```

```
newClassClass
    name = Name of the class
    superClassName = Name of superclass, default: java.lang.Object
    interfaceNames = [List of interface names, default: ()]
    methods =
        <methodName> =
            returnType = Return type, default: void
            argumentTypes = [List of arguments type, default: void]
        ...
```

The service `newClass` uses first class labels to iterate over all bindings and Piccola inspection to create a list of `MethodDeclaration`s defining the methods to generate. The new class is then generated calling the method `generateClass` of the Java part of the language extension.

An example using the service `newClass` is given in Figure 5.4. We define a subclass of the `java.awt.Frame` that implements the interface `java.awt.event.WindowListener`. All the method names starting with *window...* are necessary to implement this interface. Furthermore, we override the `getTitle()` method of the class `java.awt.Frame` and the `toString()` method of the class `java.lang.Object`.

Note that we use default values, e.g., if we do not define the return type we assume that it is `void`. Still, defining the structure of a class by hand is cumbersome. In the next sections we therefore introduce a set of services that simplifies the generation of a class.

5.4.2 Using Reflection to Facilitate Class Generation

Implementing Interfaces

We can use reflection on an interface to determine which methods we have to implement. We do not have to provide this information from JPiccola. Instead, we define two services: `newInterface` and `newInterfaceClass`.

We use the `newInterface` service to create an instance of a class implementing an interface:

```
newInterface
    className = Name of interface
    methods =
         Services which handle method calls.
```

We can omit the `methods` bindings when we generate a class using the `newInterfaceClass` service:

```
newInterfaceClass
    className = Name of interface
```

```
generateBlueFrame():
    newClass
        name = "BlueFrame3"
        superClassName = "java.awt.Frame"
        interfaceNames = [ "java.awt.event.WindowListener" ]
        methods =
            getTitle =
                returnType = Type.String
                body(args): "a" + args.getClass().getName()
            getBackground =
                returnType = "java.awt.Color"
                body(args): Color.Blue
            windowActivated =
                argumentTypeList = [ "java.awt.event.WindowEvent" ]
                body(args): ()
            windowDeactivated =
                argumentTypeList = [ "java.awt.event.WindowEvent" ]
                body(args): ()
            windowClosing =
                argumentTypeList = [ "java.awt.event.WindowEvent" ]
                body(args): args.at(1).getWindow().dispose()
            windowClosed =
                argumentTypeList = [ "java.awt.event.WindowEvent" ]
                body(args): println "Closed"
            windowOpened =
                argumentTypeList = [ "java.awt.event.WindowEvent" ]
                body(args): println "Opened"
            windowIconified =
                argumentTypeList = [ "java.awt.event.WindowEvent" ]
                body(args): ()
            windowDeiconified =
                argumentTypeList = [ "java.awt.event.WindowEvent" ]
                body(args): ()
            toString =
                returnType = Type.String
                body(args): "This: " + args.getClass().getName()
```

Figure 5.4: Using the service `newClass` to create an instance of a class that subclasses from `java.awt.Frame` and implements the `java.awt.event.WindowListener` interface.

The two services use the `generateInterface` method of the Java part of the language extension to implement one interface. Additionally, we define a service `newInterfaces` that takes a list of interfaces to generate a class that implements this list of interfaces using the `generateInterfaces` method of the Java part of the language extension.

The example given next implements the `java.awt.event.KeyListener` interface which defines three methods. The services simply notify keyboard events by printing a message.

```
generateKeyListener():
    newInterface
        className = "java.awt.event.KeyListener"
        methods =
            keyPressed(args):  println "Key Pressed"
            keyReleased(args): println "Key Released"
            keyTyped(args):    println "Key Typed"
```

Implementing Direct Subclasses

We define two services to override methods of a direct superclass. These services provide a concrete class of an abstract superclass. To create an object of a subclass, we use the `newSubClass` service:

```
newSubClass
    className = Name of superclass
    methods =
        Services which handle method calls.
```

We use the `newSubClassClass` service to generate a direct subclass of a class:

```
newSubClassClass
    className = Name of superclass
```

These services use the `generateSubClass` method of the Java part of the language extension to generate a class.

In the next example instead of implementing the interface `java.awt.event.KeyListener` we subclass from the abstract adapter class `java.awt.event.KeyAdapter`. The object created behaves exactly like the example given in the previous section.

```
generateKeyAdapter():
    newSubClass
        className = "java.awt.event.KeyAdapter"
        methods =
            keyPressed(args):  println "Key Pressed"
            keyReleased(args): println "Key Released"
            keyTyped(args):    println "Key Typed"
```

Chapter 6

From White-Box to Black-Box Components

In this chapter we use the language extension presented in the previous chapter to script components of real-world Java frameworks. The language extension enables us to access the functionality provided by the frameworks using inheritance.

We define components, services, and compositional styles on top of them. We show how we can turn white-box components of Java into black-box components in Piccola.

This chapter is structured as follows: in Section 6.1 we present a service to print text based on the Java printing framework. We implement an interface and provide a service to layout the text which the printing system calls to render a page. In Section 6.2 we use the language extension to script an event-based parser framework and present a service that enables us to declaratively define the events and tags we are interested in. To display HTML documents, we illustrate how we can script two browser components in Section 6.3. In Section 6.4 we investigate the GUI event model of Java and introduce an event composition style. We demonstrate how we can get from a low level of wiring events to components to a high level style that abstracts from the underlying Java event model. In the last Section 6.5 we discuss characteristics of good frameworks and the role of Java interfaces in Java frameworks.

6.1 Printing from JPiccola

6.1.1 The Java Printing API

The `java.awt.print` package provides classes and interfaces for printing. Using this package, we can format pages and layout their content. We can obtain information about the current printer and print to it. A printing dialog supplied by the package allows a user to change print settings.

An instance of the `Book` class represents the document to be printed. We pass it to a `PrinterJob` object which controls printing. We call its methods to set up a job, to optionally present a print dialog to the user, and to print the document.

The Java printing system uses a *callback* model. We have to provide information about the document to the printing system that calls the document to render itself when printing. More concretely, the printing systems calls the `print` method of the `Printable` interface to render a part or all of a page. Therefore, if we create a `Book` document we must supply one or more instances of the `Printable` interface. Using our language extension, this is done easily.

6.1.2 A Text Printing Service

We define a service to print text. We pass the text and optionally the name of the file which we print in the header of the page. Additionally, we control whether we either want a dialog to set page format and printer or choose to print directly.

```
printText
    text = The text to print
    fileName = The name of the file, default: ""
    showFormatDialog = true | false
    showPrintDialog = true | false
```

The interesting part of this service is the creation of the document containing our text. We implement the `print` method of the `java.awt.print.Printable` interface by using the `newInterface` service. We map this method to a service `printText` which adds a header with the name of the file and layouts the text by taking care of word wraps.

```
'getBook(X):
    'aDocument =
        newInterface
            className = "java.awt.print.Printable"
            methods =
                print(args): printText
                    args
                    fileName = X.fileName
                    text = X.text
    'aBook = Host.class("java.awt.print.Book").new()
    aBook.append
        val1 = aDocument
        val2 = X.format
    aBook
```

Note, that if we want to print a document consisting of different content, e.g., geometrical forms, we just map the `print` method to another service. This gives us great flexibility. We can even extend our service to take the service that renders the document as an argument.

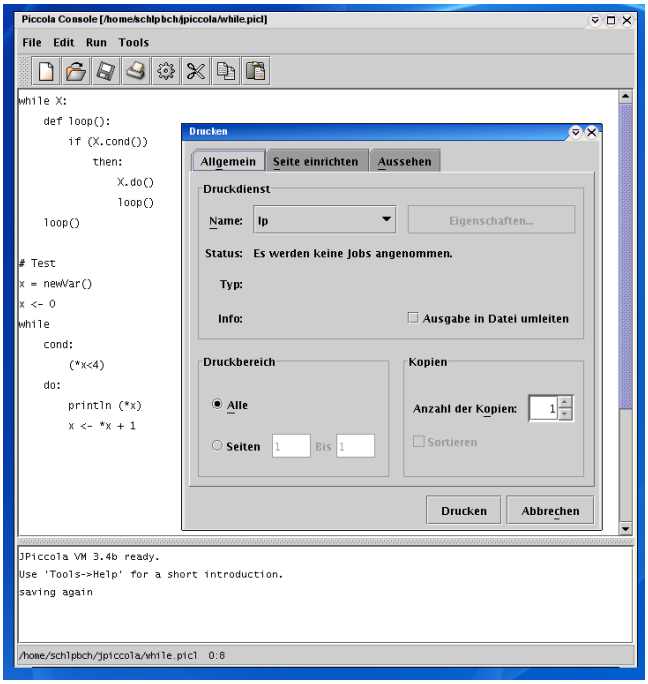

Figure 6.1: A print service in action.

We have written a simple text editor in JPiccola composed of Swing components. We use the `printText` service here to print the file we are working on. Figure 6.1 shows the dialog provided by the Java printing package that enables us to choose printer and format.

6.2 Scripting an Event Based Parser

In this section we define a parse service based on an event based parser framework that enables us to declaratively define the events and tags we are interested in.

6.2.1 Simple API for XML (SAX)

The Simple API for XML (SAX) is a *de facto* standard for event-based parsing of XML based documents. A SAX parser fires events at various important points when parsing a XML document, e.g., at the start of a document or an element.

To be notified of parse events, we have to register event handlers which have to implement Java interfaces. The most important interface is the `ContentHandler` interface that receives notification of the logical content of a document. Other important event interfaces are the `DTDHandler` interface that receives notifications of basic DTD-related events and the `ErrorHandler` interface that enables us to implement customized error handling.

Whenever an event occurs, the corresponding method is called. For example, at the start of an element the `startElement` method of the `ContentHandler` interface is called. This method receives the name of the tag and its attributes as arguments. Depending on whether we are interested in this tag or not we react to the event or not.

Implementing this decision naively we write a big switch statement that branches according to the name of the tag. Such an approach however is neither flexible nor does it scale well.

6.2.2 A Declarative Event Based Parser Service

We want to be able to specify declaratively which events and tags we are interested in. We thus define a parse service as follows:

```
parseURI
    uri = A URI that describes the location of the XML document.
    events = Services handling the events we are interested in.
    tags = Services handling the tags we are interested in.
```

This service creates a SAX parser instance and – using our language extension – implements event handlers with default services that we add to the parser. We can override these default services with the services provided in the `events` form.

Additionally, we define a service `handleTag` that maps the name of a tag to a service. This service uses inspection on forms: if the form `tags` contains a service with the name of the tag, we project on it. Otherwise, we ignore the tag.

6.2.3 Parsing an ANT Configuration File

ANT is a Java-based build tool which reads the build instructions from a XML-based configuration file. A configuration file defines a project which consists of a list of targets. A target itself consists of a set of tasks, e.g., instructions to compile all source files of a project.

In the example given in Figure 6.2 we use the `parseURI` service to generate an overview of an ANT configuration file, consisting of the name and version of the project and its targets.

We are only interested in the `startElement` event and in three tags: the `project` tag which describes the project, the `property` tag with the version of the build file and the `target` tag that describes the targets of the file. We use the `handleTag` to map the name of a tag to a service.

```
SAXParser.parseURI
    uri = uri
    events =
        startElement(args):
            SAXParser.handleTag
                tagName = args.at(2)
                attr = args.at(4)
    tags =
        project(X):
            println "Project Name: " + X.attr.getValue("name")
        property(Y):
            if (Y.attr.getValue("name") == "version")
                 then:
                     println "Version: " + Y.attr.getValue("value")
                     println ""
        target(Y):
            println "* Target:      " + Y.attr.getValue("name")
            println "  Description: " + Y.attr.getValue("description")
            println ""
```

Figure 6.2: Using a declarative parsing service to parse an ANT configuration file.

Parsing the ANT configuration file of JPiccola produces a list of all targets and their descriptions:

```
Project Name: JPiccola
Version: 3.4b

* Target:      make
  Description: Performs a release compilation of JPiccola.

* Target:      jar
  Description: Creates the standalone JAR file.

* Target:      test
  Description: Runs JUnit and PiUnit tests.

* Target:      javadoc
  Description: Creates the API documentation.

* Target:      clean
  Description: Removes all files in the build directory.
  ...
```

6.2.4 Generating a Table of Contents of a XHTML Document

The XHTML [Gro02] specification defines a XML conformant subset of HTML, the markup language used to define web pages. In this section we use the previously defined parse service to generate a table of contents of a web page. The table of contents consists of the title of the document defined by the `title` tag and the content of the two top heading tags (`h1` and `h2`).

In the example given in Figure 6.3 we are interested in five parse events: the events `startDocument` and `endDocument` we use to define the start and the ending of a HTML document. The `startElement` and the `endElement` event notifies us of the start and the end of a tag. The `characters` event delivers the content inside of a tag. This event may fire multiple times, therefore we have to collect all the data until we receive the end of element event. At this point we create a hyperlink element and add it to the HTML document.

6.3 Scripting Browser Frameworks

6.3.1 Scripting a Little Help Browser

We have written a little help browser to display simple HTML documents (see Figure 6.4). The component we use to render HTML documents is the `javax.swing.JEditorPane` class which is able to display documents adhering to the HTML 3.2 specification.

To be notified of hyperlink events, e.g., when we hover over a link or click on it, we have to implement the `javax.swing.event.HyperlinkListener` interface. We can do this easily using the `newInterface` service.

The rendering capabilities of this component however are fairly limited, e.g., real-world web pages often do not get displayed correctly. To display these kind of documents, we need a much more elaborate framework.

6.3.2 Scripting a Web Browser Framework

Overview

ICEbrowser[1] is a web browser framework developed for the Java platform. Its architecture relies on dynamically loadable rendering and scripting modules. Because of its modular and flexible design, ICEbrowser can be integrated into any Java application.

The ICEbrowser framework consists of two mandatory components and a number of optional components (see Figure 6.5). The *ICEbrowser core* and the *HTML pilot* components are mandatory. *Pilots* are rendering modules responsible for the rendering of content. They

[1] see: www.icesoft.no

```
attr = newVar()
chars = newVar()

SAXParser.parseURI
    uri = "file:///home/schlpbch/w3docs/xhtml.html"
    events =
        startDocument():
            println "<html><head><body>"
        startElement(args):
            chars.set("")
            attr.set(args.at(4))
        characters(args):
            str = String.new(val1 = args.at(1), val2 = args.at(2),
                             val3 = args.at(3))
            chars.set(chars.get() + str)
        endDocument():
            println "</body></html>"
        endElement(args):
            SAXParser.handleTag
                tagName = args.at(2)
                attr = attr.get()
    tags =
        title():
            println "<b> " + chars.get() + " </b><br>"
        h1(X):
            id =  X.attr.getValue("id")
            println "<a href='" + uri + "#" + id + "'>"
            println chars.get() + "</a><br>"
        h2(X):
            id =  X.attr.getValue("id")
            println "-> <a href='" + uri + "#" + id + "'>"
            println chars.get() + "</a><br>"
```

Figure 6.3: Using a declarative parsing service to parse a XHTML file.

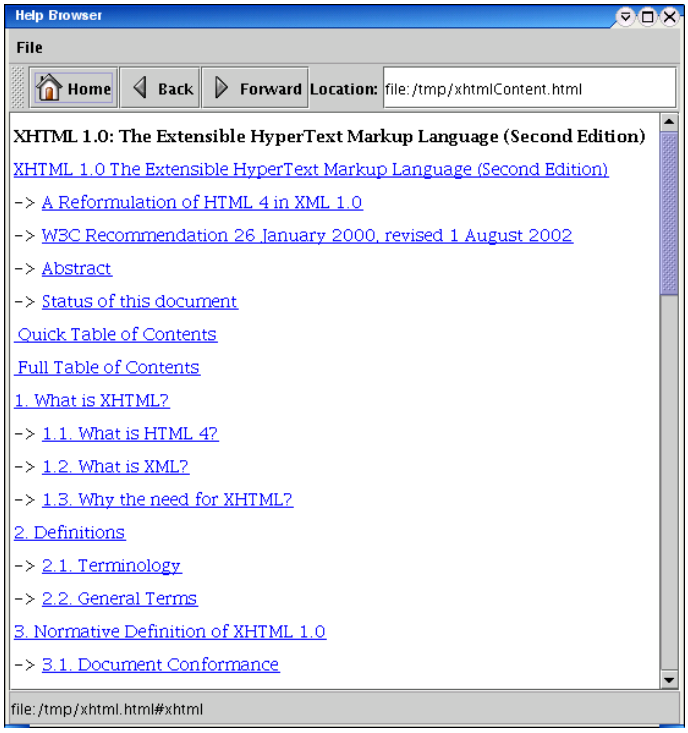

Figure 6.4: A little help browser displaying the table of contents of a HTML page.

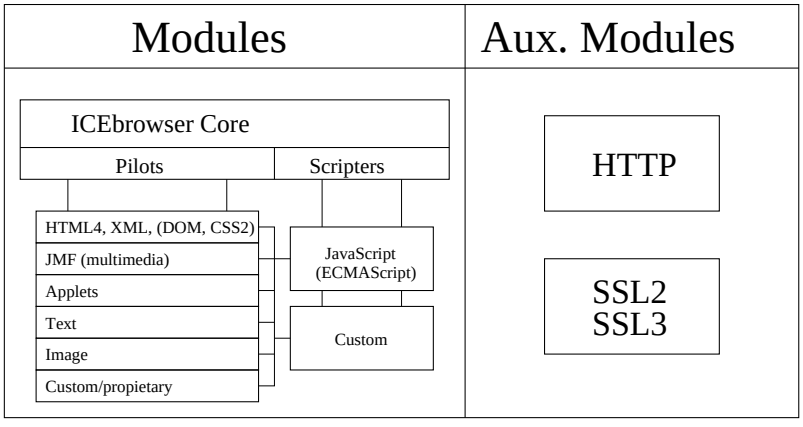

Figure 6.5: Architecture of the ICEbrowser framework.

are similar to the plug-ins of traditional browsers and are compact, platform-independent and can be loaded dynamically. A scripting module, or *scripter*, provides the glue between various pilots and the framework. The scripter knows how to execute commands in its scripting language. It also allows pilots to expose their methods into the scripting environment, without depending on a particular scripter.

The combination of the ICEbrowser core, the HTML pilot, and the JavaScript scripter provides advanced HTML rendering and Web browsing capabilities to applications.

Scripting ICEbrowser components from JPiccola

The `StormBase` class is the core class of this framework. It takes care of all browsing, e.g., it finds and instantiates the right pilot for a given document, keeps track of the URLs visited and displays the content in a AWT component.

To layout pilots, the `StormBase` class provides a method `createTopLevelContainer` that enables us to set a viewport. This method expects an instance of the `ViewportCallback` interface as its argument. We generate this interface using the `newInterface` and return a panel to display its pilots to a `StormBase` object.

Every time the browser core, such as the `StormBase` class or one of its pilots, wants to communicate with the application, it fires events to all event listeners. The events are modeled using the `java.beans` package. An application needs to implement and register a `PropertyChangeListener` interface to a `StormBase` instance to be notified.

In the Piccola code given next, we generate a `PropertyChangeListener` interface and use the `switch` service to react to the events. For example, we set the title of the page in the frame or set the URL of the page being visited in the URL field.

```
createPropertyChangeListener(X):
   newInterface
      className = "java.beans.PropertyChangeListener"
      methods =
         propertyChange(arg):
            event = arg.at(1)
            name = event.getPropertyName()
            val = event.getNewValue()
            switch (name)
               outstandingImages:
                  X.statusBar.setText(""+val+" images to be loaded")
               title:
                  X.frame.setTitle(val)
               location:
                  X.urlField.setText(val)
               default:
                  X.statusBar.setText(val)
```

Figure 6.6: Scripting a web browser using the ICEbrowser framework.

The ICEbrowser framework uses many interfaces to provide hooks for adaptation. We have only introduced the most basic ones, which are however powerful enough to script a browser to access the web (see Figure 6.6). The ICEbrowser framework additionally enables us to plug in scripting engines and printing capabilities using interfaces. And we can access the internal representation of a web page through the *Document Object Model (DOM)* defined by a set of interfaces.

Using the components provided by the ICEbrowser framework is not possible if we can not easily implement interfaces. The `newInterface` service is therefore ideally suited to script this framework from JPiccola.

6.4 A GUI Event Composition Style

6.4.1 (GUI) Event Handling in Java

The Java event model propagates an event from a *source* object to a *listener* object by invoking a method on the listener and passing in the instance of the event subclass that defines the *event* type.

An event source is an object which originates or *fires* events. In an AWT or a Swing application, the event source is typically a GUI component and the listener is commonly an *adapter* object which implements the appropriate listener or set of listeners to control

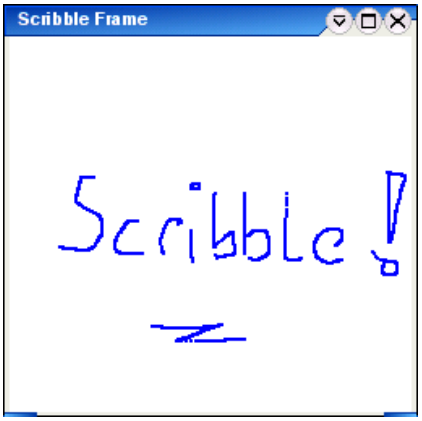

Figure 6.7: A scribble panel

the events. However, this event model is not restricted to GUIs, e.g., it is also used in the JavaBeans framework where JavaBeans components are notified of property changes using events.

Event types are encapsulated in a class hierarchy rooted at the `java.util.EventObject` class. A listener is an object that implements a specific event listener interface extended from the generic `java.util.EventListener`. An event listener interface defines one or more methods which are invoked by the event source when an event of a specific event type occurs.

In order to be notified of events we have to either implement an event listener interface or subclass from an event adapter class. An object of this class can then be added as an event listener to an object interested in this event.

6.4.2 A Scribble Panel – A Low Level Approach

In a first approach we implement a panel we can scribble on on a low level of abstraction (see Figure 6.7, the code is listed in Figure 6.8). Using our language extension, we generate two sorts of listeners: a `WindowListener` that notifies us of window events and a `MouseMotionListener` that informs us of mouse events. We add the window listener to the main frame. When the `windowClose` event is fired, we provide a service to close the frame. The mouse motion listener notifies us when the mouse is dragged in the panel. We get the coordinates of the event and draw a blue point at this position. This gives us a primitive painting program.

We could have achieved the same behavior if we had subclassed from the adapter classes `WindowAdapter` and `MouseMotionAdapter` using the `newSubClass` service. Or we could have generated a class `ScribbleFrame` as a subclass of `Frame` that implements the two interfaces using the `newClass` service.

```
'loadCore "byteCode.picl" root

'Frame = Host.class("java.awt.Frame")
'Panel = Host.class("java.awt.Panel")
'Color = Host.class("java.awt.Color")

WindowListener(aFrame):
    newInterface
        className = "java.awt.event.WindowListener"
        methods =
            windowActivated(args): ()
            windowClosed(args): ()
            windowClosing(args): aFrame.dispose()
            windowDeactivated(args): ()
            windowIconified(args): ()
            windowDeiconified(args): ()
            windowOpened(args): ()

MouseMotionListener(aFrame):
    newInterface
        className = "java.awt.event.MouseMotionListener"
        methods =
            mouseDragged(args):
                event = args.at(1)
                gc = aFrame.getGraphics()
                gc.setColor(Color.blue)
                gc.drawOval
                    val1 = event.getX()
                    val2 = event.getY()
                    val3 = 1
                    val4 = 1
            mouseMoved(args): ()

aFrame = Frame.new("Scribble Frame")

aPanel = Panel.new()
aPanel.setBackground(Color.white)
aFrame.add(aPanel)
aFrame.addWindowListener(WindowListener(aFrame))
aPanel.addMouseMotionListener(MouseMotionListener(aPanel))

aFrame.setSize(val1 = 256, val2 = 256)
aFrame.show()
```

Figure 6.8: Low level GUI event scripting

While this example works, its drawback is that we do not compose components but wire event handlers to components at a low level. Adding new event listeners or changing services reacting to events forces us to rewire the application. In the next section we therefore present a higher-level composition style.

6.4.3 A GUI Event Composition Style

We would like to express the event handling explicitely, abstracting away from the underlying Java GUI event model. We would like to compose *GUI components*, *GUI events* and *event handlers* [AN01] adhering to the style:

GUI Component.on **GUI Event** **Event Handler**

To implement this style we have to take two steps. The first step is to extend each GUI component with a service `on`. This service takes the GUI event and the event handler and registers them to the GUI component:

```
wrapGUIComponent aComponent:
    peer = aComponent.peer
    on anEvent anEventHandler:
        anEvent.register aComponent anEventHandler
```

The second step is to model GUI events as first-class values. An event has to provide a `register` service that enables us to register an event handler to a component. Unfortunately, we can not map the GUI events one-to-one to Java event listeners. For example, the `MouseMotionListener` has two methods: The method `mouseDragged` is called when the mouse is dragged and the method `mouseMoved` is called when the mouse is moved. We want to model calls to these methods as single events.

The listing in Figure 6.9 illustrates the implementation of mouse motion events. To handle these events, we introduce a helper service `MouseMotionEvent` that uses the generic `newEvent` service which returns the Curried service `register` to register the event. We pass the name of the interface class in the form `className`, the Curried service `addListener` that encapsulates the method that adds the listener to the Java GUI component and the `handler` service that adds default services to handle the events. Finally, to model the single mouse motion events as first class values, we use the helper service `MouseMotionEvent`. This service enables us to bind the script that handles an event to the corresponding service.

Our language extension enables us to implement all the event listeners of the AWT and the Swing GUI packages needed for this GUI event composition style dynamically. Previously, we would have had to write all these classes by hand and hardwire them into the JPiccola bridge.

```
newEvent args:                          # General Event
    register aComponent aEventHandler:
        'handler = (args.handler(aEventHandler), initialize(): ())
        'listenerClass = newInterfaceClass
            className = args.className
        aListener = listenerClass.new(handler)
        args.addListener(aComponent) aListener

MouseMotionEvent aSingleHandler:    # Mouse Motion Helper Service
    newEvent
        className = "java.awt.event.MouseMotionListener"
        addListener(aComponent):  aComponent.addMouseMotionListener
        handler(aClosure):
            mouseDragged(): ()
            mouseMoved(): ()
            aSingleHandler(aClosure)

# Single Mouse Motion Event
MouseDragged = MouseMotionEvent(\ X: mouseDragged = X)
MouseMoved = MouseMotionEvent(\ X: mouseMoved = X)
```

Figure 6.9: Mouse Motion Events

6.4.4 A Scribble Panel – Using the Event Composition Style

We now rewrite the scribble application using the newly defined composition style. We extend it with buttons to change the color of the pen and a status bar that displays the position of the mouse pointer (see Figure 6.10). The event and the event handlers are now composed following the style defined in the previous section. We script the GUI components to events and provide services to these events. If they can be expressed in one line of code, we provide anonymous services, otherwise, we use named services.

```
clearPanel(args): ...       # Service that clears the panel.
mouseDragged(args): ...     # Service that draws the points and updates the
                            # status bar with the event's coordinates.

aClearButton.on ActionPerformed clearPanel
aBlueButton.on ActionPerformed (\(): PenColor.set(Color.blue))
aCyanButton.on ActionPerformed (\(): PenColor.set(Color.cyan))
aRedButton.on ActionPerformed (\(): PenColor.set(Color.red))

aFrame.on WindowOpened (\(): aStatusBar.setText("Scribble!"))
aFrame.on WindowClosing (\(): aFrame.dispose())

aPanel.on MouseDragged mouseDragged
```

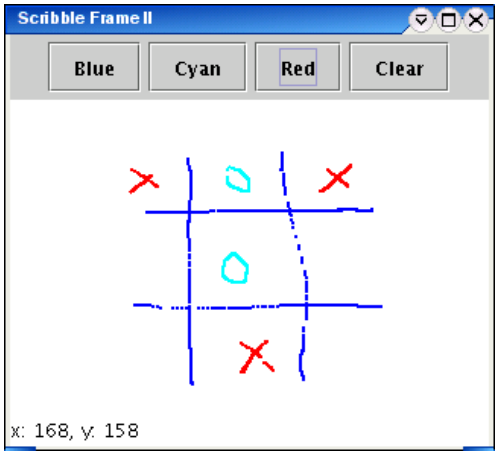

Figure 6.10: An extended scribble panel

Using the event composition style we now abstract from the underlying Java GUI event model and we explicitly state the composition of GUI components, GUI events and the event handlers.

6.5 Discussion

In this chapter we use our language extension to script different real-world Java frameworks. These frameworks force us to use inheritance to access their functionality. We have to provide objects of a specific type to be able to use the frameworks.

We show that our language extension enables us to seamlessly interact with these frameworks. We easily generate the classes required by the frameworks from JPiccola using its reflection based services.

6.5.1 Scriptability of Java Frameworks

If applications are hard to design, and toolkits are harder, then frameworks are hardest of all [GHJV95]. A framework design should be suitable for all applications in the domain. From our point of view, it should even be scriptable. In this section we discuss how difficult it is to script the Java-based frameworks presented in this chapter.

We script the AWT and Swing GUI frameworks easily. Our language extension enables us to define the event handler interfaces entirely in JPiccola and define a compositional style on top of them. GUI components are ideal for scripting. We attribute this to two circumstances: first, the problem domain of building GUI frameworks is well understood and, second, GUI frameworks are designed from the start with reuse in mind.

Similarly, scripting the event-based SAX parser is straightforward. We easily generate the event handler interfaces of the parse events using our language extension and define a service that enables us to declaratively specify the events and tags we are interested in. Parsing is a mature discipline in computer science with considerable knowledge and experience in writing reusable parsing components.

The ICEbrowser framework is a fine example of a framework designed for reuse. Targeted at the embedded market, it provides a set of components that can be combined to form a powerful web browser. Its design uses interfaces to provide hooks for adaptation.

The Java printing package is a good example of an immature framework. Its design is not optimal. It is only possible to script it at a low level, it does not provide any high level printing services, e.g., it is not possible to print simple text using a default service[2].

6.5.2 Properties of Good Frameworks

A framework consists of ready-to-use and half finished building blocks. While most parts of a framework are stable, it has to permit its users to adapt parts of it according to their needs. These parts are the *hot spots* of a framework [Pre94] and good frameworks provide adequate hot spots for adaptation.

Additionally, good frameworks are *black-box frameworks* [JF88]. Black-box frameworks are designed in a way that a user only has to understand its external interfaces to use it. They are characterized by the following properties [FS97]:

Modularity. Frameworks enhance modularity by encapsulating volatile implementation details behind stable interfaces.

Reusability. The stable interfaces provided by frameworks enhance reusability by defining generic components that can be reapplied to create new applications.

Extensibility. A framework enhances extensibility by providing explicit hook methods that allow applications to extend its stable interfaces. Hook methods systematically decouple the stable interfaces and behaviors of an application domain from the variations required by instantiations of an application in a particular context.

Inversion of control. The runtime architecture of a framework is characterized by an *inversion of control.* It enables canonical application processing steps to be customized by event handler objects that are invoked via the framework's reactive dispatching mechanism. When events occur, the framework's dispatcher reacts by invoking hook methods on pre-registered handler objects, which perform application-specific processing on the events. Inversion of control allows the framework to determine which set of application-specific methods to invoke in response to external events. Its user

[2]Its authors acknowledge this shortcoming and a third redesign of this framework is planned.

only provides the code the framework calls. This is often referred to as the *Hollywood Principle*: You do not call the framework, the framework calls you.

6.5.3 Interface versus Implementation Inheritance

Surprisingly, all the examples presented in this chapter use interface inheritance only; we implement interfaces to script the frameworks. We never use implementation inheritance; we do not subclass from any class. Why is this?

In Section 2.2.2 we argue that inheritance serves different roles. One role of inheritance is subclassing, another role is subtyping. These roles can be expressed by different forms of inheritance: implementation inheritance focuses on code reuse while interface inheritance focuses on establishing a type relationship.

The designer of the frameworks provide us interfaces as hooks into their framework. They do not want their frameworks to be used by implementation inheritance, i.e., they do not want us to reuse their code, but they want us to provide objects of a certain type to work with their frameworks. Here, the subtyping role of inheritance is more important than the code reuse role.

6.5.4 Interfaces in Java Frameworks

Some object-oriented languages do not model interfaces explicitely (e.g., C++ or Smalltalk) and their behavior has to be modeled by abstract classes. In contrast, Java supports interfaces explicitely. The separation of interfaces and classes circumvents the problems associated with multiple inheritance as interfaces neither have state nor behavior. An interface only contains declarations of constants and methods and it enables a class to be compatible with multiple interfaces, thus introducing polymorphism.

Java interfaces help to decouple interface and implementation of a framework and improve modularity by encapsulating its internals behind a stable interface. The decoupling of interface and implementation permits its designers to improve the implementation of a framework, as long as their changes do not break the functionality and semantics exported through the interface. This improves reusability and extensibility.

Furthermore, Java interfaces give designers of a framework a means to specify the hot spots of a framework. By defining what methods an interface has to implement they can control the type of adaptions compatible with the framework. Superior to abstract classes, interfaces however do not impose a specific inheritance hierarchy. Still, they enable a compiler to type check whether the adaptation you provide is compatible to the framework. In a nutshell, interfaces guarantee that when the framework calls you, you are able to pick up the phone.

Chapter 7

Conclusion

7.1 Summary

In this master's thesis we present a language extension that enables white-box reuse in a pure composition language. It permits us to reuse components of object-oriented frameworks by inheritance in a composition language at a high level of abstraction. We get the power of inheritance combined with the ease of scripting. Our language extension enables us to minimize the use of inheritance and helps avoid to the problems associated with it. We thus can migrate from class inheritance – a white-box form of reuse – to component composition as a black-box form of reuse.

The language extension extends JPiccola, the Java-based implementation of Piccola. Piccola is a small, pure, general purpose composition language based on a small set of abstractions: forms, agents and channels. Forms are extensible records unified with services. Agents are autonomous entities that communicate by sending information along channels. Based on these primitives we can define high-level composition abstractions.

If we want to use components or frameworks written in Java from JPiccola, we often have to provide objects of a certain type. We need a means to dynamically generate classes of this type. Using our language extension we can easily generate these classes and interfaces from JPiccola.

The key idea of the language extension is to generate Java classes whose objects delegate all calls to their methods to Piccola services. This approach enables us to reuse Java components by inheritance and to define their behavior entirely in Piccola.

Contrast this flexibility to the previous approach: we had to leave JPiccola, write a bridging class in Java, compile it, add it to the bridge of JPiccola and restart JPiccola. Now, we define the class we need in a few lines of code from JPiccola.

The implementation of the language extension consists of two parts: the Java part which is responsible for generating a class and loading it into the Java Virtual Machine using a

custom class loader and the Piccola part which is responsible for providing seamless access to the functionality of the Java part.

To validate the usefulness of our approach, we use the language extension to script various Java frameworks. We show how we can use Java objects as components in JPiccola, define services using them and build compositional styles on top of them.

Our experiments convince us of the usefulness of our approach. It enables us to turn black-box components into white-box components. Furthermore, the language extension smoothly integrates into JPiccola and greatly increases the number of components easily scriptable from JPiccola.

7.2 Lessons Learned

7.2.1 Defining Arbitrary Classes is Seldom Needed

Our language extension provides three sorts of services to generate a class (see Section 5.4). The first `newClass` service enables us to define arbitrary classes. The two other services are reflection based: the `newSubClass` service enables us to override all public methods of a class and the `newInterface` service enables us to implement an interface.

Using the `newClass` service, we can define the structure of a class completely by hand. When experimenting with this service we were able to build up entire inheritance hierarchies, repeatedly subclassing from generated classes. Technically, this works flawlessly.

However, in the context of our work, we never needed this flexibility. If we reuse a class by inheritance, the structure of a class is given. We do not have to generate a class with its own set of methods. To define the structure of a class by hand is cumbersome and unnecessary. We therefore introduce reflection based services that infer the structure of a class we want to subclass.

Of the two reflection-based services, we use the `newSubClass` service to generate a concrete subclass of an abstract base class. However, we are not particularly interested in subclassing from abstract classes. We do not want to implement an abstract class but want to subclass from a concrete class which provides a certain functionality. We thus rarely use this service.

The most used service is the `newInterface` service. The Java frameworks we scripted use Java interfaces to define which interface its user must implement to reuse it. The reflection-based service is therefore ideally suited for this task.

This result supports our claim made in Section 6.5.4 that Java interfaces are well-suited to specify the hot spots of a framework by providing type information without imposing an inheritance hierarchy on its users.

7.2.2 Scripting and Glueing are Fundamentally Different

Piccola is designed as a small, pure, general composition language. It gains its computational power from the scripted components. These components provide an interface adhering to a compositional style with well defined requirements.

Unfortunately, the underlying host components often do not adhere to a compositional style. We have to define glue code to overcome compositional mismatch and provide an interface compatible to a given style.

The Piccola language is designed to support both the scripting and the gluing task. While using JPiccola to script Java components we felt that Piccola is not equally suited for both tasks.

Scripting is fun

Scripting is a high level form of composing components. The components we script have well-defined interfaces and requirements and adhere to a compositional style. While scripting we abstract away from the implementation of the components and the underlying host language.

Piccola is a powerful scripting language. The functional nature of Piccola enables use to define compositional abstractions easily, e.g., using Curried services or functions as first class objects. The GUI event composition style introduced in Section 6.4 supports this claim: we model GUI components, GUI events and events handlers and compose them according to the style.

Writing glue code is – let's say – different

We use glue code to overcome compositional mismatch and define an interface to a component. When gluing components, we often must deal with very host language specific constructs. In JPiccola, we had to deal with a number of peculiarities of Java, e.g., we encountered components that expected Java arrays as arguments or we had to add type information to resolve ambiguities in method lookup. Furthermore, modeling state in Piccola is not supported directly, but has to be simulated using the `newVar` service.

None of these problems are impossible to solve from JPiccola, but often it is awkward. Writing glue code in Piccola is impaired by its generic, host language independent design. Other scripting languages much more tightly integrate with the underlying host language, i.e., they have support for Java arrays or means to define state built in to the language. This integration simplifies writing glue code.

It would however be unfair to blame Piccola completely for these problems. Often components are just not designed with composition in mind.

7.2.3 Good Components are Hard to Find

One of our goals is to script real world components to see how well our language extension, in particular, and Piccola, in general, is suited for the task of scripting. We therefore looked at *many* Java packages and tried to script them.

Surprisingly, some packages turn out to be difficult, if not impossible to script. Some of the problems we encountered are:

Tight Coupling. Often classes are tightly coupled and interdependent in a way that it is impossible to break them up into reasonable, pluggable components.

Implicit Assumptions. Some of the packages make implicit assumptions that are not explicitly stated anywhere. For example, a variable has to be set for a component to work or a component has to be initialized in a special order. This hardly leads to exchangeable components.

Concurrency Issues. Some packages use threads which can lead to concurrency and synchronization problems as JPiccola itself uses threads extensively. Resolving these problems is often impossible as it is unclear what concurrency principles and patterns the packages uses.

Global State. Some packages depend on a specific global state in order to work. For example, a global variable has to be set and is accessed throughout the package. This complicates the creation of easily pluggable components.

Need to Subclass. We are required to subclass from a class where we are actually only interested in the subtyping role of inheritance only. We discuss this point in more detail in Section 6.5.4.

We feel that these packages are at the state of *white-box frameworks* [JF88] at best. Often extensive reading of the source code is needed to understand how to access the functionality provided by a package. This leads to a steep learning curve and sharply contrasts to *black-box frameworks* where the user only needs to understand the external interfaces to use them.

Furthermore, these problems support our claim that components have to be *designed* to be used in a compositional way [ND95]. Writing a framework of reusable and configurable components is hard and involves much more than bundling a bunch of classes into a package.

7.2.4 Implementation Issues

In order to generate a class we generate byte code using the BCEL library and load it into the Java Virtual Machine using a custom class loader. This has given us a deeper understanding of the Java language and its virtual machine.

Primitive Data types

The critique of primitive data types is not new. There are good arguments to include and good arguments to exclude them from a language. We claim that the lack of primitive types would have simplified the development of the language extension considerably in two major areas:

Class Generation. The class generation is largely complicated by the fact that we have to generate different code depending on whether we deal with object types, primitive types or arrays. A unified object model would have greatly simplified the code generation.

Reflection. We often have to determine whether the type of the result returned by reflection is primitive or not. For example, the return type of the `getReturnType` method of the `method` class is of type `Class`. However a `Class` object can also represent a primitive type. There are nine predefined `Class` objects to represent the eight primitive types and `void`. They are created by the Java Virtual Machine and have the same names as the primitive types that they represent. So in a sense, primitive types are objects too. This is confusing to say the least.

7.3 Future Work

7.3.1 Writing and Identifying Good Components

One of the hardest parts of our work was to find components that we could script. We have looked at many Java packages and found it difficult to determine possible components. Some of the difficulties can be assigned to the problem of defining glue code to overcome compositional mismatch, but we feel that the problem is more fundamental.

When looking at a Java package of classes it is far from clear how to split it up into a set of components. A preliminary attempt is to identify a class with a component. However, this categorization is too simple: possible components can be of arbitrary size and consist of more than one object. A package provides another obvious candidate for a component, but often it is only used as a namespace which does not provide a good component.

Two issues have to be addressed: first, we need more experience and guidelines that help us in scripting frameworks and, second, frameworks need to be designed not only with reuse but also with scripting in mind.

More generally, we need a methodological and cultural change in the way we develop software [NGT92]. A bundle of classes does not make a framework. Designing frameworks is hard. To develop frameworks of reusable components, one must acquire domain knowledge, factor out functionality, anticipate further requirements, develop reusable software components, package the results for future application developers, evaluate the ease with

which new applications can be composed and iterate. We are still far away from such an approach to software engineering.

7.3.2 Compiling Piccola to Java Byte Code

When we started this thesis, we were thinking about compiling forms directly to byte code. JPiccola is slow. We could increase Piccola's execution speed by switching from an interpreted to a compiled form of execution. Furthermore, this switch would enable us to distribute Piccola scripts in a binary form.

However, mapping forms to Java byte code is anything but trivial. The smallest unit the Java Virtual Machine supports is a class. We are thus forced to map Piccola form expressions to Java classes. There are many open questions: what does this mapping exactly look like? What are reasonable units of compilation? How do we infer the types of the underlying host components?

We believe that the lazy evaluation technique developed by members of our group [Sch01, Ach02] can help us to determine reasonable units of compilation. By splitting up services into a purely functional part and a part holding the side effect, we get an initial partitioning: we compile the latter part and postpone the evaluation of the former as long as possible.

Piccola is completely untyped, everything is a form. If we wanted to compile Piccola forms it would be of great value if we knew what sort of types a form expects and what sort it provides. We need a way to infer this information. The compositional types currently being worked on could help us in this task.

A more general question which needs to be addressed is whether the Java Virtual Machine is an optimal virtual machine for Piccola. Even though other functional languages have been successfully mapped to the Java Virtual Machine their implementors often encountered major problems [BKR99]. The Java Virtual Machine is not primarily designed to support languages other than the Java language [Gou01, MG02]. For example, tail calls that are crucial for the efficient implementation of functional languages are missing and have to be simulated using suboptimal strategies [SO01].

7.3.3 Another Implementation Platform

Instead of trying to map Piccola onto Java byte code we could consider using a different implementation platform.

One approach would be to implement Piccola on *Parrot*[1], an interpreter explicitly designed for untyped, dynamic scripting languages. Its architecture is designed to support Perl 6, but its goal is to become the *common language runtime* for dynamic languages and to run byte code compiled from other scripting languages such as Python, Ruby or Tcl.

[1] see: www.parrotcode.org

The Parrot interpreter uses only four data types: native int, native float, string, and PMC (Parrot Magic Cookie). The PMC is a completely abstract data type the designers of Parrot introduced to make it language independent and support untyped languages. A PMC is an object of some type and has a set of function pointers assigned to it that enables it to perform various operations. A PMC can be regarded as an object in an abstract virtual class; the PMC needs a set of methods to be defined in order to respond to method calls.

We believe that a form would map quite naturally to a PMC and that we would benefit in general from Parrot's architecture designed to support Perl. The fact that Parrot is explicitly designed for untyped scripting language makes it an interesting candidate for a host language of Piccola.

Another approach would be to target Microsoft's *Common Language Runtime (CLR)*[2]. While such a move would tie us to one platform, the CLR is specifically designed to serve as a platform for different classes of languages and has explicit support for functional languages [MG02]. Furthermore, the rich type system provided by the *Common Type System (CTS)* should enable us to map Piccola forms quite easily to the CLR.

[2]The CLR is part of Microsoft's .NET framework sketched in Section 2.3.1.

Appendix A

Example of a Generated Class

This appendix illustrates the structure of a generated class. The listings given here define a subclass of the class `A` described in Section 5.2.1. For better understanding, the listing in Section A.1 shows the class written in Java code. However, our language extension actually creates byte code which is given in the listing of Section A.2.

A.1 Java Listing

```
import ch.unibe.piccola.IForm;
import ch.unibe.piccola.bridge.ArgumentList;

public class ImplA extends A {

    private IForm m_delegate;

    // Constructors
    public ImplA() {
    }

    public ImplA(IForm delegate) {
        setHandler(delegate);
    }

    // set form
    public void setHandler(IForm delegate) {
        m_delegate = delegate;
        this.initialize();
    }

    public void initialize() {
        ArgumentList args = new ArgumentList();
        args.add(this);
        getService("initialize").callback(args);
    }
```

```
    // get service method
    protected IForm getService(String name) {
        return m_delegate.project(name);
    }

    // methods that delegate calls to Piccola
    public void a() {
        ArgumentList args = new ArgumentList();
        args.add(this);
        getService("a").callback(args);
    }

    public void b(int i) {
        ArgumentList args = new ArgumentList();
        args.add(this);
        args.add(new Integer(i));   // convert 'int' to 'Integer'
        getService("b").callback(args);
    }

    public void c(Object obj) {
        ArgumentList args = new ArgumentList();
        args.add(this);
        args.add(obj);
        getService("c").callback(args);
    }

    public void d(Object obj1, Object obj2, Object obj3) {
        ArgumentList args = new ArgumentList();
        args.add(this);
        args.add(obj1);
        args.add(obj2);
        args.add(obj3);
        getService("d").callback(args);
    }

    public int e() {
        ArgumentList args = new ArgumentList();
        args.add(this);
        // convert 'Integer' to 'int'
        return ((Integer) getService("e").callback(args)).intValue();
    }

    public int f(int i, int j) {
        ArgumentList args = new ArgumentList();
        args.add(this);
        args.add(new Integer(i));
        args.add(new Integer(j));
        return ((Integer) getService("f").callback(args)).intValue();
    }
}
```

A.2 Byte Code Listing

```
Compiled from <schlpbch>
public class ch/unibe/piccola/bridge/GenA extends A
    public ch/unibe/piccola/bridge/GenA();
    public ch/unibe/piccola/bridge/GenA(ch.unibe.piccola.IForm);
    public void setHandler(ch.unibe.piccola.IForm);
    protected ch.unibe.piccola.IForm getService(java.lang.String);
    public void a();
    public void b(int);
    public void c(java.lang.Object);
    public void d(java.lang.Object, java.lang.Object, java.lang.Object);
    public int e();
    public int f(int, int);


Method ch/unibe/piccola/bridge/GenA()
   0 aload_0
   1 invokespecial #10 <Method A()>
   4 return

Method ch/unibe/piccola/bridge/GenA(ch.unibe.piccola.IForm)
   0 aload_0
   1 invokespecial #10 <Method A()>
   4 aload_0
   5 aload_1
   6 invokevirtual #18 <Method void setHandler(ch.unibe.piccola.IForm)>
   9 return

Method void initialize()
   0 new #22 <Class ch.unibe.piccola.bridge.ArgumentList>
   3 dup
   4 invokespecial #23 <Method ch.unibe.piccola.bridge.ArgumentList()>
   7 astore_1
   8 aload_1
   9 aload_0
  10 invokevirtual #27 <Method void add(java.lang.Object)>
  13 aload_0
  14 ldc #29 <String "initialize">
  16 invokevirtual #33 <Method ch.unibe.piccola.IForm getService(java.lang.String)>
  19 aload_1
  20 invokeinterface (args 2) #39 <InterfaceMethod java.lang.Object
                                   callback(ch.unibe.piccola.bridge.ArgumentList)>
  25 pop
  26 return

Method void setHandler(ch.unibe.piccola.IForm)
   0 aload_0
   1 aload_1
   2 putfield #42 <Field ch.unibe.piccola.IForm m_delegate>
   5 aload_0
   6 invokevirtual #44 <Method void initialize()>
   9 return

Method ch.unibe.piccola.IForm getService(java.lang.String)
```

```
   0 aload_0
   1 getfield #42 <Field ch.unibe.piccola.IForm m_delegate>
   4 aload_1
   5 invokeinterface (args 2) #48 <InterfaceMethod ch.unibe.piccola.IForm
                                   project(java.lang.String)>
  10 areturn

Method void a()
   0 new #22 <Class ch.unibe.piccola.bridge.ArgumentList>
   3 dup
   4 invokespecial #23 <Method ch.unibe.piccola.bridge.ArgumentList()>
   7 astore_2
   8 aload_2
   9 aload_0
  10 invokevirtual #27 <Method void add(java.lang.Object)>
  13 aload_0
  14 ldc #62 <String "a">
  16 invokevirtual #33 <Method ch.unibe.piccola.IForm getService(java.lang.String)>
  19 aload_2
  20 invokeinterface (args 2) #39 <InterfaceMethod java.lang.Object
                                   callback(ch.unibe.piccola.bridge.ArgumentList)>
  25 pop
  26 return

Method void b(int)
   0 new #22 <Class ch.unibe.piccola.bridge.ArgumentList>
   3 dup
   4 invokespecial #23 <Method ch.unibe.piccola.bridge.ArgumentList()>
   7 astore_3
   8 aload_3
   9 aload_0
  10 invokevirtual #27 <Method void add(java.lang.Object)>
  13 aload_3
  14 new #56 <Class java.lang.Integer>
  17 dup
  18 iload_1
  19 invokespecial #65 <Method java.lang.Integer(int)>
  22 invokevirtual #27 <Method void add(java.lang.Object)>
  25 aload_0
  26 ldc #67 <String "b">
  28 invokevirtual #33 <Method ch.unibe.piccola.IForm getService(java.lang.String)>
  31 aload_3
  32 invokeinterface (args 2) #39 <InterfaceMethod java.lang.Object
                                   callback(ch.unibe.piccola.bridge.ArgumentList)>
  37 pop
  38 return

Method void c(java.lang.Object)
   0 new #22 <Class ch.unibe.piccola.bridge.ArgumentList>
   3 dup
   4 invokespecial #23 <Method ch.unibe.piccola.bridge.ArgumentList()>
   7 astore_3
   8 aload_3
   9 aload_0
  10 invokevirtual #27 <Method void add(java.lang.Object)>
  13 aload_3
  14 aload_1
```

```
  15 invokevirtual #27 <Method void add(java.lang.Object)>
  18 aload_0
  19 ldc #51 <String "c">
  21 invokevirtual #33 <Method ch.unibe.piccola.IForm getService(java.lang.String)>
  24 aload_3
  25 invokeinterface (args 2) #39 <InterfaceMethod java.lang.Object
                                   callback(ch.unibe.piccola.bridge.ArgumentList)>
  30 pop
  31 return

Method void d(java.lang.Object, java.lang.Object, java.lang.Object)
   0 new #22 <Class ch.unibe.piccola.bridge.ArgumentList>
   3 dup
   4 invokespecial #23 <Method ch.unibe.piccola.bridge.ArgumentList()>
   7 astore 5
   9 aload 5
  11 aload_0
  12 invokevirtual #27 <Method void add(java.lang.Object)>
  15 aload 5
  17 aload_1
  18 invokevirtual #27 <Method void add(java.lang.Object)>
  21 aload 5
  23 aload_2
  24 invokevirtual #27 <Method void add(java.lang.Object)>
  27 aload 5
  29 aload_3
  30 invokevirtual #27 <Method void add(java.lang.Object)>
  33 aload_0
  34 ldc #70 <String "d">
  36 invokevirtual #33 <Method ch.unibe.piccola.IForm getService(java.lang.String)>
  39 aload 5
  41 invokeinterface (args 2) #39 <InterfaceMethod java.lang.Object
                                   callback(ch.unibe.piccola.bridge.ArgumentList)>
  46 pop
  47 return

Method int e()
   0 new #22 <Class ch.unibe.piccola.bridge.ArgumentList>
   3 dup
   4 invokespecial #23 <Method ch.unibe.piccola.bridge.ArgumentList()>
   7 astore_2
   8 aload_2
   9 aload_0
  10 invokevirtual #27 <Method void add(java.lang.Object)>
  13 aload_0
  14 ldc #54 <String "e">
  16 invokevirtual #33 <Method ch.unibe.piccola.IForm getService(java.lang.String)>
  19 aload_2
  20 invokeinterface (args 2) #39 <InterfaceMethod java.lang.Object
                                   callback(ch.unibe.piccola.bridge.ArgumentList)>
  25 checkcast #56 <Class java.lang.Integer>
  28 invokevirtual #60 <Method int intValue()>
  31 ireturn

Method int f(int, int)
   0 new #22 <Class ch.unibe.piccola.bridge.ArgumentList>
   3 dup
```

```
 4 invokespecial #23 <Method ch.unibe.piccola.bridge.ArgumentList()>
 7 astore 4
 9 aload 4
11 aload_0
12 invokevirtual #27 <Method void add(java.lang.Object)>
15 aload 4
17 new #56 <Class java.lang.Integer>
20 dup
21 iload_1
22 invokespecial #65 <Method java.lang.Integer(int)>
25 invokevirtual #27 <Method void add(java.lang.Object)>
28 aload 4
30 new #56 <Class java.lang.Integer>
33 dup
34 iload_2
35 invokespecial #65 <Method java.lang.Integer(int)>
38 invokevirtual #27 <Method void add(java.lang.Object)>
41 aload_0
42 ldc #75 <String "f">
44 invokevirtual #33 <Method ch.unibe.piccola.IForm getService(java.lang.String)>
47 aload 4
49 invokeinterface (args 2) #39 <InterfaceMethod java.lang.Object
                                callback(ch.unibe.piccola.bridge.ArgumentList)>
54 checkcast #56 <Class java.lang.Integer>
57 invokevirtual #60 <Method int intValue()>
60 ireturn
```

Bibliography

[Ach02] Franz Achermann. *Forms, Agents and Channels - Defining Composition Abstraction with Style*. PhD thesis, University of Bern, January 2002.

[AKN00] Franz Achermann, Stefan Kneubuehl, and Oscar Nierstrasz. Scripting coordination styles. In António Porto and Gruia-Catalin Roman, editors, *Coordination '2000*, volume 1906 of *LNCS*, pages 19–35, Limassol, Cyprus, September 2000. Springer-Verlag.

[AN01] Franz Achermann and Oscar Nierstrasz. Applications = components + scripts – a tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[BD98] Boris Bokowski and Markus Dahm. Poor man's genericity for Java. In Clemens Cap, editor, *Proceedings JIT'98*, 1998.

[BKR99] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 129–140, 1999.

[Bos99] Jan Bosch. Superimposition: A component adaptation technique. *Information and Software Technology*, 41(5):257–273, March 1999.

[CHO88] Will Clinger, Anne Hartheimer, and Eric Ost. Implementation strategies for continuations. In *Proceedings of the 1988 ACM conference on LISP and functional programming*. ACM Press, 1988.

[Com02] Apple Computer. The Objective-C programming language, 2002.

[Dah01] Markus Dahm. Byte code engineering with the BCEL API. Technical report, Freie Universität Berlin, Institut für Informatik, 2001.

[Dyb87] R. Kent Dybvig. *The SCHEME programming language*. Prentice Hall, 1987.

[FS97] Mohamed E. Fayad and Douglas C. Schmidt. Object-oriented application frameworks (special issue introduction). *Communications of the ACM*, 40(10):39–42, October 1997.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, Mass., 1995.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification Second Edition*. Addison Wesley, 1996.

[GM96] James Gosling and Henry McGilton. *The Java Language Environment - A White Paper*. Sun Microsystems Computer Company, 1996.

[Gon98] Li Gong. Secure Java class loading. In *IEEE Internet Computing*, volume 2, 1998.

[Gou01] John Gough. Stacking them up: A comparison of virtual machines. 2001.

[Gro02] W3C HTML Working Group. XHTML 1.0 the extensible hypertext markup language (second edition). Technical report, World Wide Web Consortium, 2002.

[GS01] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. *ACM SIGPLAN Notices*, 36(3):248–260, 2001.

[Haa01] Enver Haase. Justice - a free class file verifier for Java. Master's thesis, Freie Universität Berlin, Institut für Informatik, 2001.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[Lau94] Christina Lau. *Object-Oriented Programming Using SOM and DSOM*. Van Nostrand Reinhold, March 1994.

[LB98] Sheng Ling and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings OOPSLA'98*, 1998.

[LP91] Wilf LaLonde and John Pugh. Subclassing $\neq$ subtyping $\neq$ is-a. *Journal of Object-Oriented Programming*, 3(5):57–62, January 1991.

[Lut96] Mark Lutz. *Programming Python*. O'Reilly & Associates, Inc., 1996.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification Second Edition*. Addison-Wesley, 1999.

[MD97] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly, 1997.

[Meu98] Wolfgang De Meuer. The story of the simplest MOP in the world – or – the scheme of object-orientation. In J. Noble, I. Moore, and A. Taivalsaari, editors, *Prototype-based Programming*. Springer-Verlag, 1998.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction, Second edition.* Prentice-Hall, 1997.

[MG02] Erik Meijer and John Gough. Technical overiew of the Common Language Runtime. 2002.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π-Calculus.* Cambridge University Press, 1999.

[MPY01] Erik Meijer, Nigel Perry, and Arjan van Yzendoorn. Scripting .NET using Mondrian. In *Proceedings ECOOP'01*, 2001.

[MS97] Leonid Mikhajlova and Emil Sekerinski. The fragile base class problem and its solution. Technical Report 117, Turku Centre for Computer Science, Turku, Finland, May 1997.

[ND95] Oscar Nierstrasz and Laurent Dami. Component-oriented software technology. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice-Hall, 1995.

[NGT92] Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, September 1992.

[Nie02] Pat Niemeyer. *BeanShell*, 2002.

[NM95] Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a composition language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Langages for Concurrent Systems*, volume 924 of *LNCS*, pages 147–161. Springer-Verlag, 1995.

[NTdMS91] Oscar Nierstrasz, Dennis Tsichritzis, Vicki de Mey, and Marc Stadelmann. Objects + scripts = applications. In *Proceedings, Esprit 1991 Conference*, pages 534–552, Dordrecht, NL, 1991. Kluwer Academic Publishers.

[Ode00] Martin Odersky. Functional nets. In *Proc. European Symposium on Programming*, volume 1782 of *LNCS*, pages 1–25. Springer-Verlag, March 2000.

[Ous98] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

[PR02] Samuel Pedroni and Noel Rappin. *Jython Essentials.* O'Reilly, 2002.

[Pre94] Wolfgang Pree. Meta patterns - A means for capturing the essentials of reusable object-oriented design. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP'94*, LNCS 821, pages 150–162, Bologna, Italy, July 1994. Springer-Verlag.

[PS96] Cuno Pfister and Clemens Szyperski. Why objects are not enough. In *First International Component Users Conference*, 1996.

[Sak92] Markku Sakkinen. The darker side of C++ revisited. *Structured Programming*, 1992.

[Sch99] Jean-Guy Schneider. *Components, Scripts, and Glue: A conceptual framework for software composition*. Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.

[Sch01] Nathanael Schärli. Supporting pure composition by inter-language bridging on the meta-level. Diploma thesis, University of Bern, September 2001.

[SN99] Jean-Guy Schneider and Oscar Nierstrasz. Components, scripts and glue. In Leonor Barroca, Jon Hall, and Patrick Hall, editors, *Software Architectures – Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, pages 38–45, November 1986. Published as Proceedings OOPSLA '86, ACM SIGPLAN Notices, volume 21, number 11.

[SO01] Michel Schinz and Martin Odersky. Tail call elimination on the Java virtual machine. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[SPM02] Jason Smith, Nigel Perry, and Erik Meijer. Mondrian for .NET. 2002.

[Szy98] Clemens Alden Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[Szy02] Clemens Alden Szyperski. Point, counterpoint. *Software Development*, 8(2), February 2002.

[Tai96] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.

[Weg87] Peter Wegner. Dimensions of object-based language design. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, pages 168–182, December 1987. Published as Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, number 12.

[WS96] Wolfgang Weck and Clemens Szyperski. Do we need inheritance? In *Workshop on Composability Issues in Object-Orientation at ECOOP'96*, July 1996.